Habanero-Java: the New Adventures of Old X10

Vincent Cavé
Rice University
vcave@rice.edu

Jisheng Zhao
Rice University
jz10@rice.edu

Jun Shirako
Rice University
shirako@rice.edu

Vivek Sarkar
Rice University
vsarkar@rice.edu

Abstract

In this paper, we present the Habanero-Java (HJ) language developed at Rice University as an extension to the original Java-based definition of the X10 language. HJ includes a powerful set of task-parallel programming constructs that can be added as simple extensions to standard Java programs to take advantage of today's multi-core and heterogeneous architectures. The language puts a particular emphasis on the usability and safety of parallel constructs. For example, no HJ program using `async`, `finish`, `isolated`, and `phaser` constructs can create a logical deadlock cycle. In addition, the `future` and `data-driven task` variants of the `async` construct facilitate a functional approach to parallel programming. Finally, any HJ program written with `async`, `finish`, and `phaser` constructs that is data-race free is guaranteed to also be deterministic.

HJ also features two key enhancements that address well known limitations in the use of Java in scientific computing — the inclusion of complex numbers as a primitive data type, and the inclusion of array-views that support multidimensional views of one-dimensional arrays. The HJ compiler generates standard Java class-files that can run on any JVM for Java 5 or higher. The HJ runtime is responsible for orchestrating the creation, execution, and termination of HJ tasks, and features both work-sharing and work-stealing schedulers. HJ is used at Rice University as an introductory parallel programming language for second-year undergraduate students. A wide variety of benchmarks have been ported to HJ, including a full application that was originally written in Fortran 90. HJ has a rich development and runtime environment that includes integration with DrJava, the addition of a data race detection tool, and service as a target platform for the Intel Concurrent Collections coordination language

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]:

Categories and Subject Descriptors Languages, Design

Keywords Language, Parallel Programming, Data-Race Detection

1. Introduction

The Java language and runtime environment has had a profound worldwide impact on computer software since its introduction nearly two decades ago. It has enabled the creation of a rich ecosystem of libraries, frameworks, and tools that promises to deliver significant value for many years to come. The Java Virtual Machine (JVM) has proved to be a robust platform for portable execution of software on a wide range of hardware platforms, and also capable of executing programs written in other languages (*e.g.,* Scala, Groovy, AspectJ). However, this evolution of the Java ecosystem is in danger of being out-paced by the rapid shift of computer systems to homogeneous and heterogeneous multicore processors. Though the `java.util.concurrent` library offers a rich set of functionalities in support of multicore parallelism, its library interface is primarily accessible to experts and provides a low-level view of parallel programming that consists of threads, tasks, locks, and atomic operations[1]. Further, the lack of closures in Java makes the library interfaces for specifying `Runnable` tasks especially onerous for mainstream programmers.

In this paper, we present the Habanero-Java (HJ) language developed at Rice University during 2007-2010 as a pedagogic extension to the original Java-based definition of the X10 language [10][2]. In addition to its use as a research language in the Rice Habanero Multicore Software research project [19], HJ is used in a new sophomore-level course on "Fundamentals of Parallel Programming" (COMP 322 [1]) which has become a required course for all Computer Science majors at Rice. The HJ programming language is focused on providing safe, usable yet efficient parallel constructs to take advantage of homogeneous and heterogeneous multi-core heterogeneous architectures. HJ language constructs are supported by a compiler and a runtime system. The HJ compiler generates standard Java classfiles that can run on any JVM for Java 5 or higher. The HJ runtime provides both work-sharing and work-stealing scheduling implementations [17]. The work-stealing scheduler supports both work-first and help-first scheduling policies. Since the best policy choice depends on the nature of the program, a third "adaptive" policy is also provided that automatically switches between the two policies to achieve better performance.

HJ offers two main advantages in its use as an introductory parallel programming language for second-year undergraduate students: First, students already know the Java language and are familiar with the Java compiler and runtime tool-chain. They can build

[1] However, these operations can play a critical role in implementing new Java-based parallel programming languages.

[2] See http://x10-lang.org for the latest version of X10.

on this knowledge as Java is essentially a subset of the HJ language. Second, since HJ is a high-level parallel programming language, it makes it easier to focus on general parallel programming concepts, algorithms and patterns without being distracted by low-level details of threads and locks. The COMP 322 course is also a great opportunity to test the usability of HJ and get a sense of how it is perceived by non-experts in parallel programming. The usability of HJ was enhanced by the creation of DrHJ [28], an extension of the DrJava IDE developed at Rice University that supports the HJ language. DrHJ was used in laboratory assignments and programming homeworks for the COMP 322 course at Rice.

The programmability of the HJ language and the efficiency of its runtime performance have been asserted in various contexts. Benchmarks suites such as JGF [20], BOTS [12] and Shootout have been ported to HJ, as well as the full Dipole-1D [22] application from the oil and gas industry that was originally written in Fortran 90. HJ has been used as a target platform for the Intel Concurrent Collections [7] coordination language.

The rest of the paper is organized as follows. Section 2 summarizes the HJ parallel constructs and the complex and array-view language extensions to Java. Section 3 gives an overview of the compiler and runtime system used to implement the HJ language. Section 4 discusses how HJ interfaces with its host platform, as well as some HJ runtime feedback features that aim to improve productivity. Section 5 summarizes our experiences in using HJ as an introductory language for parallel programming in the COMP 322 course. Section 6 summarizes experiences with benchmarks and applications written in HJ. Finally, Sections 7 and 8 discuss related work and our conclusions.

2. The HJ Language

The current HJ implementation supports Java v1.4 as its base language, though sequential code in Java 5/6/7 libraries and classes can be called from HJ programs. (The HJ runtime system is fully compatible with the latest Java release, but the Polyglot-based [26] HJ front-end does not currently support generics; support for Java generics and annotations in the HJ front-end is currently in progress.) The code generated by the HJ compiler consists of Java classfiles that can be executed on any standard JVM.

The HJ extensions to Java are primarily focused on task parallelism. Similar extensions to C and Scala are being pursued in the Habanero C and Habanero Scala projects at Rice. Much of HJ's Java-based runtime system is being reused for Habanero Scala, since Scala programs also execute on a JVM.

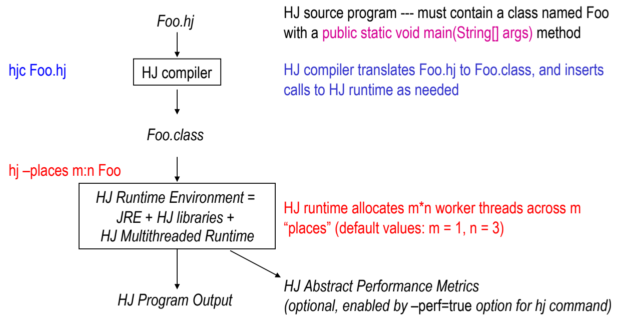
Foo.hj
HJ source program --- must contain a class named Foo with a public static void main(String[] args) method
hjc Foo.hj
HJ compiler
HJ compiler translates Foo.hj to Foo.class, and inserts calls to HJ runtime as needed
Foo.class
hj –places m:n Foo
HJ Runtime Environment = JRE + HJ libraries + HJ Multithreaded Runtime
HJ runtime allocates m*n worker threads across m "places" (default values: m = 1, n = 3)
HJ Program Output
HJ Abstract Performance Metrics (optional, enabled by –perf=true option for hj command)

Figure 1. Habanero-Java (HJ) Compilation and Execution Environment

Figure 1 summarizes the compilation and execution environment for HJ programs. It is similar to that for Java programs. HJ programs are stored in files with a `.hj` extension, and the `hjc` command is used to compile an HJ program. The `hj` command is used to execute an HJ program after it has been compiled. The root file, Foo.hj (say), must contain a class named Foo with a `main()` method. Program execution begins with a single task at the start of `main()`.

2.1 Sequential Extensions to Java

points: A *point* is an element of a k-dimensional Cartesian space ($k \geq 1$) with integer-valued coordinates, where k is the rank of the point. A point's dimensions are numbered from 0 to $k - 1$. The following operations are defined on a point-valued expression `p1`:

- `p1.rank` — returns rank of point `p1`
- `p1.get(i)` — returns element in dimension `i` of point `p1`, or element in dimension $(i \mod p1.rank)$ if $i < 0$ or $i \geq p1.rank$.
- `p1.lt(p2)`, `p1.le(p2)`, `p1.gt(p2)`, or `p1.ge(p2)` returns true if and only if `p1` is lexicographically $<$, $\leq$, $>$, or $\geq$ `p2`. These operations are only defined when `p1.rank` = `p1.rank`.

regions: A k-dimensional *region* is a set of k-dimensional points, defined as a Cartesian product of *low:high* contiguous subranges in each dimension. Thus, $[1 : 10]$ is a 1-dimensional region consisting of the 10 points $[1], \ldots, [10]$, and $[1 : 10, -5 : 5]$ is a 2-dimensional region consisting of 110 points since the first dimension has 10 values $(1 \ldots 10)$ and the second dimension has 11 values $(-5 \ldots 5)$. Likewise, the region `[0:200,1:100]` specifies a collection of two-dimensional points `(i,j)` with `i` ranging from `0` to `200` and `j` ranging from `1` to `100`. Regions are used in HJ to define the range for sequential point-wise `for` and parallel `forall` loops.

pointwise for: A task executes a point-wise `for` statement by sequentially enumerating the points in its region in canonical lexicographic order, and binding the components of the points to the index variables defined in the `for` statement. A convenience relative to the standard Java idiom, "`for (int i = low; i <= high; i++)`", in which the upper bound, `high`, is re-evaluated in each iteration of a Java loop, is that `high` is only evaluated once in an HJ `[low:high]` region expression. Another convenience is that loops can be easily converted from sequential to parallel (or vice versa) by replacing `for` by `forall` (or `forall` by `for`), as we will see in Section 2.2.

complex as a primitive type: HJ supports complex numbers by adding two new primitive types: *complex32* (32-bit single-precision) and *complex64* (64-bit double-precision). These types can be used like any other primitive types in a Java program. The compiler provides support for declaration, assignment, arithmetic operations, conversions, promotions and complex arrays. Complex variables can be declared as constants, local variables and field of objects. A method can take complex numbers as arguments, and also return a complex number as a return value. A complex number is declared by specifying a pair representing the real and the imaginary part. HJ provides *real* and *imag* accessors to retrieve either part from a complex expression. The syntax for arithmetic operations is identical to operations involving regular primitive types.

array-views: Array-views [21] is an abstraction built on top of a one-dimensional array, that lets programmers "view" an array using a custom multidimensional index space. Array-views enable programmers to use arrays in a very flexible way compared to standard Java arrays. For instance, a programmer can create a view over a one-dimensional array and decide that the right way to access elements is through the region [1:N], because it makes more sense to them for the origin to be at index 1 *e.g.,* when porting programs from Fortran. If the programmer wishes, they also create a new view over the same base array using a 2-dimensional region such as [1:sqrtN,1:sqrtN]. An element of an array-view can be accessed using the square bracket notation as in standard Java arrays. In the case of multi-dimensional array views, commas separate indices of each dimension as in Fortran arrays.

2.2 HJ Parallel Extensions

In this section, we briefly summarize the main parallel constructs available in HJ.

async: The statement "**async** ⟨*stmt*⟩" causes the parent task to create a new child task to execute ⟨*stmt*⟩ *asynchronously* (*i.e.,* before, after, or in parallel) with the remainder of the parent task. Figure 2 illustrates this concept by showing a code schema in which the parent task, T_0, uses an `async` construct to create a child task T_1. Thus, `STMT1` in task T_1 can potentially execute in parallel with `STMT2` in task T_0.

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

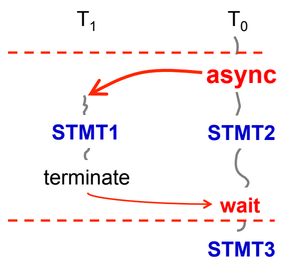

Figure 2. An example code schema with `async` and `finish` constructs

`async` is a powerful primitive because it can be used to enable any statement to execute as a parallel task, including for-loop iterations and method calls. In general, an HJ program can create an unbounded number of tasks at runtime. The HJ runtime system is responsible for scheduling these tasks on a fixed number of processors. It does so by creating a fixed number of *worker threads*, typically one worker per processor core or hardware context. These workers repeatedly pull work from a logical work queue when they are idle, and push work on the queue when they generate more work. The work queue entries can include *asyncs* and *continuations*. An `async` is the creation of a new task, such as T_1 in Figure 2. A continuation represents a potential suspension point for a task, which (as shown in in Figure 2) can include the point after an `async` creation as well as the point following the end of a `finish` scope. Continuations are also referred to as *task-switching* points, because they are program points at which a worker may switch execution between different tasks.

As with Java threads, local variables are *private* to each task, whereas static and instance fields may be *shared* among tasks. An inner `async` is allowed to read a local variable declared in an outer scope. This semantics is similar to that of parameters in method calls — the value of the outer local variable is simply copied on entry to the `async`. However, an inner `async` is not permitted to modify a local variable declared in an outer scope. The ability to read non-final local variables in an outer scope is more general than the standard Java restriction that a method in an inner-class may only read a local variable in an outer scope if its declared to be final.

HJ also supports a `seq` clause for an `async` statement with the following syntax and semantics:
`async seq(cond) <stmt>` ≡
`if (cond) <stmt> else async <stmt>`
The `seq` clause simplifies programmer-controlled serialization of task creation to deal with overhead. It is restricted to cases when no blocking operation such as `phaser next` operations and `future get()` operations is performed inside `<stmt>`. The main benefit of the `seq` clause is that it removes the burden on the programmer to specify `<stmt>` twice with the accompanying software engineering hazard of ensuring that the two copies remain consistent. In the future, the HJ system will explore approaches in which the compiler and/or runtime system can select the serialization condition automatically for any `async` statement.

finish: `finish` is a generalized join operation. The statement "**finish** ⟨*stmt*⟩" causes the parent task to execute ⟨*stmt*⟩ and then wait until all `async` tasks created within ⟨*stmt*⟩ have completed, including transitively spawned tasks. Each dynamic instance T_A of an `async` task has a unique *Immediately Enclosing Finish* (IEF) instance F of a `finish` statement during program execution, where F is the innermost `finish` containing T_A [33]. There is an implicit `finish` scope surrounding the body of `main()` so program execution will only end after all `async` tasks have completed.

Like `async`, `finish` is a powerful primitive because it can be wrapped around any statement thereby supporting modularity in parallel programming. The scopes of `async` and `finish` can span method boundaries in general. As an example, the `finish` statement in Figure 2 is used by task T_0 to ensure that child task T_1 has completed executing `STMT1` before T_0 executes `STMT3`. If T_1 created a child `async` task, T_2 (a "grandchild" of T_0), T_0 will wait for both T_1 and T_2 to complete in the `finish` scope before executing `STMT3`. If you want to convert a sequential program into a parallel program, one approach is to insert `async` statements at points where the parallelism is desired, and then insert `finish` statements to ensure that the parallel version produces the same result as the sequential version.

Besides termination detection, the `finish` statement plays an important role with regard to exception semantics. If any `async` throws an exception, then its IEF statement throws a *MultiException* [10] formed from the collection of all exceptions thrown by all `async`'s in the IEF.

future: HJ also includes support for `async` tasks with return values in the form of `futures`. The statement, "`final future<T> f = async<T> Expr;`" creates a new child task to evaluate `Expr` that is ready to execute immediately. In this case, `f` contains a "`future` handle" to the newly created task and the operation `f.get()` (also known as a *force* operation) can be performed to obtain the result of the `future` task. If the `future` task has not completed as yet, the task performing the `f.get()` operation blocks until the result of `Expr` becomes available. An important constraint in HJ is that *all variables of type* `future<T>` *must be declared with a* `final` *modifier*, thereby ensuring that the value of the reference cannot change after initialization. This rule ensures that no deadlock cycle can be created with `future` tasks. Finally, HJ also permits the creation of `future` tasks with `void` return type; in that case, the `get()` operation simply serves as a join on the `future` task.

phasers: The `phaser` construct [33] integrates collective and point-to-point synchronization by giving each task the option of registering with a `phaser` in *signal-only*/*wait-only* mode for producer/consumer synchronization or *signal-wait* mode for barrier synchronization. These properties, along with the generality of *dynamic parallelism*, *phase-ordering* and *deadlock-freedom* safety properties, distinguish `phasers` from synchronization constructs in past work including barriers [18] and X10's clocks [10]. The latest release of `j.u.c` in Java 7 includes `Phaser` synchronizer objects, which are derived in part [25] from the `phaser` construct in HJ. (The `j.u.c. Phaser` class only supports a subset of the functionality available in HJ `phasers`.)

In general, a task may be registered on multiple `phasers`, and a `phaser` may have multiple tasks registered on it. Three key `phaser` operations are:

• *new:* When a task A_i performs a `new phaser()` operation, it results in the creation of a new `phaser` *ph* such that A_i is registered with *ph* in the *signal-wait* mode (by default).

• *registration:* The statement, `async phased` (*ph1*⟨*mode1*⟩, *ph2*⟨*mode2*⟩, ...) ⟨*stmt*⟩, creates a child task that is registered on phaser *ph1* with *mode1*, `phaser` *ph2* with *mode2*, etc. The child tasks registrations must be subset of the parent task's registrations. `async phased` ⟨*stmt*⟩ simply propagates all of the parents `phaser` registrations to the child.
• *next:* The `next` operation has the effect of advancing each `phaser` on which the invoking task A_i is registered to its next phase, thereby synchronizing all tasks registered on the same `phaser`. In addition, a `next` statement for `phasers` can optionally include a *single* statement, `next` {`S`}. This guarantees that the statement `S` is executed exactly once during the phase transition [33, 41].
foreach: The statement `foreach (point p : R) S` supports parallel iteration over all the points in region `R` by launching each iteration as a separate `async`. A `foreach` statement does not have an implicit `finish` (join) operation, but its termination can be ensured by enclosing it within a `finish` statement at an appropriate outer level. Further, any exceptions thrown by the spawned iterations are propagated to its IEF instance. Thus, we see that `foreach (point p : R) S`(p) is equivalent to a combination of a sequential `for` loop with a local `async` construct for each iteration, which can be written as follows: `for (point p : R) async` {`S`(p)}.
forall: The `forall` construct is an enhancement of the `foreach` construct. The statement `forall (point p : R) S` supports parallel iteration over all the points in region `R` by launching each iteration as a separate `async`, and including an implicit `finish` to wait for all of the spawned `asyncs` to terminate.

Each dynamic instance of a `forall` statement also includes an implicit `phaser` object (let us call it `ph`) that is set up so that all iterations in the `forall` are registered on `ph` in *signal-wait* mode. One way to relate `forall` to `foreach` is to think of `forall` ⟨*stmt*⟩ as syntactic sugar for "`ph=new phaser(); finish foreach phased (ph)` ⟨*stmt*⟩". Since the scope of `ph` is limited to the implicit `finish` in the `forall`, the parent task will drop its registration on `ph` after all the `forall` iterations are created. The `forall` statement was designed for use as the common way to express parallel loops. However, programmers who need to perform finer-grained control over `phaser` registration for parallel loop iterations will find it more convenient to use `foreach` instead.
isolated: The HJ construct, `isolated` ⟨*stmt1*⟩, guarantees that each instance of ⟨*stmt1*⟩ will be performed in mutual exclusion with all other potentially parallel *interfering* instances of `isolated` statements ⟨*stmt2*⟩. Two instances of `isolated` statements, ⟨*stmt1*⟩ and ⟨*stmt2*⟩, are said to interfere with each other if both access the same shared location, such that at least one of the accesses is a write. As advocated in [23], we use the `isolated` keyword instead of `atomic` to make explicit the fact that the construct supports weak isolation rather than strong atomicity. Commutative operations, such as updates to histogram tables or insertions into a shared data structure, are a natural fit for `isolated` blocks executed by multiple tasks in deterministic parallel programs.

The current HJ implementation takes a simple *single-lock* approach to implementing `isolated` statements, by treating each entry of an `isolated` statement as an *acquire()* operation on the lock, and each exit of an `isolated` statement as a *release()* operation on the lock. Though correct, this approach essentially implements `isolated` statements as *critical sections*, thereby serializing interfering and non-interfering `isolated` statement instances.

An alternate approach for implementing `isolated` statements being explored by the research community is *Transactional Memory* (TM) [23]. However, there is as yet no currently available practical TM approach in widespread use. Instead, we encourage students in COMP 322 to explore the use of Java *atomic variables* to replace `isolated` statements with common access patterns by library calls to Java atomic utilities. Thus, atomic variables provided a restricted solution to scalable implementations of `isolated`. If an `isolated` statement matches an available pattern, then it can be implemented by using an atomic variable; otherwise, the default single-lock implementation of `isolated` has to be used instead. Recently, a new implementation technique called *delegated isolation* [24] has been designed and prototyped for HJ `isolated` statements, and shown to be superior to both single-lock and TM approaches in many cases. We expect to include this technique in an HJ release in the near future.

When comparing implementations of `isolated` statements, the three cases to consider in practice can be qualitatively described as follows:

1. *Low contention:* In this case, `isolated` statements are executed infrequently, and a single-lock approach as in HJ is often the best solution. Other solutions, such as TM and atomic variables, incur additional overhead compared to the single-lock approach because of their book-keeping and checks necessary but there is no visible benefit from that overhead because contention is low.
2. *Moderate contention:* In this case, the serialization of all `isolated` statements in a single-lock approach limits the performance of the parallel program due to Amdahl's Law, but a finer-grained approach that only serializes interfering `isolated` statements results in good scalability. Atomic variables usually shine in this scenario since the benefit obtained from reduced serialization far outweighs any extra overhead incurred.
3. *High contention:* In this case, there are phases in the program when interfering `isolated` statements dominate the program execution time. In such situations, approaches such as TM and atomic variables are of little help since they cannot eliminate the interference inherent in the program. The best alternative to `isolated` in such cases is to use reduction primitives such as accumulators [32] instead.

data-driven tasks: Data-Driven Tasks [36] are an extension of `future` tasks in HJ that support the dataflow model. This extension is enabled by adding an `await` clause to the `async` statement as follows:

`async await` ($DDF_a, DDF_b, \cdots$) ⟨ `statement` ⟩

Each of $DDF_a, DDF_b, \cdots$ is an instance of the HJ standard class `DataDrivenFuture`. A DDF acts as a container for a single-assignment value, like regular `future` objects. However, unlike `future` objects, DDF's can be used in an `await` clause, and any `async` task can be a potential producer for a DDF (though only one task can be the actual producer at runtime because of the single-assignment property).

Specifically, the following rules apply to `DataDrivenFuture` objects:

- A variable of type `DataDrivenFuture` is a reference to a DDF object. Unlike regular `future` objects, there is no requirement that variables of type `DataDrivenFuture` be declared with a `final` modifier.
- The following four operations can be performed on variables of type `DataDrivenFuture`:
  1. *Create* — an instance of a DDF container can be created using the standard Java statement, `new DataDrivenFuture()`.
  2. *Produce* — any task can provide a value, V, for a DDF container by performing the operation, DDF.`put`(V). After the `put` operation, the DDF's value is said to become *available*. If another `put` operation is attempted on the same DDF, the

second put will throw an exception because of its violation of the single-assignment rule.

3. *Await* — a new `async` task can be created with an `await` clause, `await` $(DDF_a, DDF_b, \cdots)$. The task will only start execution after all the DDFs in the `await` clause become available.

4. *Consume* — any task that contains a DDF in its `await` clause is permitted to perform a `get()` operation on the DDF. A *cast* operation will be required to cast the result of `get()` to the expected object type. If a `get()` operation is attempted by a task that has no `await` clause or does not contain the DDF in its `await` clause, then the `get()` will throw an exception. Thus, a `get()` on a DDF will never lead to a blocking operation (unlike a `get()` on a regular `future` object).

- It is possible for an instance of an `async` with an `await` clause to never be enabled, if one of its input DDF never becomes available. This can be viewed as a special case of *deadlock*. However, this deadlock case is resolved by ensuring that each `finish` construct moves past the end-finish when all enabled `async` tasks in its scope have terminated, thereby ignoring any remaining non-enabled `async` tasks.

places: The *place* construct in HJ provides a way for the programmer to specify affinity among `async` tasks. A place is an abstraction for a set of worker threads. When an HJ program is launched with the command, "`hj -places p:w`", a total of $p \times w$ worker threads are created with w workers per place. The places are numbered in the range $0 \ldots p-1$ and can be referenced in an HJ program, as described below. The number of places remains fixed during program execution; there is no construct to create a new place after the program is launched. This is consistent with other runtime systems, such as OpenMP, CUDA and MPI, that require the number of threads/processes to be specified when an application is launched. However, the management of individual worker threads within a place is not visible to an HJ program, giving the runtime system the freedom to create additional worker threads in a place, if needed, after starting with w workers per place.

The main benefit of use $p > 1$ places is that an optional `at` clause can be specified on an `async` statement or expression of the form, "`async at(`*place-expr*`)` ...", where *place-expr* is a place-valued expression. This clause dictates that the child `async` task can only be executed by a worker thread at the specified place. Data locality can be controlled by assigning two tasks with the same data affinity to execute in the same place. If the `at` clause is omitted, then the child task is scheduled by default to execute at the same place as its parent task. The main program task is assumed to start in place 0.

Thus, each task has a designated place. The value of a task's place can be retrieved by using the keyword, `here`. If a program only uses a single place, all `async` tasks just run at place 0 by default and there is no need to specify an `at` clause for any of them. The current release of HJ supports a flat partition of tasks into places. Support for hierarchical places [40] will be incorporated in a future release.

2.3 HJ Code Examples

In this section, we briefly discuss two HJ code examples to illustrate some of the constructs described in this section.

Figure 3 shows a code fragment from the BOTS Health benchmark [12] rewritten in HJ. The `async seq` construct in line 5-7 executes the function, `sim_village_par(v)`, sequentially if condition `(sim_level - village.level >= bots_cutoff_value)` is true, otherwise it creates a child task to invoke `sim_village_par(v)`.

```
  void sim_village_par(Village village) {
   // Traverse village hierarchy
1: finish {
2:   Iterator it=village.forward.iterator();
3:   while (it.hasNext()) {
4:     Village v = (Village)it.next();
5:     async seq ((sim_level - village.level)
6:                >= bots_cutoff_value)
7:       sim_village_par(v);
     } // while
8:   ... ...;
9:  } // finish:
10:   ... ...  }
```

Figure 3. Code fragment from BOTS Health benchmark written in HJ

```
finish {
  delta.f = epsilon+1; iters.i = 0;
  phaser ph = new phaser();
  accumulator ac = new accumulator(ph, ph.SUM,
                                   int.class);
  foreach phased ( point[j] : [1:n]) {
    // Each iteration will be registered on ph
    while ( delta.f > epsilon ) {
      newA[j] = (oldA[j-1]+oldA[j+1])/2.0f;
      diff[j] = Math.abs(newA[j]-oldA[j]);
      ac.send(diff[j]);
      // Local work can be overlapped with
      // accumulator operations
      . . .
      next {
        // barrier w/ single statement
        delta.f = ac.result(); iters.i++;
      }
      temp = newA; newA = oldA; oldA = temp;
    } // while
  } // foreach
} // finish
System.out.println("Iterations: " + iters.i);
```

Figure 4. One-Dimensional Iterative Averaging using HJ Phasers and Accumulators

As a result, multiple child tasks created in multiple iterations can execute in parallel with the parent task. The parent task waits at the end of line 9 for all these child tasks to complete since the scope of the `finish` construct in this code fragment ends at line 9.

Figure 4 contains a one-dimensional iterative averaging example program that uses HJ `phasers` and accumulators [32]. The `ac.send()` call enables each `foreach` iteration to send a local value as input to an asynchronous sum reduction operation. The result of the sum is accessed in the call to `ac.result()` in the next-with-single statement. This approach gives the HJ system flexibility on how best to implement the `send()` and `result()` operations. Note that `ac.result()` is only computed once per loop iteration since it is included in a single statement [33]. Also, if available, local work can be overlapped with the `ac.send()` accumulator operations as indicated by the ... code region in Figure 4.

3. HJ Compiler and Runtime Implementation

As with Java, the HJ tool-chain consists of a compilation command and an execution command. The HJ compiler (hjc) processes .hj files as input and produces standard Java classfiles as output. The HJ runtime command (hj) starts a Java virtual machine and executes the main() method of the HJ runtime system, which in turn initializes the HJ runtime and instantiates the application to be run by using reflection.

3.1 The Habanero-Java compiler

The Habanero-Java compiler (hjc), presented in Figure 5, is written in Java and is composed of two major components. The Polyglot-based front-end parses HJ source code to create an abstract syntax tree (AST). The Soot-based back-end operates on a three-address intermediate representation and generates classfiles as output.

Polyglot [26] is a highly extensible source-to-source translator framework for the Java programming language. It aims at facilitating the creation of language extensions for Java by providing a framework that allows parsing and creation of an AST representation of a Java program. The Soot optimization framework [37] provides several intermediate representations for analyzing and transforming Java bytecode. All the HJ compiler analyses and transformations are performed on the Jimple intermediate representation (IR), which is a typed 3-address IR. One advantage of operating at the IR-level, as opposed to source code, is that it offers the possibility of generating low-level control flow with goto's and labels (a capability that is required to support code generation for work-stealing task schedulers).

The HJ compiler uses the LPG parser to parse the HJ grammar (which extends the Java 1.4 grammar) and extends Polyglot to handle HJ constructs by creating new AST nodes to represent them. The front-end performs semantic checks for the programming constructs introduced in Section 2. For instance, it checks that a task does not write into local variables belonging to a parent/ancestor task. Once a program has successfully passed both the syntactic and semantic analyses, the Polyglot AST is transformed into the Jimple IR and handed to the Soot back-end. The Jimple IR is extended to a parallel intermediate representation (PIR) [42] which features new Jimple IR nodes to represent HJ constructs. The back-end then gradually applies a series of transformations to go from the high-level PIR to lower-level representations. We can roughly classify them as high-level, mid-level and low-level variants of the PIR. The high-level PIR is hierarchical in structure and useful for performing compiler analyses such as May-Happen-in-Parallel (MHP) analysis [3]. The mid-level PIR has a flat control flow structure and is useful for performing optimizations such as Load Elimination [5] that build on classical data flow frameworks. For example, a `forall` loop is kept intact in the high-level PIR but expanded into a for-loop with `async` statements in the mid-level PIR. All HJ constructs are later expanded into standard Java operations in the low-level PIR, with calls to the HJ runtime library inserted as needed. For instance, an `async` is transformed into an instance of an anonymous inner class (closure) that can be passed to the HJ task scheduler. Most traditional Java optimizations can be performed at the low-level PIR level.

3.2 The Habanero-Java Runtime System

The HJ runtime system provides support for many of the HJ constructs described in Section 2. It is also responsible for orchestrating the creation, scheduling, execution and termination of tasks. The core of the runtime scheduler provides support for the finish/async model, which includes tracking the Immediately Enclosing Finish (IEF) scope for each dynamic task. Constructs such as `phasers` and `data-driven tasks` also need runtime support to maintain their internal data structures and keep track of synchronization status.

The HJ runtime has a set of worker threads that can execute tasks. When the hj command is invoked, control is transferred to the HJ runtime which wraps the main method of the application to be executed in a `finish` and starts executing it. Whenever a new task is created, the runtime associates the task with the current `finish` scope and increments its `finish` state counter. The newly created task is then scheduled for execution. At this point, no assumptions are made as to where and when the task will be executed. The runtime maintains a stack of `finish` scopes. When a task waits at the end of a `finish`, it only needs to wait for the `finish` counter to reach zero since that indicates that all (transitively) spawned child tasks in the finish scope have terminated.

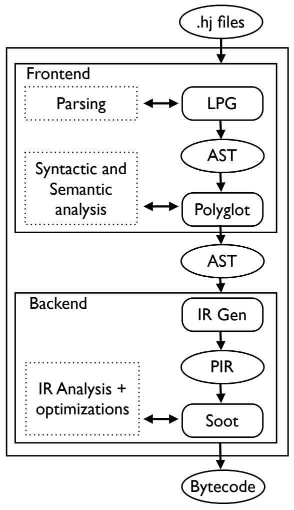

Figure 5. Architecture of the HJ compiler (hjc).

With this basic support for the finish/async model in place, the runtime has several options as to how the tasks should be scheduled on workers. The HJ runtime includes both work-sharing and work-stealing task schedulers. The work-sharing scheduler supports the full HJ language but relies on a single queue, from where workers put and get tasks; this scheduler may create new workers in a place if needed. The work-stealing implementation relies on a distributed deque implementation and includes support to alleviate the cost of blocking operations such as `finish` without creating new workers; however, this scheduler only supports a subset of HJ's parallel constructs (primarily `async`, `finish` and `isolated`). The decision of which policy to use must be made at compile time since the generated bytecode is tailored to the appropriate scheduler (work-sharing is the default). Note that an HJ program can be recompiled for either scheduling policy without requiring source code modifications. Further details on the HJ work-sharing and work-stealing schedulers can be found in [17].

4. HJ Runtime Deployment and Feedback Tools

The tool-chain for HJ programs is similar to that of Java. An HJ distribution includes a compiler command (hjc) and an execution command (hj). Both commands take options similar to Java's commands *e.g.,* sourcepath (-sp), classpath (-cp), destination folder for classes (-d), etc. The hjc compiler option "-rt" allows the user to specify which runtime scheduler the code should be compiled for.

4.1 Interactions with system host

The HJ runtime can interact with the host system in various ways. First, an HJ program can call native CPU and GPU code through the extern interface described below. Second, the user can tune the HJ runtime deployment using the "-places" option and its accompanying thread-binding mechanism. These deployment options allow the user to control the mapping of an HJ program on a specific multicore processor without having to modify or recompile the program.

Invocation of native code from an HJ program: HJ's `extern` capability is a restricted and easy-to-use version of the Java Native Interface (JNI). The `extern` keyword, introduced in X10 1.5, enables HJ methods to pass and return values of primitive types to and from calls to native C code. The programmer just has to declare a method as `extern` in an HJ file, and implement its body as a function in a C file. All the necessary casting from the Java native type system to the C native type system is handled by a stub generated by the HJ compiler. Similarly to the Java native interface, the implementation of native `extern` methods must be bundled as a shared library object.

A key enhancement to the `extern` capability in HJ is the addition of array-views as parameters in `extern` methods/functions. Sharing multi-dimensional arrays between Java and native code is challenging because the native side cannot see a multi-dimensional Java array as a chunk of contiguous memory. To circumvent this problem, the programmer can either linearize array accesses in the Java code or access the Java array from the native side to perform a copy. Both solutions incur overheads in programmability and/or performance. Since an array-view is an abstraction built atop a one-dimensional Java array, it can be passed through the `extern` interface to the native code, where it can be considered as a simple chunk of contiguous memory. In this context, array-views are particularly useful as they eliminate the need to adapt any existing code either on the HJ side or the native side. The programmer can then easily share data with existing native library code that uses one dimensional arrays by calling them through the `extern` interface. On the HJ side, the programmer remains free to view the array as multi-dimensional through an array-view.

Array-views can also make it easier to bind HJ code with either Fortran or C native code by allowing the HJ programmer to easily switch the view from row-major to column-major layouts, without changing any of the array accesses. Integration with Fortran code is also made easier by using HJ views that re-base the index of the first element of an array from zero to one.

Integration of native code and GPUs: The combination of array-views and `extern` makes the switch from a classical native code implementation to a GPU implementation seamless. The programmer only need to replace the native implementation by the GPU implementation and compile a new version of the native shared library. The next execution of the HJ program will then perform calls to the GPU through the `extern` interface [39].

Exploiting Locality and Affinity: The hj command option, *"-places p:w"*, allows the user to specify how many places (p) and workers per place (w) the runtime should be initialized with. Specifying the number of places and worker per places can be a critical performance tuning option, depending on the nature of the application and the target architecture. The HJ runtime supports thread-binding capabilities to deliver on the programmer's expectation that workers in the same place should have greater affinity with each other than workers in different places. When launching an HJ program, a configuration file can be used to specify which physical cpu_id (the id used by the OS to identify a core) an HJ worker thread should be bound too. Figure 6 shows an example configuration file that maps four cpu_ids to two places, each with two workers.

4.2 Productivity Tools

The HJ environment currently includes three kinds of tools to improve programmer productivity. First, HJ features an *abstract execution metrics framework* that extracts various metrics from a dynamic execution of an HJ program, such as the total work, critical path length (span) and the ideal speedup. These metrics can help assess the theoretical efficiency of a parallel program, before actually evaluating its performance on a large number of cores. Second, a *data-race detection tool* [30] helps programmers identify the source of data race errors in their program. Finally, the HJ environment includes the DrHJ integrated development environment [28], which was implemented as an extension to the DrJava IDE.

```
2 2            // nb_places nb_workers_per_places
0 : 0 -> 0     // place_id : worker_id -> cpu_id
0 : 1 -> 1
1 : 0 -> 2
1 : 1 -> 3
```

Figure 6. Example of an HJ runtime thread-binding configuration file for the work-stealing runtime, mapping four workers from two places to four cpu ids [16]

The Abstract Execution Metrics API: The HJ runtime provides an API for the programmer to register and request abstract execution metrics. The abstract execution metrics can be particularly useful when debugging performance problems, and when comparing alternate implementations of an application. The programmer can insert a call of the form, perf.addLocalOps(N), anywhere in a task to indicate execution of N application-specific abstract operations e.g., floating-point operations, comparison operations, stencil operations, or any other data structure operations. Multiple calls to perf.addLocalOps() are permitted within the same task. They have the effect of adding to the abstract execution time of that task. The main advantage of using abstract execution times is that the performance metrics will be the same regardless of which physical machine the HJ program is executed on. The main disadvantage is that the abstraction may not be representative of actual performance on a given machine. When an HJ program is executed with the -perf=true option, abstract metrics are printed at the end of program execution. It captures the total number of operations executed (WORK) and the critical path length (CPL) of the call graph generated by the program execution. The ratio, WORK/CPL is also printed as a measure of ideal speedup for the abstract metrics.

Race Detection: A central property affecting the correctness of parallel algorithms is data-race freedom. Data-race freedom is a desirable property as in some cases it can imply determinism. For instance, in the absence of data races, all parallel programs with `async` and `finish` are guaranteed to be deterministic.

The HJ compiler and runtime implement an efficient dynamic analysis algorithm that checks the presence of data races in finish/async style parallel computations. The analysis implemented is a generalization of Feng and Leiserson's SP-bags algorithm [13] which was designed for checking determinism of spawn-sync Cilk programs. Since the SP-bags algorithm cannot be applied directly to finish/async parallel programs (which have more general structures than spawn/sync parallel programs), a new algorithm called ESP-bags was designed and implemented for HJ in [30]. Both the SP-bags and ESP-bags algorithms are sound for a given input. This means that if a data race exists for an input, it will be detected, regardless of the schedule. This race detector for HJ programs has been especially useful in the COMP 322 course, when students needed help debugging their programs.

DrHJ, an IDE for HJ: DrHJ [28] is an extension of the DrJava integrated development environment developed at Rice University. It is mainly composed of three elements, a navigation pane that shows documents currently opened, an editor pane that shows the source code, and a console pane that displays compiler messages and program output. The editor helps students edit HJ source code with syntax highlighting for HJ keywords. Compilation and execution of HJ programs can be done directly from the IDE. The interface also allows students to turn on the race detection mode. When this mode is used and a data race is detected, the student gets an error message indicating where in the source file the conflicting

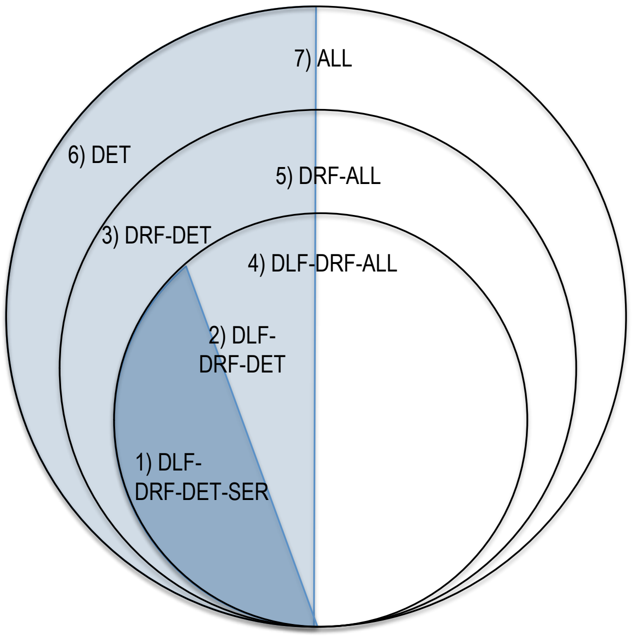

Figure 7. Pedagogical classes of HJ programs

operation happened. If the race occurs while operating on array elements, the index of the conflicting operation is provided as well. The integration of HJ with the original DrJava IDE has been done through the DrJava compiler adapter interface. A compiler adapter takes care of building invocations to a particular compiler and runtime as well as providing language specific information to the GUI such as a list of keywords to be highlighted, the file extension the compiler supports, etc. DrHJ is bundled as a single jar file and requires no particular installation beside having Java present in the environment path.

5. Pedagogical Approach

As mentioned earlier, HJ is used in a new sophomore-level course on "Fundamentals of Parallel Programming" (COMP 322 [1]) which has become a required course for all Computer Science majors at Rice. The orthogonal nature of HJ's features enables the creation of rich varieties of parallel programs. Figure 7 illustrates different classes of HJ programs that can enable students to progress along a sequence from simpler to more complex examples. The following abbreviations are used to define this classification:
DLF = DeadLock-Free
DRF = DataRace-Free
SER = Serializable
DET = Deterministic (includes SER)
ALL = Deterministic and nondeterministic (includes DET)

Thus, the 7 classes of HJ programs shown in Figure 7 can be summarized as follows:

1. DLF-DRF-DET-SER: Any HJ program in this class is deadlock-free, data-race-free, deterministic and serializable. If a program is data-race-free and only uses the following three parallel constructs, then it is guaranteed to belong to this class: `async` (without an await clause), `finish`, and `future` get. The DLF guarantee comes from the HJ constructs used (async, finish, future get). The DET guarantee follows from the DRF property for these programs. Finally, programs in this class are serializable (SER) because the sequential program obtained by removing the `async` and `finish` operations is guaranteed to represent a legal schedule for the original parallel program.
2. DLF-DRF-DET: Any HJ program in this class is deadlock-free, data-race-free and deterministic, but not necessarily serializable. If a program is data-race-free and only uses `phasers` with registration, next, signal operations (but no explicit wait) in addition to the constructs listed above for DLF-DRF-DET-SER, then it is guaranteed to belong to this class. The addition of `phasers` leads to the possibility of obtaining non-serializable programs since it may not be easy to convert an HJ program that uses `phasers` into an equivalent sequential program in general [35].
3. DRF-DET: Any HJ program in this class is data-race-free and deterministic. If a program is data-race-free and only uses `data-driven futures` and `phasers` with wait operations in addition to the constructs listed above for DLF-DRF-DET, then it is guaranteed to belong to this class. Both `data-driven futures` and `phasers` with explicit wait operations can lead to programs that deadlock. However, the deadlock will still be deterministic *i.e.,* all executions of the program with the same input will quiesce with exactly the set of suspended tasks (if any).
4. DLF-DRF-ALL: Any HJ program in this class is deadlock-free and data-race-free, but may not be deterministic. If a program is data-race-free and only uses `isolated` statements in addition to the constructs listed above for DLF-DRF-DET, then it is guaranteed to belong to this class. Even though HJ's `isolated` statements are simpler than Java's `synchronized` statements, this classification shows that it is very easy to lose determinism when using a critical section construct like `isolated`.
5. DRF-ALL: Any program in this class is data-race-free, but may not be deterministic. If a program is data-race-free and only uses `isolated` statements in addition to the constructs listed above for DRF-DET, then it is guaranteed to belong to this class.
6. DET: Any HJ program in this class is deterministic.
7. ALL: Set of all HJ programs.

There are a number of other pedagogical benefits of the HJ language in addition to the classification. The Java and HJ languages are very close, which makes porting applications from one language to the other quite straightforward. Other than the *volatile*, *synchronized* and *native* keywords in Java, HJ can be considered to be a superset of Java 1.4 (soon to be extended to Java 5). The *places* construct enables students to experiment with locality optimizations for memory hierarchies in a portable way. The data race detection tool, DrHJ, helps students debug race conditions in their program (especially since DrHJ reports all potential data races). The low annotation overhead of being able to wrap any Java statement with an `async` keyword, makes it easy to experiment with different task decomposition. Finally, the management of tasks is completed handled by the HJ runtime, unlike tasks in thread pools where the programmer has to be involved in the different steps of a task's lifecycle.

6. Application Experience

As indicated below, the HJ project has benefited from early experiences with the HJ language in both academic and industrial settings.

6.1 Benchmarking

Some Java (Java Grande Forum, Shootout) and native (Barcelona OpenMP Task Suite) benchmark suites have been ported to HJ.

The experience shown that porting existing Java code to HJ is easy. If the application already exhibits parallelism, it can be as simple as adding `finish` and `async` keywords or transforming a `for` loop into a HJ `forall` loop. Converting parallel-Java to HJ is also relatively simple in many cases. For instance, an anonymous Runnable inner class can usually be converted to an HJ `async` construct. Codes that are more task-oriented, and require the application to maintain task handles, usually find a natural conversion to HJ `futures`. There also natural synergies between HJ language constructs and OpenMP pragmas. For example the parallel 'for' OpenMP pragma can be mapped to an HJ `forall` construct. The 'single' and 'barrier' pragmas in OpenMP can be replaced by `phasers` constructs such as single and next. The 'critical' pragma can be translated to HJ `isolated` blocks.

The hierarchical `phaser` synchronization construct [34] exhibits good performances with respect to Java and OpenMP barriers. Microbenchmarks on a 128-thread UltraSPARC T2 SM shows a 64-thread barrier overhead for hierarchical `phasers` is 126.6× less than a CyclicBarrier, 27.2× less than an OpenMP barrier. Hierarchical `phasers` barrier overhead using all 128 threads is 335.1× less than a CyclicBarrier, 89.2× less than an OpenMP barrier. Using the `phaser` accumulator feature, the barrier and reduction overhead of a hierarchical `phaser` is 25.2× smaller than an OpenMP reduction when 64 threads are used. When using 128 threads the `phaser` overhead for performing the barrier and the reduction is 77.2× less than OpenMP. Experiments on some of the JGF (LUFact, SOR, MolDyn) and NAS (CG and MG) benchmarks show `phaser` can achieve between 15× and 45× speed-up when preferred over a traditional Java CyclicBarrier implementation.

Regarding task scheduling, generally speaking the work-stealing runtime is more efficient than the work-sharing runtime. However, the work-stealing runtime requires the user to select the most appropriate scheduling policy at compile time. Experimental results presented in [17] from a variety of benchmarks show that the adaptive scheduler achieves 0.98x-9.2× speedup over the help-first scheduler and 0.97x-4.5× speedup over the work-first scheduler for 64-thread executions. In contrast, the help-first policy is 9.2× slower than work-first only in the worst case, and the work-first policy is 3.7× slower than help-first only in the worst case. The adaptive policy enables the user to achieve better performance without having to worry about selecting the most appropriate policy. Further, for large irregular recursive parallel computations, the adaptive scheduler runs with bounded stack usage and achieves performance (and supports data sizes) that cannot be delivered by the use of any single fixed policy [17].

6.2 Dipole1D: Porting a production application to HJ

Dipole1D [22] is the kernel of an inversion program for generating smooth 1D models from Controlled-Source Electromagnetic and Magnetotelluric data (OCCAM1DCSEM). A joint effort with collaborators at BHP-Billiton has been completed to port Dipole1D, a legacy Fortran 90 application, to HJ. The original code consisted of a few thousand lines of Fortran code that relied heavily on complex arithmetic. In Java, a complex number can either be represented as a pair of reals or by a boxed class. Initial ports to HJ showed poor performance (when boxing complex numbers as objects) or poor code quality (when using pairs of real), since the Java language does not include any first-class support for complex numbers. Consequently the HJ language has been extended to support complex numbers as a primitive type. The later HJ version of Dipole1D was much easier to write, as the programmer can more easily compare it to the Fortran 90 version. We used array-views to re-base all array accesses through views with index origin = 1, so as to eliminate the traditional and error-prone index conversion phase when moving from Fortran to C or Java. The Dipole1D implementation has been evaluated over different input sets that exercise different parts of the code. Half the tests showed similar to better performance than the original Fortran 90 implementation. Two thirds of the remaining tests showed less than 50 percent of overhead, whereas the rest had an overhead factor greater than 2×. For this last category, the culprit was an inefficient implementation of the StrictMath.log10 function for Java. The Dipole1D application is a good candidate for parallelization as its outer-most loop is embarrassingly parallel. Using `finish` and `async` constructs the application achieved a linear speedup when varying the number of workers up to four. The best speed-up (6x) was achieved with 8 workers before the memory bandwidth became saturated. These results are especially encouraging, when one considers the minimal effort needed to insert `finish` and `async` keywords in the code.

6.3 CnC-HJ: a high-level declarative programming model

The Concurrent Collection (CnC) model falls into the same family as dataflow and stream-processing languages. CnC is provably deterministic and is suited for many applications incorporating static and dynamic forms of task, data, loop, pipeline, and tree parallelism. The three main constructs in the CnC coordination language are step collections, data item collections, and control tag collections. Statically, each of these constructs is a collection representing a set of dynamic instances. Step instances are the unit of distribution and scheduling. Item instances are the unit of synchronization and communication. Control tag instances are the unit of control.

CnC-HJ is an implementation of the CnC model on top of the Habanero-Java language [7]. The CnC-HJ translator processes a CnC graph and generates skeletons of HJ source code. In addition to the graph, the CnC programmer provides code for individual steps in a CnC graph. Though the step code is often sequential, the CnC-HJ implementation allows the user to exploit parallelism within a step if they desire to. The HJ extern interface can also used to delegate some heavy computations to a GPU [15].

The overall scheduling of CnC steps is performed by the CnC runtime, which runs atop the HJ runtime system. Various CnC steps scheduling policies have been implemented in the CnC runtime [7]. However, the best performance is obtained when CnC steps are implemented as `data-driven futures` *i.e.,* `async` tasks with await clauses [36].

7. Related Work

Modern languages and libraries provide lightweight dynamic task parallel execution models for improved programmer productivity. In dynamic task parallelism, computations are dynamically created as *tasks* and the runtime scheduler is responsible for scheduling and synchronizing the tasks across the cores. We can roughly classify implementations in three categories:

1. New languages, such as X10 [38], Chapel [9], and Fortress [4].
2. Extensions to existing languages, such as the Cilk [14] and OpenMP 3.0 [27] extensions to C, and the HJ extensions to Java.
3. Libraries extensions that provides parallel APIs, such as Intel Threading Building Blocks [31], Java Concurrency Utilities [29] and the Microsoft Task Parallel Library [11].

There are many practical advantages and disadvantages to choosing a language or a library approach [8]. A key advantage of a library-based approach to task parallelism is that it can integrate with existing code easily without relying on new compiler support. However, the use of library APIs to express all aspects of task parallelism can lead to code that is hard to understand and modify, especially for beginning programmers. A key advantage of a language-based approach is that the intent of the programmer is easier to express and understand, both by other programmers and by program analysis tools. However, a language-based approach usually requires the standardization of new language constructs. In this regard, the latest X10 language specification offers an interesting design point that may be representative of future directions in both sequential and parallel programming.

HJ can be classified as being part of the group of parallel programming languages that extend an existing language. We believe it is important to leverage the existing knowledge of a programmer community by extending an existing language, especially when selecting a pedagogic language for teaching parallel programming. Java is an interesting choice as the language is now commonly used and relies on a managed runtime, providing safety, exceptions and garbage collection. Moreover, generating standard Java bytecode allows HJ to take advantage of different JVM implementations developed by vendors for various hardware architecture. HJ programmers can also take advantage of the rich eco-system of Java libraries in their applications.

Compared to Cilk's spawn-sync computations which must be fully-strict, HJ's finish/async computations are terminally-strict but need not be fully-strict. Fully-strict computations can be scheduled with provably efficient time and space bounds using work-stealing with the work-first policy [6]. The same theoretical time and space bounds were extended to terminally-strict computations such as finish/async parallel computations [2] in languages like X10 and HJ. Concretely, it means that HJ is more general than Cilk since HJ activities can outlive their parent task. HJ also has a number of other constructs that are not supported by Cilk *e.g.,* futures, phasers, and isolated.

Another difference between the Cilk and HJ implementations is that the HJ implementation supports the help-first policy in addition to the work-first policy, and also includes an adaptive mode [17] that can switch back and forth between the two policies depending on the kind of parallelism the application exhibits at runtime. The adaptive policy is provably space-efficient if the serial depth-first execution of the computation does not exceed the stack threshold. If it exceeds the threshold, the scheduler moves the stack pressure to the heap.

8. Conclusion

HJ shows that a combination of compiler and runtime techniques can provide programmers with a simple, safe and powerful high-level parallel programming language without sacrificing performance. By relying on simple orthogonal parallel constructs with important safety properties, HJ allows programmers with a basic knowledge of Java to get started with parallel programming concepts by writing or refactoring applications to harness the power of multicore architecture. The finish/async model enables the creation of deadlock-free parallel applications. `Phasers` unify point-to-point and collective synchronizations in a single construct. `Isolated` provides a simple mutual exclusion construct that is amenable to efficient implementation.

While programmers focus on the design of their application, the HJ runtime delivers performance and scalability with respect to the available number of cores as well as the nature of parallelism that an application exhibits (recursive, flat or unstructured). The various scheduling policies achieve efficient scheduling and load balancing, while adapting to the specific nature of parallelism present in an application. `Phasers` provide an efficient and scalable hierarchical implementation to perform synchronization operations between tasks, which is critical as we move toward machines made of thousands of cores. Language extensions such as the addition of complex primitive types, array-views and the extern clause help improve programmer's productivity and simplify porting and inter-operability with native code. Runtime deployment options give users the flexibility to tune the HJ runtime to better match the system host architecture. Finally, the HJ runtime feedback capacities such as the abstract execution metrics and the race detection tools help the programmer to get feedback on theoretical performance as well as the presence of potential data race bugs.

The usability of HJ has been assessed in various occasions when porting benchmarks from Java or Fortran to HJ. The language has also been used in the introductory parallel programming class for sophomores at Rice University (COMP 322). It allowed students to build on their previous knowledge of Java and focus on the fundamentals of parallel programming and algorithm design instead of being distracted by the low-level intricacies of using a Java API like the `java.util.concurrent` library.

We conclude that since the vast majority of programmers are not parallel programming experts, it is essential for new generations of programming languages to provide easy-to-use first-class constructs for parallelism. Additionally, a language implementation must be accompanied by an efficient and configurable runtime framework that offers full performance transparency to the programmer. Finally, a programming language must also be accompanied by a well-integrated set of tools that focus on programmer productivity. We believe that HJ achieves all these goals for programmers who have a basic knowledge of sequential programming in Java.

Acknowledgments

This work was supported in part by the National Science Foundation under the HECURA program, award number CCF-0833166. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation. We also gratefully acknowledge support from an IBM Open Collaborative Research Faculty Award for this work.

We would like to thank all the X10 team members for their contributions to the Java-based X10 v1.5 release in 2007 which provided a starting point for the HJ implementation, and all members of the Habanero group for their contributions to the Habanero-Java project in the form of source code, bug reports, program examples, and suggestions. Special thanks go to the undergraduate students at Rice University who took COMP 322 in Spring 2011 for their enthusiasm and feedback on HJ.

References

[1] COMP 322: Fundamentals of Parallel Programming. URL `https://wiki.rice.edu/confluence/display/PARPROG/COMP322`.

[2] Shivali Agarwal, Rajkishore Barik, Dan Bonachea, Vivek Sarkar, Rudrapatna K. Shyamasundar, and Katherine Yelick. Deadlock-free scheduling of x10 computations with bounded resources. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on parallelism algorithms and architectures*, pages 229–240, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-667-7.

[3] Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and Rudrapatna K. Shyamasundar. May-happen-in-parallel analysis of x10 programs. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 183–193, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-602-8.

[4] Eric Allan, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress language specification version 1.0. Technical report, Sun Microsystems, April 2005.

[5] Rajkishore Barik and Vivek Sarkar. Interprocedural load elimination for dynamic optimization of parallel programs. In *PACT'09, Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 41–52, Washington, DC, USA, Sep 2009. IEEE Computer Society. ISBN 978-0-7695-3771-9.

[6] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 356–368, 1994.

[7] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff P. Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sağnak Taşırlar. The CnC Programming Model. *Journal of Scientific Programming*, 2010.

[8] Vincent Cavé, Zoran Budimlić, and Vivek Sarkar. Comparing the usability of library vs. language approaches to task parallelism. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, pages 9:1–9:6, 2010. ISBN 978-1-4503-0547-1.

[9] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007. ISSN 1094-3420.

[10] P. Charles et al. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA'05*, pages 519–538, New York, NY, USA, 2005. ISBN 1-59593-031-0.

[11] Joe Duffy. *Concurrent Programming on Windows*. Addison-Wesley, 2008. ISBN 9780321434821.

[12] Alejandro Duran et al. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *ICPP'09*, pages 124–131, 2009.

[13] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in cilk programs. *Theory Comput. Syst.*, 32(3):301–326, 1999.

[14] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, New York, NY, USA, 1998. ACM. ISBN 0-89791-987-4.

[15] Max Grossman, Alina Simion Sbrlea, Zoran Budimli, and Vivek Sarkar. CnC-CUDA: Declarative Programming for GPUs. In *2010 Workshop on Languages and Compilers for Parallel Computing (LCPC)*, volume 6548 of *Lecture Notes in Computer Science*, pages 230–245. Springer Berlin / Heidelberg, 2011.

[16] Yi Guo. *A Scalable Locality-aware Adaptive Work-stealing Scheduler for Multi-core Task Parallelism*. PhD thesis, Rice University, Aug 2010.

[17] Yi Guo, Jisheng Zhao, Vincent Cavé, and Vivek Sarkar. SLAW: a Scalable Locality-aware Adaptive Work-stealing Scheduler. In *IPDPS '10: Proceedings of the 2010 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, Apr 2010. IEEE Computer Society.

[18] R. Gupta. The fuzzy barrier: a mechanism for high speed synchronization of processors. In *ASPLOS-III*, pages 54–63, New York, USA, 1989. ACM. ISBN 0-89791-300-0.

[19] Habanero. Habanero Multicore Software Research Project web page. http://habanero.rice.edu, January 2008.

[20] Java Grande Forum. The Java Grande Forum Benchmark Suite. http://www.epcc.ed.ac.uk/javagrande/javag.html.

[21] Mackale Joyner, Zoran Budimlic, and Vivek Sarkar. Subregion Analysis and Bounds Check Elimination for High Level Arrays. In *20th International Conference on Compiler Construction*. Springer-Verlag, 2011.

[22] K Key. One-dimensional inversion of multi-component, multi-frequency marine CSEM data: Methodology and synthetic studies for resolving thin resistive layers: Geophysics. in review, 2008.

[23] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[24] Roberto Lublinerman, Jisheng Zhao, Zoran Budimlic, Swarat Chaudhuri, and Vivek Sarkar. Delegated Isolation. In *Proceedings of OOPSLA 2011*, 2011.

[25] Alex Miller. Set your Java 7 Phasers to stun. http://tech.puredanger.com/2008/07/08/java7-phasers/, 2008.

[26] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *12th International Conference on Compiler Construction*, pages 138–152. Springer-Verlag, 2003.

[27] OpenMP. OpenMP Application Program Interface, Version 3.0, May 2008. http://www.openmp.org/mp-documents/spec30.pdf.

[28] Jarred Payne, Vincent Cavé, Raghavan Raman, Mathias Ricken, Robert Cartwright, and Vivek Sarkar. Tool Demonstration: DrHJ — a lightweight pedagogic IDE for Habanero Java. In *PPPJ'11: Proceedings of 9th International Conference on the Principles and Practice of Programming in Java*, 2011.

[29] Tim Peierls, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2006. ISBN 0321349601.

[30] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. Efficient Data Race Detection for Async-Finish Parallelism. In *RV'10, Proceedings of the 1th International Conference on Runtime Verification*. Springer, Nov 2010.

[31] James Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007. ISBN 9780596514808.

[32] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phaser accumulators: A new reduction construct for dynamic parallelism. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3751-1.

[33] J. Shirako et al. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICS '08*, pages 277–288, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-158-3.

[34] Jun Shirako and Vivek Sarkar. Hierarchical Phasers for Scalable Synchronization and Reduction. In *IPDPS '10: Proceedings of the 2010 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2010. IEEE Computer Society.

[35] Jun Shirako, Jisheng Zhao, V. Krishna Nandivada, and Vivek Sarkar. Chunking parallel loops in the presence of synchronization. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 181–192, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-498-0.

[36] Sagnak Tasirlar and Vivek Sarkar. Data-driven tasks and their implementation. In *Proceedings of International Conference on Parallel Processing (ICPP)*, 2011.

[37] R. Vallée-Rai et al. Soot - a Java Optimization Framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999. URL `www.sable.mcgill.ca/publications`.

[38] Vijay Saraswat and Bard Bloom. Report on the Programming Language X10, Version 2.0.5, July 2010. http://x10.codehaus.org.

[39] Yonghong Yan, Max Grossman, and Vivek Sarkar. Jcuda: A programmer-friendly interface for accelerating java programs with cuda. In *Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, pages 887–899, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03868-6.

[40] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement. In *Languages and Compilers for Parallel Computing, 22nd International Workshop, LCPC 2009*, volume 5898 of *Lecture Notes in Computer Science*. Springer, 2009. ISBN 978-3-642-13373-2.

[41] K. Yelick et al. Productivity and performance using partitioned global address space languages. In *PASCO'07*, pages 24–32, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-741-4.

[42] Jisheng Zhao, Jun Shirako, V. Krishna Nandivada, and Vivek Sarkar. Reducing Task Creation and Termination Overhead in Explicitly Parallel Program. In *PACT'10, Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, Sep 2010.