Habanero-Java Library: a Java 8 Framework for Multicore Programming

Shams Imam
Rice University
shams@rice.edu

Vivek Sarkar
Rice University
vsarkar@rice.edu

Abstract

With the advent of the multicore era, it is clear that future growth in application performance will primarily come from increased parallelism. We believe parallelism should be introduced early into the Computer Science curriculum to educate students on the fundamentals of parallel computation. In this paper, we introduce the newly-created Habanero-Java library (HJlib), a pure Java 8 library implementation of the pedagogic parallel programming model [12]. HJlib has been used in teaching a sophomore-level course titled "Fundamentals of Parallel Programming" at Rice University.

HJlib adds to the Java ecosystem a powerful and portable task parallel programming model that can be used to parallelize both regular and irregular applications. By relying on simple orthogonal parallel constructs with important safety properties, HJlib 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 HJlib APIs make extensive use of lambda expressions and can run on any Java 8 JVM. HJlib runtime feedback capabilities, such as the abstract execution metrics and the deadlock detector, help the programmer to obtain feedback on theoretical performance as well as the presence of potential bugs in their program.

Being an implementation of a pedagogic programming model, HJlib is also an attractive tool for both educators and researchers. HJlib is actively being used in multiple research projects at Rice and also by external independent collaborators. These projects include exploitation of homogeneous and heterogeneous multicore parallelism in big data applications written for the Hadoop platform [20, 43].

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming

General Terms Design, Languages

Keywords Task Parallelism, Habanero-Java Library, Lambda Expressions, Java Task Parallel Library

PPPJ'14, September 23 – 26, 2014, Cracow, Poland..

http://dx.doi.org/10.1145/2647508.2647514

1. Introduction

Multicore processors are now ubiquitous in server, desktop, and laptop hardware [33]. They are also making their way into smaller devices, such as smartphones and tablets. With the advent of the multicore era, it is clear that future growth in application performance will primarily come from increased parallelism. Programming models that can express large degrees of parallelism offer a scalable solution for the future where core counts are expected to increase exponentially. Parallelism, is hence, the future of computing. We believe parallelism should be introduced early into the Computer Science curriculum to educate students on the fundamentals of parallel computation. At Rice University, we already offer a course titled "COMP 322 - Fundamentals of Parallel Programming" [4] for second-year undergraduate students to address this issue. COMP 322 has become a required course for all Computer Science majors at Rice, and its curricular material is being used in other universities as well.

In this paper, we introduce the newly-created Habanero-Java library (HJlib), a pure library implementation of the pedagogic Habanero-Java (HJ) language [12]. HJlib is our effort to move away from the use of a custom language in COMP 322. The library puts a particular emphasis on the usability and safety of parallel constructs, which are further described in Section 3. For example, no HJlib 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. Event-driven programming is also supported via the *actor* programming construct. Finally, any HJlib program written with async, finish, and phaser constructs that is data race free is guaranteed to also be deterministic. In addition to being an implementation of a pedagogic programming model, HJlib is actively being used in multiple research projects at Rice (including projects to exploit multicore parallelism in Java-based Hadoop "big data" applications [20, 43]) and also by external independent collaborators.

Among existing programming languages, Java provides a robust (though relatively low-level) support for multithreading and concurrency. The introduction of the Fork/Join Framework [27] in Java 7 gave Java developers the ability to start leveraging parallelism at an application, rather than system-programming, level. We build on the capabilities offered by the Fork/Join Framework to provide a higher level of abstraction and a wider range of parallel programming constructs in HJlib. HJlib is built using lambda expressions and can run on any Java 8 JVM. Older JVMs can be targeted by relying on external bytecode transformations tools (e.g. retrolambda [29]) for compatibility. HJlib adds to the Java ecosystem a powerful and portable task parallel programming model that can be used to parallelize both regular and irregular applications.

HJlib supports an explicit parallel programming model and promotes an execution model for multicore processors based on three

orthogonal dimensions for portable parallelism: *a*) lightweight dynamic task creation and termination; *b*) collective and point-to-point synchronization; and *c*) mutual exclusion and isolation. The HJlib runtime is responsible for orchestrating the creation, execution, and termination of tasks, and builds on a work-stealing scheduler (`java.util.concurrent.ForkJoinPool`) in the standard JDK. A separation of concerns enables programmers to focus on the design of their application, while the HJlib runtime delivers performance and scalability with respect to the available number of cores as well as the nature of parallelism that an application exhibits.

This paper makes the following contributions:

- A pure library implementation of a pedagogic task-parallel programming model called Habanero-Java library (HJlib). HJlib is built using lambda expressions and can run on any Java 8 JVM without any other dependencies. The library supports a multitude of parallel programming constructs further described in Section 3.
- The `EventDrivenControl` API which can be used to implement custom synchronization constructs while executing asynchronous tasks managed by the HJlib runtime. Existing HJlib synchronization constructs such as futures, data-driven futures, and phasers, are all built using this API. Expert library developers can use this API to extend HJlib to add custom parallel constructs, if needed.
- A new framework for Abstract Execution Metrics (AEM). Compared to previous work which supported only async and finish computations [12], our implementation supports AEM for the full gamut of synchronization constructs in HJlib. An API is available for users to add metrics support for custom synchronization constructs. The metrics can be particularly useful when debugging performance problems, and when comparing different parallel algorithms for the same problem.
- We introduce a new deadlock detector that detects deadlocks in HJlib programs with diagnostic information. The diagnostic information includes source code locations that help users in debugging their parallel programs. Our detector also supports custom synchronization constructs written using the `EventDrivenControl` API.

The paper is organized as follows: in Section 2 we motivate the design of HJlib as a pedagogical Java library for exploiting multicore parallelism. Section 3 summarizes the parallel constructs supported in HJlib that enable writing of programs that exploit task, data, and pipeline parallel programming patterns. We describe our implementation and the `EventDrivenControl` API in Section 4: this API allows users to write their own parallel constructs while reusing our runtime. Section 5 introduces the AEM framework supported by HJlib which can be used to reason about the performance of parallel programs at the algorithmic level. In Section 6, we describe the new deadlock detector available in HJlib. Section 7 discusses example programs and benchmarks available in the HJlib release which help users to get started with learning how to write HJlib programs. Section 8 discusses related work; we summarize the paper and outline possible directions for future work in Section 9.

2. Motivation

Programs typically exhibit varying degrees of task, data, and pipeline parallelism [19]. Current mainstream programming models are based on concurrent system programming primitives, and provide limited support for expressing parallelism at the application level. Programmers, hence, need higher level parallel programming models to reduce the burden of reasoning about and writing parallel programs. In addition, the programming model must carefully distinguish between parallelism (using deterministic task decomposition to effectively utilize multiple computational resources) and concurrency (correctly coordinating non-deterministic interactions among multiple tasks). At Rice, we are developing a pedagogic parallel programming model to challenge this issue as part of the Habanero Extreme Scale Software Research Project [2].

The previous implementation of the pedagogic parallel programming model was the Habanero-Java (HJ) language [12]. 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 current and future multicore architectures. There are many practical advantages and disadvantages to choosing a language or a library approach. 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 [11]. However, a language-based approach requires the standardization of new language constructs for mainstream adoption.

The implementation of HJ requires language extensions to Java with special compiler support. The current HJ implementation is based on a subset of Java 5 (which includes generics, but not enums and foreach loops), which is a source of confusion among students and collaborators. Although HJ generates Java bytecode, it is a language with its own type system separate from Java. For example, when a programmer instantiates a new `Integer` object, the implementation package is `hj.lang` rather than `java.lang`. This was a major source of confusion for external collaborators and a major source of overhead in migrating existing Java code to HJ or using standard Java libraries and APIs in a newly written HJ program.

The use of library APIs to express all aspects of task parallelism has the drawback of allowing programmers to make lexical or scoping errors which could easily be caught by a compiler with a language-based approach. However, a key advantage of a library-based approach is that it can integrate with existing code more easily. It does not introduce any new language rules nor does it require modifying a compiler. In addition, users can write the code in an IDE of their choice and use their preferred debuggers.

Being an implementation of a pedagogic programming model, HJlib is also an attractive tool for educators. There are numerous educational resources available from the COMP 322 course offered at Rice University [4]. In addition to lecture notes and videos, HJlib has extensive documentation [5] and examples available to help users get started. Java is amongst the most popular languages taught in introductory programming courses [15, 30] and this bodes well for an increased use of HJlib as other institutions start introducing parallel programming earlier in their curricula. Using HJlib offers two main advantages in its use as an introductory parallel programming language for first or 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 on this knowledge as HJlib programs are essentially Java programs and conform to the Java syntax and language rules. Second, since HJlib is a high-level parallel programming model, it makes it easier to focus on general parallel programming concepts, algorithms and patterns without being distracted by low-level details, such as threads and locks.

The COMP 322 course is also a great opportunity to get a sense of how HJlib is perceived by non-experts in parallel programming. Overall, the use of HJlib in COMP 322 garnered positive feedback from students due to *a*) the lack of differences in rules from the Java language; *b*) the ability to use an IDE of their choice while writing programs; and *c*) the ability to debug their programs using standard Java IDEs. Encouraged by this, we are now planning to

use HJlib for the COMP 322 equivalent of a massively open online course to be offered by Rice.

The ultimate goal of parallel computation is to increase performance by reducing computation time. Extracting performance from parallel solutions requires reasoning about the asymptotic complexity of the parallel algorithm. In HJlib, we use the notion of AEM to enable users to reason about the performance of their parallel algorithms. The metrics involve computing the total work done and the critical path length of the dynamic computation graph. These metrics can be used to reason about the ideal abstract execution time of the computation, assuming an infinite number of processors. While the HJ language supported AEM for `async-finish` computations, HJlib supports the metrics for a much wider variety of parallel constructs. The main advantage of using abstract execution times is that these performance metrics are reproducible; they will be the same regardless of which physical machine the HJlib program is executed on. HJlib also provides a deadlock detector to enable users to debug their programs while using the various synchronization constructs (if they venture beyond the deadlock-free subset of HJlib).

3. Background: Parallel Constructs

HJlib integrates a wide range of parallel programming constructs (e.g., async tasks, forall, isolated, futures, data-driven tasks, barriers, phasers, actors) in a single programming model that enables unique combinations of these constructs (e.g., nested combinations of task and actor parallelism). The orthogonal classes of parallel constructs enables programmers with a basic knowledge of Java to get started quickly with expressing a wide range of parallel programming patterns. HJlib is capable of expressing many different forms of parallel patterns including data parallelism, pipeline parallelism, stream parallelism, loop parallelism, and divide-and-conquer parallelism.

3.1 The Async/Finish Constructs

The Async/Finish Model (AFM) is well-suited to exploit task parallelism in divide-and-conquer style and loop-style programs. In the AFM, a task can *fork* a group of child tasks. These child tasks can recursively fork even more tasks. Tasks are created at *fork* points and HJlib provides the `async` method to create a task. The statement `async(()->` $\langle stmt \rangle$`)` causes the parent task to create a new child task to execute $\langle stmt \rangle$ (logically) in parallel with the parent task.

```
1  class AsyncFinishPrimer {
2    public static void main(String[] args) {
3      finish(() -> {
4        println("Task O"); // Task-O
5        finish(() -> {
6          async(() -> { // Task-A
7            println("Task A");
8          });
9          async(() -> { // Task-B
10           println("Task B");
11           async(() -> { // Task-B1 created by Task-B
12             println("Task B1")
13           });
14           async(() -> { // Task-B2 created by Task-B
15             println("Task B2");
16           });
17         });
18       }); // Wait for tasks A, B, B1, B2 to finish
19       println("Task C") // Task-C
20     });
21   } }
```

Figure 1: HJlib version of a simple Fork/Join program using `async` and `finish` constructs.

Further, a parent/ancestor task can selectively *join* on a subset of child/descendant tasks. This is the primary form of synchronization between tasks in `async-finish` style programs. The `finish` method in HJlib is used to represent a join operation. The task executing `finish(()->` $\langle stmt \rangle$`)` has to wait for all child tasks created inside $\langle stmt \rangle$ to terminate before it can proceed.

`async-finish` style computations are known to be deadlock free [14]. In the absence of data races, these programs are deterministic [34]. The scopes of `async` and `finish` can span method boundaries. As a result, parallelizing sequential programs using `async-finish` is fairly easy. `async`s are inserted to wrap statements which can be executed in parallel. Then these `async`s are wrapped inside `finish` blocks to ensure the parallel version produces the same result as the sequential version. This process of inserting `finish` blocks can also be automated [40].

3.2 Loop Parallelism

Loop parallelism is especially important when it comes to handling large sets of data in parallel. HJlib supports a variant of for loops over rectangular regions using the `forall` and `forasync` methods to iteratively spawn parallel tasks inside the loop. HJlib also provides an implicit conversion for `Iterables` to remove the restriction of rectangular regions and to simplify use of `forall` and `forasync` methods on Java collections. The syntax and semantics of one-dimensional versions of `forall` and `forasync` are shown in Figure 2. An implicit `finish` is included for all iterations of the `forall`, while there is no implicit `finish` in `forasync`. Another typical way to take advantage of loop-level parallelism is to partition the data to be processed into chunks and create one computational task to process each chunk of data. HJlib supports the `forallChunked` and `forasyncChunked` methods to support this style of computation.

```
1  forall(start, end, (i) -> f(i)) ==
2    finish(() -> {
3      for (int i = start; i <= end; i++) {
4        async(() -> f(i));
5      }
6    });
7  forasync(start, end, (i) -> f(i)) ==
8    for (int i = start; i <= end; i++) {
9      async(() -> f(i));
10   }
```

Figure 2: Syntax and semantics of one-dimensional versions of `forall` and `forasync`.

3.3 Coordination constructs

There are often dependences among parallel tasks. In such scenarios, coordination between tasks is required to determine when dependent tasks can be executed. HJlib augments the AFM with a handful of coordination constructs: `isolated`, futures, data-driven futures, phasers, and actors. In addition, as we will see in Section 4, HJlib provides an API for end-users to be able to create their own synchronization constructs.

3.3.1 Synchronized access using `isolated`

A concern common in most shared memory models is the issue of data races and the need to synchronize the accesses to shared resources/variables between tasks. HJlib provides an `isolated(()->` $\langle stmt \rangle$`)` construct to support weak isolation, i.e. atomicity is guaranteed only with respect to other statements also executing inside `isolated` scopes. No guarantees are provided on interactions with non-`isolated` statements; accesses to shared variables by parallel tasks outside `isolated` blocks may participate in data races. In HJlib, we do not allow other parallel constructs to be

nested inside `isolated` blocks and the runtime reports an error in such scenarios.

```
public class IsolatedPrimer {
  public static void main(String[] args) {
    finish(() -> {
      final int[] counter = new int[4];
      finish(() -> {
        forall(1, 399, (i) -> {
          isolated(() -> {
            int n = i % 4;
            counter[n] = counter[n] + 1;
          });
        });
        // the statement below would introduce a
        // data race as outside an isolated scope
        /* counter[0] = counter[0] + 1; */
      });
      for (int i = 0; i < 4; i++) {
        assert("data race found", counter[i]==100);
      }
    });
  }
}
```

Figure 3: HJlib `isolated` statements at work. Each `isolated` block executes sequentially and there are no data races.

The example in Figure 3 demonstrates the use of global isolation which causes all `isolated` statements to be serialized. This can be a serious performance bottleneck in applications with moderate contention [12]. HJlib adds support for finer-grained support to `isolated` blocks, called object-based isolation, of the form `isolated(`$variable11, ..., variableN$`, () -> ⟨stmt⟩)` which provides better performance. The object-based isolation version serializes only conflicting `isolated` blocks, the conflicts are determined by the intersection of the representative set of locks for the objects $variable11, ..., variableN$.

3.3.2 Futures

A `future` represents the result of an asynchronous computation and extends HJlib `async` to support return values [23]. The statement `HjFuture<T> f = future<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()` 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 `future` task completes and the result of *expr* becomes available. One advantage of using `futures` is that there can never be a data race on accesses to a `future`'s return value. In addition, if all `futures` are stored in immutable variables, it ensures that no deadlock cycle can be created with `future` tasks.

```
public class FibFuturePrimer {
  public int fib(int n) {
    if (n < 2) {
      return n;
    } else {
      HjFuture<Integer> x = future(() -> fib(n-1));
      HjFuture<Integer> y = future(() -> fib(n-2));
      return x.get() + y.get();
  } }
}
```

Figure 4: HJlib Fib using `futures`. A relatively large value of n will cause the program to run out of memory due to excessive creation of threads in the current HJlib runtime.

3.3.3 Data-Driven Futures (DDFs)

DDFs are an extension to futures to support the dataflow model [41]. DDFs support a single assignment property in which each DDF must have at most one producer. Any `async` can register on a DDF as a consumer causing the execution of the `async` to be delayed until a value becomes available in the DDF. The exact syntax for an `async` waiting on a DDF is as follows: `asyncAwait(`$ddf1, ..., ddfN$`, () -> ⟨stmt⟩)`. An `async` waiting on a chain of DDFs can only begin executing after a `put()` has been invoked on all the DDFs. Accesses to values passed inside DDFs are guaranteed to be data race free and deterministic.

```
public class FibDdfPrimer {
  public int fib(int n, HjDDF<Integer> v) {
    if (n < 2) {
      v.put(n);
    } else {
      HjDDF<Integer> x = newDataDrivenFuture();
      HjDDF<Integer> y = newDataDrivenFuture();
      async(() -> fib(n-1, x));
      async(() -> fib(n-2, y));
      asyncAwait(x, y, v.put(x.get() + y.get()));
  } }
}
```

Figure 5: HJlib Fib using DDFs. Each call to `fib()` produces an `async` task that waits on values to be produced by its children before it computes the local result and stores it in the `v` (result) DDF. This version is more scalable compared to the futures version in Figure 4. It requires the programmer to change the natural flow of the program to think in terms of continuations and the DDFs.

3.3.4 Phasers

The `phaser` construct [37] unifies collective and point-to-point synchronization for phased computations. Each task has the option of registering with a `phaser` in *signal-only/wait-only mode* for producer/consumer synchronization or *signal-wait* mode for barrier synchronization. The latest release of Java includes Phaser synchronizer objects, which are derived in part [31] from the `phaser` construct in the Habanero runtime. In general, a task may be registered on multiple `phasers`, and a `phaser` may have multiple tasks registered on it. `phasers` ensure deadlock freedom when programmers use only the `next` statements in their programs. In programs where tasks are involved with multiple point-to-point coordination, explicit use of `doWait()` and `doSignal()` on multiple `phasers` might be required. In such scenarios, some effort is required on the part of the programmer to carefully reason about the sequence of such calls to ensure correctness and deadlock freedom.

3.3.5 Finish Accumulators

Finish accumulators support predefined parallel reductions for dynamic task parallelism [38]. Finish accumulators are designed to be integrated into the `async` and `finish` constructs so as to guarantee determinism for accumulation and to avoid any possible race conditions in referring to intermediate results. Parallel tasks asynchronously transmit their data to finish accumulators with `put` operations and retrieve the results by `get` operations. A finish accumulator, `ac`, is accessible to sub-tasks if and only if `ac` is associated with a `finish` statement and the sub-tasks are created within the `finish` scope. To ensure an absence of races, `get` operations by sub-tasks return the value at the beginning of the associated `finish` scope and are not affected by put operations within the same finish scope.

3.3.6 Actors

An actor is defined as an object that has the capability to process incoming messages produced by other actors [7]. Typically, the ac-

tor has a mailbox to store its incoming messages. An actor also maintains local state which is initialized during creation. Henceforth, only the actor is allowed to update its local state using data (usually immutable) from the messages it receives and from the intermediate results it computes while processing the message. The actor is restricted to process at most one message at a time. There is no restriction on the order in which the actor decides to process incoming messages. As an actor processes a message, it is allowed to change its behavior affecting how it processes the subsequent messages. From the AFM perspective, actors in HJlib are treated as long running `asyncs` and hence can nest any of the Async/Finish compliant constructs in their message-processing body [24]. This simplifies termination detection and enables exposing parallelism inside the actor while processing messages.

4. Implementation

HJlib is implemented as a pure Java library with no dependencies outside the standard JDK. The implementation relies heavily on the use of functional interfaces and lambda expressions introduced in Java 8. Lambdas allow the syntax of HJlib programs to be close to that of programs written in the HJ language. The entire library has been implemented from scratch to support all the parallel constructs described in Section 3. The implementation also publishes events to allow custom listeners to be attached to the runtime to enable implementation of additional features such as abstract execution metrics (Section 5) and deadlock detection (Section 6). In addition, the library provides a core API (Section 4.1) to enable users to implement their own synchronization constructs (SyncCons). The implementation techniques described below will be useful for language designers and implementers (who may choose to build `async-finish` style task-parallel implementations on other serial languages or target architectures). It will also aid application and library programmers who may build their frameworks on top of HJlib.

The HJlib runtime is responsible for orchestrating the creation, execution, and termination of tasks. The management of actual threads and related thread pools is done by the runtime and is transparent to the tasks in the program. HJlib uses the `ExecutorService` and Fork/Join framework [27] to manage the parallel execution of tasks using work-stealing schedulers. Our runtime uses the *help-first* policy [21] for task scheduling. Under this policy, spawning a child task enqueues it in the task queue and allows the parent task to continue execution past the spawn operation. The child task can then be executed by any of the worker threads.

The Fork/Join framework is designed to make divide-and-conquer algorithms easy to parallelize. HJlib provides additional synchronization constructs (SyncCon) (such as futures, phasers, actors, etc.) that allow parallelizing a larger class of algorithms and applications. Synchronization constraints can prevent a currently executing task from making further progress as it *waits* to synchronize with other ready but not executing task(s). Our runtime implements such waits by blocking the worker thread, but first it smartly attempts to make further progress in the overall computation (Section 4.3).

The HJlib runtime is also responsible for the management of data structures to resolve blocking conditions in the presence of arbitrary dependences or synchronization constraints. Figure 6 provides a diagrammatic explanation of the HJlib runtime. The runtime places tasks into queues while the pool of worker threads continuously attempt to execute tasks dequeued from these queues. In addition, execution of tasks may result in more tasks being spawned and enqueued into the queues. Worker threads may get blocked due to synchronization constraints on the tasks. When the synchronization constraint is resolved, the task is unblocked and the worker

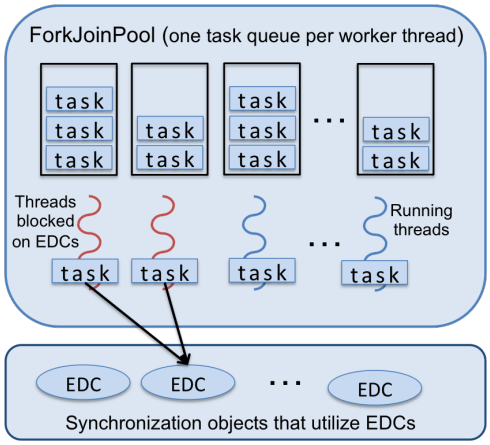

Figure 6: The HJlib runtime includes worker threads and ready task queues like most other task parallel runtimes. In addition, there are EDCs which maintain a list of blocked tasks to implement higher-level synchronization constructs.

thread can resume execution. An application starts with a single *main* task in the work queue which promptly gets executed by one of the worker threads. The application terminates when: *a*) the work queues are empty; and *b*) all synchronization constraints in the program have been satisfied (i.e. there are no deadlocks).

4.1 `EventDrivenControl` API

Event-Driven Controls (EDCs) are an extension to Data-Driven Controls (DDCs) which were presented in [24]. A DDC lazily binds a value and a closure called the execution body (EB), both the value and the EB follow the dynamic single-assignment property ensuring data-race freedom. When the value becomes available, the EB is executed using the provided value. We generalize DDCs to EDCs to allow multiple EBs to be attached to the EDC as callbacks. We treat the availability of a value in the EDC as an *event* and use the event to trigger the execution of EBs. Due to the single-assignment property, the registered EBs are executed at most once. The EB could store book-keeping data and act as a synchronous callback into the runtime. Figure 7 shows a simplified implementation of an EDC excluding concurrency concerns.

To allow library/language developers to create their own SyncCons, we expose EDCs as an API in our runtime. The API contains the following operations:

- The static `newEDC()` factory method is used to instantiate a new EDC. EDCs are initialized without a resolved value and with an empty EB list. The EDC can be used like a regular object, e.g. stored as a field, passed around as parameters, invoked as receivers for methods, etc.
- The static `suspend(anEdcInstance)` method signals possible creation of a suspension point. If the EDC passed as an argument has not been resolved, the current task is blocked until the EDC is resolved.
- The `isValueAvailable()` method can be used to check whether the value in the EDC has been resolved.
- The `setValue(someValue)` method resolves the EDC, i.e. it binds a value with the EDC and triggers the execution of any EBs registered with the EDC.
- The `getValue()` method retrieves the value associated with the EDC. It is only safe to call this method if the value in the EDC has already been resolved. If execution proceeds past a

```
1  class EventDrivenControl {
2    ValueType value = ...;
3    List<ExecBody> ebList = ...;
4    /** triggers callback execution **/
5    void setValue(ValueType theValue) {
6      if (!valueAvailable()) {
7        value = theValue;
8        // execute the callbacks/EBs
9        ebList.each().scheduleWith(value);
10     } } else {
11       // check for error
12   } }
13   /** enables callback registration **/
14   void addExecutionBody(ExecBody theBody) {
15     if (valueAvailable()) {
16       // value available, execute immediately
17       theBody.scheduleWith(value);
18     } else {
19       // need to wait for the value
20       ebList.add(theBody);
21 } } }
```

Figure 7: Simplified representation of an EDC not displaying synchronizations or validations. Both the value and the execution body can be lazily attached. The execution body uses the value of the EDC in the `scheduleWith()` method.

call to `suspend()`, it is guaranteed that a value is available in the EDC.

This API is used in our runtime implementation to support the constructs such as `finish`, futures, phasers, etc. The implementation details for each of these constructs are available in [25]. The key idea is to translate the SyncCons into producer-consumer constraints on EDCs and to suspend the consumer of an EDC when waiting on the producer to resolve the EDC. We claim that any task-parallel SyncCon can be translated in such a manner and hence be supported by our runtime. As a result, library developers can use this API and implement their custom SyncCons and synchronization patterns. In Section 4.2, we show how this API can be used to implement a new SyncCon called EventCount.

4.2 EventCount Synchronization Construct

An *eventcount* is an object that keeps a count of the number of events in a particular class that have occurred so far in the execution of the system [35]. The primitive operations on an eventcount may be concurrent. Events are the executions of three primitive operations:

- `advance`: It is used to signal the occurrence of an event. The effect of this operation is to increase the integer value of the eventcount by 1.
- `await(v)`: It suspends the calling task until the value of the eventcount is at least `v`. The await primitive may not return immediately once the v^{th} advance on the eventcount is executed; the only guarantee is that at least `v` advances have been performed by the time `await(v)` returns.
- `read`: The value returned by `read()` counts all of the `advance` operations that precede the execution. It may or may not count those in progress during the `read`.

Figure 8 highlights a simple implementation of the eventcount synchronization construct using EDCs. A `Map` is used to track the representative EDC for each event value.[1] The `advance` operation increments the counter atomically and then resolves the EDC corresponding to that event count (line 15). The `await` operation looks up the EDC entry corresponding to the event count `v` and then calls

[1] A more sophisticated implementation would minimize the entries stored in the `Map` to avoid memory leaks.

```
1  public final class EventCount {
2
3    Map<Long, EventDrivenControl> eventMap = new ↩
         ConcurrentHashMap<>();
4    AtomicLong eventCounter = new AtomicLong(0);
5
6    public EventCount() {
7      EventDrivenControl edc = newEDC();
8      eventMap.put(0L, edc);
9      edc.setValue(Boolean.TRUE);
10   }
11   public void advance() {
12     long v = eventCounter.incrementAndGet();
13     eventMap.putIfAbsent(v, newEDC());
14     EventDrivenControl edc = eventMap.get(v);
15     edc.setValue(Boolean.TRUE);
16   }
17   public void await(final long v) {
18     eventMap.putIfAbsent(v, newEDC());
19     EventDrivenControl edc = eventMap.get(v);
20     suspend(edc);
21   }
22   public long read() {
23     return eventCounter.get();
24   }
25 }
26
27 public class EventCountPrimer {
28   public static void main(String[] args) {
29     finish(() -> {
30       final EventCount ec = new EventCount();
31       forasync(1, 10, (i) -> {
32         println("Task-" + i);
33         ec.advance();
34       });
35       ec.await(5);
36       println("At least 5 iterations completed");
37     });
38   }
39 }
```

Figure 8: Implementation of the eventcount synchronization construct using `EventDrivenControls`. The example program in `EventCountPrimer` shows the use of the `advance` and `await` operations. The `await` at line 35 waits for *any* 5 iterations from the `forasync` loop to complete.

the `suspend` method on the EDC (line 20). If the EDC has already been resolved, the `suspend` method will return immediately. Otherwise, the `suspend` method will block until an appropriate call to `advance` causes the count to reach `v` and the EDC is resolved. The `read` operation is the simplest to implement; it looks up the value of the counter and returns it. Note that the `await` operation is blocking (since it invokes `suspend`), our runtime will allow a task executing the `await` operation to benefit from the optimization described in Section 4.3.

4.3 Handling Blocking Operations

In the presence of synchronization points from constructs such as futures, barriers, and phasers, our implementation reverts to thread-blocking scheduling of tasks. The scheduler then spawns additional worker threads to compensate for blocked worker threads. But this adds to overhead in the runtime as each thread needs its own system resources. Our implementation minimizes the overheads of blocking by attempting to make progress in the overall computation before blocking the worker thread. When a thread detects it is going to block (in the `EventDrivenControl.suspend()` method call), it searches for 'non-blocking' tasks and executes them before blocking. This requires HJlib to distinguish between tasks that contain blocking operations and those that do not, we enforce that by having users implement different functional interfaces for the bodies of their tasks. This is a useful optimization since the user does not need to provide an explicit continuation task when a blocking condition is discovered. Often executing these non-blocking tasks can

cause the EDC to get resolved and the `suspend()` method can return without blocking the worker thread. The optimization is safe since executing a non-blocking task is always guaranteed to complete without blocking. This disallows a deadlock scenario where a blocking point is reached which can only be resolved by executing a task earlier in the call stack. The runtime ensures that non-blocking tasks do not inadvertently call blocking operations and reports a runtime error if this occurs.

```
1  public static void suspend(EventDrivenControl edc) {
2    // execute tasks, return early if EDC is resolved
3    while (!edc.isValueAvailable() &&
4        !nbTasks.isEmpty()) {
5      final Task task = nbTasks.pop();
6      task.run();
7    }
8    // check once again, attempt to avoid blocking
9    if (edc.valueAvailable()) {
10     return;
11   }
12   // Use latch and block until EDC is resolved
13   final CountDownLatch cdl = new CountDownLatch(1);
14   edc.addExecutionBody((v) -> cdl.countDown());
15   // arrange for spawning additional worker thread
16   // before blocking the worker thread
17   handleAwait(cdl);
18 }
```

Figure 9: Algorithm for EDC `suspend` method. The runtime tries to execute non-blocking tasks before blocking the worker thread.

The algorithm for the `suspend()` method is displayed in Figure 9. The runtime tries to execute non-blocking tasks before blocking the worker thread (lines 4-7). If while executing the non-blocking tasks, the EDC gets resolved, the `suspend` method returns (line 10) without blocking the thread. If all the available non-blocking tasks get executed and the EDC has still not resolved, the runtime goes ahead to block the worker thread until the EDC is resolved (line 17). Our implementation uses a `CountDownLatch` to actually block the thread by making a call to the `await` method.

5. Abstract Execution Metrics

In order to effectively parallelize a program, we need to know which parts of a program take the most computation time. We also need to know the computation time required by different fragments of a program. Using actual computation time may not be repeatable as it includes system times such as delays due to cache misses and thread context switches. As an alternate we can measure the performance by measuring the number of abstract operations performed. These operations can be weighted combinations of floating-point operations, comparison operations, stencil operations, or any other data structure operations. We call such metrics for performance measurement abstract execution metrics (AEM). The AEM can be particularly useful when debugging performance problems, and when comparing alternate implementations of an application (e.g. sequential vs. parallel implementation). For example, students in the COMP 322 course first reason about the performance of their parallel programs (e.g. array sum, quicksort, etc.) by analyzing the AEM before looking at real execution times for performance later in the course. The main advantage of using abstract execution times is that the performance metrics will be the same regardless of which physical machine the HJlib program is executed on. The main disadvantage is that the abstraction may not be representative of actual performance on a given machine.

In addition to tracking the total number of abstract operations performed by a computation, the HJlib runtime also computes the critical path length (CPL) of the computation. The runtime dynamically forms the directed-acyclic graph that represents the computation while scheduling tasks and the critical path is the longest necessary path through this graph when respecting dependences between tasks. The critical path is useful since it determines the shortest time (in terms of operations) possible to complete the entire computation.

The HJlib runtime provides an API for the programmer to register and request AEM. The programmer can insert a call of the form, `doWork(N)`, anywhere in a task to indicate execution of `N` application-specific abstract operations. Multiple calls to `doWork()` are permitted within the same task, they have the effect of adding to the abstract execution time of that task. The HJlib runtime maintains metrics for each individual task and coordinates the logic for aggregating the metrics among tasks for the various synchronization constructs (i.e. futures, phasers, `isolated`, actors, etc.). Like the EDC API, an AEM API is also exposed for users to implement metrics support for their custom synchronization constructs. Using this API, the metrics for the overall program can be found by querying for the AEM after execution of the program.

The `-Dhj.abstractMetrics=true` option is used when executing an HJlib program to enable AEM. The metrics for a given task captures the total number of operations executed (WORK) and the critical path length (CPL) of the call graph generated by the program execution for that specific task. The ratio, WORK/CPL, is also printed as a measure of the maximum possible speedup for the program from its dynamic computation graph. This ratio is useful for programmers while tuning their application for parallel speedup; it can be viewed as the maximum performance improvement factor due to parallelism that can be obtained if we ideally had an unbounded number of processors. The AEM for a task can be obtained and printed using the following code snippet:

```
final HjMetrics metrics = abstractMetrics();
AbstractMetricsManager.dumpStatistics(metrics);
```

5.1 Computing Critical Path Length

Non-blocking (`async`) task The CPL of a non-blocking task is trivial to compute. It simply equals the total work (represented by calls to `doWork()`) done in the task. Blocking tasks contain other synchronization constructs, their CPL computation is described in the following paragraphs.

Finish scope Naively, one would assume the CPL of a `finish` scope to be the maximum of the CPLs of the tasks spawned inside the `finish` scope. However, one must take care of the sequential part of the computation inside the `finish` block and the dependence restrictions among the spawned tasks. The *representative* CPL of each task inside a `finish` scope consists of the sum of three fragments:

- Start CPL: This is the CPL inside the `finish` scope before the task starts executing. It is the maximum of two values: the CPL of the fragment executed inside the finish scope before this task is spawned; and the CPL of dependences that delay the task from starting execution.
- Task CPL: The actual CPL of the task's body when it is executed.
- End-Continuation CPL: This is the CPL of any computation inside the `finish` scope that is executed after the task ends execution.

The CPL of a `finish` scope is the maximum of the CPL of the sequential part and the maximum of the representative CPLs of all tasks spawned inside the `finish` scope.

Future and Data-Driven Future Computing the CPL of a future task follows the same rules as computing the CPL of an asynchronous task. It equals the sum of the Start CPL and the Task CPL, we call this the *completion* CPL. The more interesting case

is while computing the CPL at the point where the `get()` operation is invoked. Whenever a task invokes a `get()` operation on a future, the CPL of the task needs to be updated to the maximum of its current CPL and the completion CPL of the future's producer task. In contrast, computing the CPL while using data-driven futures is much simpler since the start of the task is delayed due to input dependences of the `asyncAwait` clause. So, the CPL of the `get` operation is already accounted for in the start CPL of the task.

```
1  public class FutureMetrics {
2    public static void main(String[] args) {
3      HjSystemProperty.abstractMetrics.set(true);
4      finish(() -> {
5        finish(() -> {
6          HjFuture<Integer> f1 = future(() -> {
7            doWork(2);
8            return 10;
9          });
10         async(() -> {
11           doWork(1);
12           f1.get();
13           doWork(3);
14         });
15       });
16       HjMetrics actualMetrics = abstractMetrics();
17       // Metrics: WORK=6, CPL=5
18     });
19   }
20 }
```

Figure 10: Abstract metrics for a program that uses futures.

Figure 10 displays an example HJlib program that report metrics while using futures. The future created at line 6 does two units of work before resolving the value of the future (by returning 10). The `async` created at line 10 does one unit of work before trying to retrieve the value of the future. At this point, the CPL for the task is updated to two (maximum between CPL of the task and completion CPL of the future). As the task completes execution, it does three more units of work and the CPL of the task is updated to five. When the `finish` scope completes, it updates its CPL to the maximum of all the tasks and sets the CPL to five. The total work done in the program is the aggregate of all the arguments passed to `doWork()` and is six in this example. As a result, the maximum parallel speedup that can be obtained from this program is 1.20.

Phasers Tasks that use phasers are part of multi-phased computations involving other tasks. In such computations, the CPL needs to be updated at each phase for all the participating tasks. Phasers produce two main events, `signal` and `wait`. At each of these events, a local metric object associated with the phaser needs to be updated for its CPL (and WORK). When a task enters the `signal` state on a phaser (by calling the `doSignal()` method), the task needs to update the CPL of the metric associated with the phase to the maximum of the task's current CPL and the existing CPL of the phaser. When a task enters the `wait` state on a phaser (by calling the `doWait()` or `next()` methods), the task needs to update its own CPL with the maximum of its current CPL and the CPL value of the phaser at that phase.

Figure 11 displays an HJlib program that reports abstract metrics for phaser-based programs. The two `asyncs` at line 7 and line 14 are registered in `SIG_WAIT` mode and hence treat the `next()` operation as a barrier. As a result, the computation has three phases delineated by the two `next()` operations. The total work done in the application is 18. However, the CPL is computed as the CPL of each individual phase. The `async` at line 14 provides the CPL for the first phase (5 units) and third phase (3 units). The `async` at line 7 provides the CPL for the second phase (5 units). Consequently, the CPL of the program is the sum of the three CPLs, i.e. 13 units .

```
1  public class PhaserMetrics {
2    public static void main(String[] args) {
3      HjSystemProperty.abstractMetrics.set(true);
4      finish(() -> {
5        finish(() -> {
6          HjPhaser ph = newPhaser(SIG_WAIT);
7          asyncPhased(ph.inMode(SIG_WAIT), () -> {
8            doWork(1);
9            next();
10           doWork(5);
11           next();
12           doWork(3);
13         });
14         asyncPhased(ph.inMode(SIG_WAIT), () -> {
15           doWork(3);
16           next();
17           doWork(1);
18           next();
19           doWork(5);
20         });
21       });
22       HjMetrics actualMetrics = abstractMetrics();
23       // Metrics: WORK=18, CPL=13
24     });
25   }
26 }
```

Figure 11: Abstract metrics for a program that uses phasers.

Actors Like phasers do for each phase of phased-computations, each actor maintains its own metric object to store the CPL and work. The two main events associated with actors are sending of messages and processing of messages, these events affect the CPL of an actor-based computation. When a message is sent to an actor, the message is packaged with a metric object wrapping the CPL of the sender. As a message is picked to be processed from its mailbox, the actor updates its own CPL to the maximum of the current CPL and the CPL of the message sender. Then as this message is processed by the actor, the CPL is updated like the CPL for any other task. When the actor finally exits, it notifies its enclosing `finish` scope of its total work and CPL. This ends up updating the CPL of the `finish` scope as mentioned earlier.

Isolated Since HJlib maintains multiple locks to support both global and object-based isolation, a metric object is maintained for each lock. When the runtime uses a lock in some `isolated` code fragment, it serializes the computation for other code fragments that will use the same lock. As each lock is successfully acquired, the CPL of the task's metric is updated to the maximum value using the lock's CPL. After the `isolated` block completes execution and before releasing the lock, the CPL from lock's metric is updated to the maximum of the current value and the task's CPL.

Figure 12 displays an HJlib program that reports AEM for programs using `isolated` statements. Lines 6 and 13 create two `asyncs` that each do 7 units of work. In the absence of the `isolated` statements the CPL of the program would be 7 units. However, the presence of the `isolated` statements causes serialization of the two fragments that each do 3 units of work. As a result, the CPL of the overall computation goes up to 10 units.

6. Deadlock Detection

A deadlock is caused when one or more tasks come into conflict over some resource, in such a way that no further execution is possible for at least one of the tasks. Deadlock prevention is the name given to schemes that guarantee that deadlocks can never happen because of the way the synchronization constructs are structured. No HJlib program using `async`, `finish`, `isolated`, and `phaser`[2]

[2] Using only the `next()` operation

```
1  public class IsolatedMetrics {
2    public static void main(String[] args) {
3      HjSystemProperty.abstractMetrics.set(true);
4      finish(() -> {
5        finish(() -> {
6          async(() -> {
7            doWork(2);
8            isolated(() -> {
9              doWork(3);
10           });
11           doWork(2);
12         });
13         async(() -> {
14           doWork(2);
15           isolated(() -> {
16             doWork(3);
17           });
18           doWork(2);
19         });
20       });
21       HjMetrics actualMetrics = abstractMetrics();
22       // Metrics: WORK=14, CPL=10
23     });
24   }
25 }
```

Figure 12: Abstract metrics for a program that uses `isolated`.

constructs can create a logical deadlock cycle. However, programmers can use some of the other constructs (DDFs, actors, eventcounts, custom synchronization constructs built using EDCs, etc.) to write programs that deadlock. For example, programs written using actors might forget to terminate the actor or programs using DDFs may have faulty logic that never resolves the DDF causing awaiting `async` tasks to be blocked forever.

The HJlib runtime follows the deadlock detection scheme where deadlocks are allowed to occur. The runtime continuously examines the computation to detect that a deadlock has occurred. In our current implementation, the runtime doesn't attempt to correct the deadlock, instead the runtime reports a diagnostic error message with tasks participating in the deadlock and terminates the program. When configured to do so, the runtime also includes the exact line number of the source code where each task is blocked in its computation. Since the blocked tasks are available, they can be queried for their stack trace just before blocking to obtain the user code that caused the blocking operation. When the deadlock error is reported, this diagnostic information can be provided to the user to allow debugging of the program.

Our basic algorithm for detecting deadlocks relies on tracking the number of ready tasks in the work queue and the number of blocked or pending tasks. Pending tasks are identified by tasks that have been created, but not scheduled for execution. Blocked tasks are discovered by their use of the `suspend()` method on EDCs. Since all our synchronization constructs are built using EDCs, all constructs are supported by the deadlock detector. Whenever a task blocks it is added to a blocked tasks collection, when it gets unblocked it is removed from the collection. If the work queue becomes empty and there are no more actively executing tasks on the worker threads, there is no more computational progress being made by the program. In such a state, if the blocked tasks collection is non-empty, then we have a deadlock. The runtime immediately discovers this state (i.e. it does not use a polling strategy at intermittent intervals) and reports it. The drawback of this approach is that the runtime needs to wait for all ready tasks to complete executing before reporting the deadlock.

Figure 13 displays an example of a deadlock when writing HJlib programs with DDFs. The programs deadlocks as tasks at line 8 and line 12 are never executed as their dependences are never satisfied. The program instantiates two DDFs, but there is no execution flow

```
1  public class DdfDeadlockExample {
2    public static void main(String[] args) {
3      HjSystemProperty.trackDeadlocks.set(true);
4      finish(() -> {
5        HjDataDrivenFuture<Integer> right = newDDF();
6        HjDataDrivenFuture<Integer> left = newDDF();
7        // waits on left before resolving right
8        asyncAwait(left, () -> {
9          right.put(1);
10       });
11       // waits on right before resolving left
12       asyncAwait(right, () -> {
13         left.put(2);
14       });
15   });
16   }
17 }
```

Figure 13: A simple HJlib program that deadlocks because the `asyncAwait` tasks at line 8 and line 12 are never executed. As a result the `finish` at line 4 cannot complete.

that resolves these DDFs at lines 8 and 12. Since the tasks do not execute, the `finish` at line 4 cannot complete the join operation and no computational progress can be made. The deadlock detector will identify this scenario as soon as line 15 is executed and report that *three* tasks are involved in a deadlock. These tasks are the main task running the `finish` block and the two `asyncAwait` tasks.

```
1  public class PhaserDeadlockExample {
2    public static void main(String[] args) {
3      HjSystemProperty.trackDeadlocks.set(true);
4      finish(() -> {
5        HjPhaser ph1 = newPhaser(SIG_WAIT);
6        HjPhaser ph2 = newPhaser(SIG_WAIT);
7        asyncPhased(ph1.in(WAIT), ph2.in(SIG), () -> {
8          ph1.doWait(); {
9          ph2.signal(); {
10       });
11       asyncPhased(ph1.in(SIG), ph2.in(WAIT), () -> {
12         ph2.doWait(); {
13         ph1.signal(); {
14       });
15     });
16   }
17 }
```

Figure 14: HJlib program that deadlocks when using phasers with explicit `signal` and `wait` operations.

No HJlib program using phasers and only `next()` operations can create a logical deadlock cycle. However, this guarantee is lost when users write programs that call the `signal` and `wait` operations explicitly. Figure 14 displays such a program that deadlocks due to incorrect use of the `signal` and `wait` operations. Line 7 create an `async` task that is registered in wait mode on the first phaser and in signal mode on the second phaser. Similarly, line 11 create an `async` task that is registered in signal mode on the first phaser and in wait mode on the second phaser. Both tasks perform the wait operation in the current phase as the first line of their computation (lines 8 and 12). Since the corresponding phasers have pending signals for the current phase (lines 9 and 13), both the tasks block indefinitely and the program deadlocks. The `doWait()` operations use EDCs under the covers which allows the deadlock detector to identify the blocked tasks.

Figure 15 displays a program that deadlocks while using custom SyncCons implemented by a user, e.g. the eventcount construct described in Section 4.2. Since the `await` operation is implemented using EDCs, the deadlock detector can track when the task using eventcounts get blocked. Line 6 creates ten tasks that advance the value of `eventCount` to ten. However, `async` task created at line 9 is waiting for the `eventCount` to reach a value of twenty

```
1  public class EventCountDeadlockExample {
2    public static void main(String[] args) {
3      HjSystemProperty.trackDeadlocks.set(true);
4      finish(() -> {
5        HjEventCount eventCount = newEventCount();
6        forasync(1, 10, (i) -> {
7          eventCount.advance();
8        });
9        async(() -> {
10         eventCount.await(20);
11       });
12     });
13   }
14 }
```

Figure 15: HJlib program that deadlocks when using eventcounts.

(at line 10). As a result, this task will block indefinitely and the deadlock detector can report the deadlock.

7. Examples, Benchmarks, and Applications

Parallel programming is an inherently complex intellectual activity. Being a pedagogic language, it is imperative HJlib have good documentation and useful beginner examples. The COMP 322 course lectures and videos are available online and explain the various parallel constructs in HJlib. They also include online lab exercises for interested users to write HJlib programs using various constructs. In addition, HJlib has extensive documentation [5] and examples available to help users get started.

Experience has shown that porting existing Java code to HJlib is trivial. If the application already exhibits parallelism, it can be as simple as adding `finish` and `async` keywords or transforming a for loop into a `forall` loop. COMP 322 students use these constructs and abstract metrics support in their first couple of assignments to start writing parallel programs in HJlib. Being a standard Java library, the HJlib programs can be written in any standard Java IDE. In addition, being able to use the debugger support from IDEs greatly helps users when they write their own example applications and try and understand debug their programs.

COMP 322 students wrote a handful of HJlib programs as their lab exercises or programming assignments with the level of difficulty increasing as the semester progressed. Students start out with applications that are easier to parallelize such as NQueens and matrix multiplication which requires only `async-finish` or `forall` constructs to be added to the sequential programs. Students were required to report their parallel speedup numbers in terms of execution time for many of these exercises by running their programs on an 8 core (two quad-core) 2.83GHz Intel Xeon Harpertown CPU with 16GB RAM per node. For example, by parallelizing PI computation using actors students reported speedup around 6.4×. In another exercise to parallelize algebraic expression evaluation, speedups of around 4× were obtained by using DDFs. Parallelizing an implementation of SmithWaterman where students were allowed to use any of the HJlib parallel constructs, students reported speedup around 6.9×. Finally, in an exercise where students were required to parallelize solving Sudoku puzzles by using any HJlib construct as well as changing the main algorithm, students reported speedups over 15×. Speedups greater than 8 could be achieved since the total work done was different between the sequential and parallel algorithms.

HJlib includes a large set of benchmarks for the full gamut of parallel constructs it supports. In addition to micro-benchmarks for `async-finish` style programs, it also include benchmarks for phasers, futures, DDFs, and actors. Some Java programs (Java Grande Forum [17], Shootout [1]) and benchmark suites (IMSuite [22], STAMP [10]) have also been ported to HJlib. Applications such as LULESH [26] and some of PBBS [39] applications have also been ported to HJlib from their native implementations.

The availability of these benchmarks and applications provide an excellent platform to test out different runtime implementations in HJlib. Additional runtimes can be implemented independently and configured to be used by HJlib. Currently, we are implementing a cooperative runtime [25] which avoids all thread-blocking operations. Being able to compare the performance of benchmarks across runtimes helps us tune our implementations.

8. Related Work

Modern languages and libraries provide lightweight dynamic task parallel execution models for improved programmer productivity. There are a handful of library extensions that provides parallel APIs, such as Intel Threading Building Blocks (TBB) [36], Java Concurrency Utilities (JUC) [6] and the Microsoft Task Parallel Library (TPL) [28]. TBB, JUC, and TPL expose constructs to create asynchronous tasks and futures. JUC also provides support for constructs similar to phasers. The focus of these libraries is an industrial quality product and they are not necessarily amenable for pedagogic interests. For example, they support far fewer parallel constructs; do not include support for AEM or deadlock detection; and have no clear description of how to implement newer synchronization constructs on them.

The Streams API [18] in Java 8 greatly improves the processing of elements from collections and other sources and aids in writing data parallel programs. The Streams API is largely present to make the Collections Framework parallel-friendly. The Streams API also offers reduction operations such as reduce, collect, summation, maximum, and count. HJlib focuses on task parallelism, but such data parallel constructs can easily be implemented. In fact, such data parallel constructs have been added as an extension to HJlib while implementing the PBBS benchmarks [39].

The GPars framework [42] offers Java developers intuitive and safe ways to handle Java or Groovy tasks concurrently. GPars has a larger range of parallel constructs than HJlib, it supports dataflow concurrency, actor programming model, communicating sequential processes, fork/join paralleism, etc. It is not always safe for different parallel constructs to be integrated while writing programs with GPars. GPars relies on external libraries to offer support for many of the parallel constructs. HJlib, on the other hand, presents a unified framework where all the parallel constructs can be safely composed. In addition, HJlib has no third-party dependencies and all the parallel constructs are implemented using the standard JDK and the `EventDrivenControl` API.

There are also new parallel languages such as X10 [14, 16], Chapel [13], and Fortress [8]. These languages provide constructs to support fork-join style computations, but suffer from the drawback that they are new languages and require their own compiler and toolchain. Of these languages, Chapel has been used most successfully as a teaching language for parallel programming [3]. One of Chapel's original design goals was ease of use, which is exploited to quickly introduce the language in the context of an algorithms course to demonstrate parallelism in divide and conquer algorithms. In addition there are languages that are extensions to existing languages, such as the Cilk [9] and OpenMP 3.0 [32] extensions to C. Compared to Cilk's spawn-sync computations (and OpenMP parallel for loops) which must be fully-strict, HJlib's `async-finish` computations are terminally-strict but need not be fully-strict. HJlib also has a number of other constructs that are not all supported by these languages in a unified manner e.g., futures, phasers, isolated, and actors.

9. Summary and Future Work

In this paper, we introduced Habanero-Java library (HJlib), which is a pure library implementation of our pedagogic parallel programming model. HJlib is built using lambda expressions and can run on any Java 8 JVM. While programmers focus on the design of their application, the HJlib runtime delivers performance and scalability with respect to the available number of cores as well as the nature of parallelism that an application exhibits. HJlib offers the `EventDrivenControl` API which can be used to implement custom synchronization constructs while executing asynchronous tasks managed by the HJlib runtime. Finally, the HJlib runtime feedback capabilities such as the AEM and the deadlock detector help the programmer to get feedback on theoretical performance as well as the presence of potential bugs in their program.

The usability of HJlib has been assessed in various occasions when porting benchmarks from Java, C, or Fortran to HJ. HJlib has also been used in the introductory parallel programming class for second-year undergraduate students at Rice University (COMP 322). It has 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. By relying on simple orthogonal parallel constructs with important safety properties, HJlib 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.

Being an implementation of a pedagogic programming model, HJlib is also an attractive tool for both educators and researchers. HJlib is actively being used in multiple research projects at Rice and also by external independent collaborators. We are currently working on building a cooperative runtime for HJlib which will allow blocking operations in synchronization constructs to be implemented without blocking the underlying worker thread. In another project, we are extending the runtime to support eureka-style computations. The AEM facility is being extended to provide graphical support in the form of computation graphs. HJlib parallel constructs are being used in the implementation of the emulator for Fresh Breeze multiprocessor architecture to speedup the emulation process itself. There are also plans to include HJ's data race detector into HJlib and in exploiting multicore parallelism in Hadoop applications [43].

We are also working on extending the AEM support with computation graphs (CGs) for HJlib programs. The CG provides an intuitive graphical view of the parallel program's execution. A CG is an acyclic graph that consists of:

- A set of nodes, where each node represents a step consisting of an arbitrary sequential computation. A task can be partitioned into multiple steps, the key constraint is that a step should not contain any parallelism or synchronization. Each node represents some amount of work units aggregated from the calls to `doWork()`.
- A set of directed edges that represent ordering constraints among steps.

The CG can be used to determine if two steps may execute in parallel with each other. The CG also identifies dependence relations between steps and makes it easier to visualize the critical path of the program.

Acknowledgments: This material is based upon work supported by the National Science Foundation under Grant No. 1242507. 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 are grateful to Vincent Cavé, Jisheng Zhao, and Jun Shirako for discussions on the Habanero Java runtime system, isolated blocks, and phasers, respectively. We are grateful to the anonymous reviewers whose feedback on an earlier draft helped improve the presentation of this paper.

References

[1] The Computer Language Benchmarks Game. URL http://benchmarksgame.alioth.debian.org/.

[2] Habanero Extreme Scale Software Research Project, 2014. URL https://wiki.rice.edu/confluence/display/HABANERO/Habanero+Extreme+Scale+Software+Research+Project.

[3] Chapel Educator Resources, 2014. URL http://chapel.cray.com/education.html.

[4] COMP 322: Fundamentals of Parallel Programming, 2014. URL https://wiki.rice.edu/confluence/display/PARPROG/COMP322.

[5] Habanero-Java Library Javadoc, 2014. URL http://www.cs.rice.edu/~vs3/hjlib/doc/.

[6] Java Concurrency Utilities, 2014. URL http://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/.

[7] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-01092-5.

[8] E. Allan, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, G. L. Ryu, Sukyoung Steele Jr., and S. Tobin-Hochstadt. The Fortress Language Specification Version 1.0, March 2008. URL http://research.sun.com/projects/plrg/fortress.pdf.

[9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM. ISBN 0-89791-700-6. .

[10] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[11] V. Cavé, Z. Budimlić, and V. 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, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0547-1. . URL http://doi.acm.org/10.1145/1937117.1937126.

[12] V. Cavé, J. Zhao, Y. Guo, and V. Sarkar. Habanero-Java: the New Adventures of Old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0935-6. . URL http://doi.acm.org/10.1145/2093157.2093165.

[13] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312, August 2007. ISSN 1094-3420. . URL http://dx.doi.org/10.1177/1094342007078442.

[14] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-Oriented Approach to Non-uniform Cluster Computing. *SIGPLAN Not.*, 40: 519–538, Oct. 2005. ISSN 0362-1340. .

[15] S. Davies, J. A. Polack-Wahl, and K. Anewalt. A Snapshot of Current Practices in Teaching the Introductory Programming Sequence. In T. J. Cortina, E. L. Walker, L. A. S. King, and D. R. Musicant, editors, *Proceedings of the 42Nd ACM Technical Symposium on Computer Science Education*, SIGCSE '11, pages 625–630. ACM, 2011. ISBN 978-1-4503-0500-6. . URL http://doi.acm.org/10.1145/1953163.1953339.

[16] K. Ebcioglu, V. Saraswat, and V. Sarkar. X10: An Experimental Language for High Productivity Programming of Scalable Systems. *In Proceedings of the Second Workshop on Productivity and Performance in High-End Computing (PPHEC-05)*, January 2005. URL http://dist.codehaus.org/x10/documentation/papers/P-PHEC05-paper.pdf.

[17] EPCC. The Java Grande Forum Multi-threaded Benchmarks. URL http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/threads/s1contents.html.

[18] Goetz, Brian. State of the Lambda: Libraries Edition, 2013. URL http://cr.openjdk.java.net/~briangoetz/lambda/lambda-libraries-final.html.

[19] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. *SIGOPS Oper. Syst. Rev.*, 40:151–162, October 2006. ISSN 0163-5980.

[20] M. Grossman, M. Breternitz, and V. Sarkar. HadoopCL: MapReduce on Distributed Heterogeneous Platforms Through Seamless Integration of Hadoop and OpenCL. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pages 1918–1927, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4979-8. . URL http://dx.doi.org/10.1109/IPDPSW.2013.246.

[21] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-First and Help-First Scheduling Policies for Async-Finish Task Parallelism. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3751-1. .

[22] S. Gupta and V. K. Nandivada. IMSuite: A Benchmark Suite for Simulating Distributed Algorithms. *CoRR*, abs/1310.2814, 2013.

[23] R. H. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7:501–538, October 1985. ISSN 0164-0925. .

[24] S. Imam and V. Sarkar. Integrating Task Parallelism with Actors. In *Proceedings of the ACM international conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 753–772, New York, NY, USA, 2012. ACM. . URL http://doi.acm.org/10.1145/2384616.2384671.

[25] S. Imam and V. Sarkar. Cooperative Scheduling of Parallel Tasks with General Synchronization Patterns. In *Proceedings of the 28th European conference on Object-Oriented Programming* , ECOOP '14, New York, NY, USA, 2014. ACM.

[26] I. Karlin, J. Keasler, and R. Neely. LULESH 2.0 Updates and Changes. Technical Report LLNL-TR-641973, August 2013.

[27] D. Lea. A Java Fork/Join Framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM. . URL http://doi.acm.org/10.1145/337449.337465.

[28] D. Leijen, W. Schulte, and S. Burckhardt. The Design of a Task Parallel Library. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 227–242, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. . URL http://doi.acm.org/10.1145/1640089.1640106.

[29] Luontola, Esko. Retrolambda: Use Lambdas on Java 7, 2013. URL https://github.com/orfjackal/retrolambda.

[30] C. G. Mason, M. and M. Raadt. Trends in Introductory Programming Courses in Australian Universities - Languages, Environments and Pedagogy. In M. Raadt and A. Carbone, editors, *Australasian Computing Education Conference (ACE2012)*, volume 123 of *CRPIT*, pages 33–42, Melbourne, Australia, 2012. ACS. URL http://crpit.com/confpapers/CRPITV123Mason.pdf.

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

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

[33] J. Ponge. Fork and Join: Java Can Excel at Painless Parallel Programming Too!, 2011. URL http://www.oracle.com/technetwork/articles/java/fork-join-422606.html.

[34] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Efficient Data Race Detection for Async-Finish Parallelism. In *Proceedings of the First international conference on Runtime verification*, RV'10, pages 368–383, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-16611-3, 978-3-642-16611-2.

[35] D. P. Reed and R. K. Kanodia. Synchronization with Eventcounts and Sequencers. *Commun. ACM*, 22(2):115–123, Feb. 1979. ISSN 0001-0782. . URL http://doi.acm.org/10.1145/359060.359076.

[36] J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007. ISBN 9780596514808.

[37] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phasers: a Unified Deadlock-Free Construct for Collective and Point-to-Point Synchronization. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, ICS '08, pages 277–288, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-158-3. . URL http://doi.acm.org/10.1145/1375527.1375568.

[38] J. Shirako, V. Cavé, J. Zhao, and V. Sarkar. Finish Accumulators: An Efficient Reduction Construct for Dynamic Task Parallelism. In *The 4th Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, March 2013.

[39] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief Announcement: The Problem Based Benchmark Suite. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 68–70, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1213-4. . URL http://doi.acm.org/10.1145/2312005.2312018.

[40] R. Surendran, R. Raman, S. Chaudhuri, J. Mellor-Crummey, and V. Sarkar. Test-Driven Repair of Data Races in Structured Parallel Programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 15–25, New York, NY, USA, June 2014. ACM. ISBN 978-1-4503-2784-8. . URL http://doi.acm.org/10.1145/2594291.2594335.

[41] S. Taşırlar and V. Sarkar. Data-Driven Tasks and their Implementation. In *Proceedings of the International Conference on Parallel Processing (ICPP) 2011*, ICPP '11, pages 652–661, Washington, DC, USA, September 2011. IEEE Computer Society. ISBN 978-0-7695-4510-3. . URL http://dx.doi.org/10.1109/ICPP.2011.87.

[42] The GPars team. The GPars Project - Reference Documentation, 2014. URL http://www.gpars.org/guide/.

[43] Zhang, Yunming and Cox, Alan and Sarkar, Vivek. HJ-Hadoop: An Optimized MapReduce Runtime for Multi-core Systems. In *5th USENIX Workshop on Hot Topics in Parallelism (HotPar '13)*, June 2013. Accepted as poster with accompanying paper.