Comparing the Usability of Library vs. Language Approaches to Task Parallelism

Vincent Cavé
Rice University
vcave@rice.edu

Zoran Budimlić
Rice University
zoran@rice.edu

Vivek Sarkar
Rice University
vsarkar@rice.edu

Abstract

In this paper, we compare the usability of a library approach with a language approach to task parallelism. There are many practical advantages and disadvantages to both approaches. A key advantage of a library-based approach is that it can be deployed without requiring any change in the tool chain, including compilers and IDEs. However, the use of library APIs to express all aspects of task parallelism can lead to code that is hard to understand and modify. 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 constructs and (possibly) of new keywords. In this paper, we compare the `java.util.concurrent` (`j.u.c`) library [10] from Java 7 and the Habanero-Java (HJ) [13] language, supported by our experiences in teaching both models at Rice University.

1. Introduction

The computer industry is at a major inflection point due to the end of a decades-long trend of exponentially increasing processor clock frequencies. It is widely agreed that parallelism in the form of multiple power-efficient cores must be exploited to compensate for this lack of frequency scaling. Unlike previous cases of hardware evolution, this shift towards homogeneous and heterogeneous manycore computing will have a profound impact on software. Two complementary approaches to address this problem are 1) the use of *implicitly-parallel high-level programming models* such as the map-reduce pattern [2] and data parallel languages such as Ct, NESL [1], and Matlab, and 2) the use of *task-parallel programming models* such as Java Concurrency [10], Intel Threading Building Blocks (TBB), .Net Task Parallel Library [17], OpenMP 3.0, Cilk [8], X10 [7], and Habanero-Java (HJ) [13]. In this paper, we focus on task-parallel programming models, and study the usability of library vs. language approaches to task parallelism. The comparison in this paper is performed between the `java.util.concurrent` (`j.u.c`) library [10] from Java 7 and the Habanero-Java (HJ) [13] language, and is supported by our experiences in teaching both models at Rice University.

There are many practical advantages and disadvantages to both approaches. A key advantage of a library-based approach to task parallelism is that it can be deployed without requiring any change in the tool chain including compilers and IDEs. However, the use of library APIs to express all aspects of task parallelism can lead to code that is hard to understand and modify. 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. We try to highlight the pros and cons of each approach in terms of usability, with a focus on how task creation, scheduling, and synchronization are expressed in both approaches.

The rest of the paper is organized as follows. Section 2 includes background on the `j.u.c` package and the HJ language. Sections 3–5 compare how the Habanero-Java language and the Java Concurrent Utilities library express the notions of tasks, their synchronization and parallel loop processing. Section 6 discusses usability issues with respect to code migration and porting, and Section 7 contains our conclusions.

2. Background

2.1 java.util.concurrent package

The `java.util.concurrent` (`j.u.c`) package grew out of JSR 166, and has been included in all Java releases since Java 5.0 [10]. It builds on the lower-level Java `Thread` construct and `Runnable` interface in standard Java. It includes an `Executor` framework with support for using *worker thread pools* to execute tasks as logical units of work. Each thread pool has an associated *work queue* that holds tasks waiting to be executed. Different kinds of thread pools can be created (*e.g.,* `newFixedThreadPool()`, `newSingleThreadExecutor()`) to enforce different execution policies. A `Callable` interface is also provided to support tasks that can return values and whose exceptions can be caught by application logic. A number of concurrent data structures are also provided including implementations of `Queue`, `BlockingQueue` as well as `ConcurrentHashMap`, `ConcurrentSkipListMap`, and `ConcurrentSkipListSet`. The `j.u.c` package includes *synchronizer* objects, such as `CountDownLatch`, `FutureTask`, `Semaphore`, and `CyclicBarrier` that can be used to help coordinate the execution of tasks. Alternatives to exclusive locks are also provided in the form of `ReadWriteLock`'s, `ReentrantLock`'s and *atomic variables* to reduce the impact of lock contention. The latest release of `j.u.c` in Java 7 includes `ForkJoinTasks` with support for work-stealing schedulers, and `Phaser` synchronizer objects which are derived in part [16] from the phaser construct in HJ.

While the `j.u.c` package contains a wide variety of choices to enforce different execution policies, we believe that its usability design is biased more towards advanced users than mainstream users. As an example, the following comment from [10] regarding the use

of `ReentrantLock` is a good indication of the target audience for `j.u.c`

> "`ReentrantLock` is an advanced tool for situations where intrinsic locking is not practical. Use it if you need its advanced features: timed, polled, or interruptible lock acquisition, fair queueing, or non-block-structured locking. Otherwise, prefer `synchronized`."

2.2 Habanero-Java (HJ)

The Habanero-Java (HJ) language [13] developed at Rice University was derived from X10 v1.5 [22][1], starting in 2007. The current HJ implementation uses Java v1.4 as its base language, but the concurrency extensions in HJ can be applied just as well to sequential programs written in Java 5 or Java 7. The code generated by the HJ compiler consists of Java classfiles that can be executed on any standard JVM. Likewise, the HJ runtime system is written in standard Java, and can also be executed on any standard JVM. A brief summary of the key constructs in HJ relevant to this paper is included below.

async: Async is a construct for creating a new asynchronous task. The statement `async` ⟨*stmt*⟩ causes the parent task to create a new child task to execute ⟨*stmt*⟩ (logically) in parallel with the parent task. ⟨*stmt*⟩ is permitted to read/write any data in the heap and to read (but not write) any local variable belonging to the parent task's lexical environment.

finish: The statement `finish` ⟨*stmt*⟩ causes the parent task to execute ⟨*stmt*⟩ and then wait until all sub-tasks created within ⟨*stmt*⟩ have terminated (including transitively spawned tasks). Operationally, each instruction executed in an HJ task has an unique *Immediately Enclosing Finish* (IEF) statement instance [4].

Besides termination detection, the `finish` statement plays an important role with regard to exception semantics. An HJ task may terminate normally or abruptly. A statement terminates abruptly when it throws an exception that is not handled within its scope, otherwise it terminates normally. If any *async* task terminates abruptly by throwing an exception, then its IEF statement also terminates abruptly and throws a *MultiException* [7] formed from the collection of all exceptions thrown by all abruptly-terminating tasks in the IEF.

future: HJ 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. (`Expr` may consist of a statement block ending with a return statement.) 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.

isolated: The *isolated* construct enables execution of a statement in isolation (mutual exclusion) relative to all other instances of isolated statements. The statement `isolated` ⟨*Stmt*⟩ executes ⟨*Stmt*⟩ in isolation with respect to other *isolated* statements. As advocated in [14], 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.

phasers: The *phaser* construct [4] 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* and the *phase-ordering* and *deadlock-freedom* safety properties, distinguish phasers from synchronization constructs in past work including barriers [12**?**], counting semaphores [18], and X10's clocks [7]. The latest release of `j.u.c` in Java 7 includes `Phaser` synchronizer objects which are derived in part [16] from the phaser construct in HJ.

In general, a task may be registered on multiple phasers, and a phaser may have multiple tasks registered on it. Two 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.

• *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 [4, 5]. We define the exception semantics of the single statement as follows: an exception thrown in the single statement causes all the tasks blocked on that `next` operation to terminate abruptly with a single instance of the exception thrown to the IEF task[2]. While our HJ phaser implementation also supports explicit `signal` and `wait` operations on phasers, it is important to point out that structuring the parallel program so that all the tasks use only the `next` operation for synchronization guarantees deadlock freedom among the synchronizing tasks, a key usability feature of HJ.

forall: 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. A *point* is an element of an n-dimensional Cartesian space ($n \geq 1$) with integer-valued coordinates. A *region* is a set of points, and can be used to specify an array allocation or an iteration range as in the case of `forall`.

Each dynamic instance of a `forall` statement 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[3]. 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.

3. Task Creation And Scheduling

3.1 Task Creation

The `j.u.c` library requires tasks to be objects implementing the Runnable interface. Generally, programmers can either create separate new classes, or anonymous inner classes implementing the Runnable interface. The first approach is a good programming practice in the sense that it forces the programmer to understand what are the inputs and outputs of the task since they need to be made available to the class. However, this tends to separate the code of the task from the context in which it will be executed. The second approach that uses anonymous inner classes places the code of the task much closer to the rest of the application code. However, anonymous inner class declarations degrade the readability of the code as they introduce a new type creation and method declarations right in the middle of the application code.

In contrast, the HJ *async Stmt* construct allows programmers to create a task anywhere in the code where parallelism is required.

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

[2] Since the scope of a phaser is limited to its IEF, all tasks registered on a phaser must have the same IEF.

[3] For readers familiar with the `foreach` statement in `X10` and HJ, 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*⟩".

```
public static ArrayList<Integer>
  quickSort(ArrayList<Integer> a) {
    if (a.isEmpty()) return new ArrayList<Integer>();
    ArrayList<Integer> l = new ArrayList<Integer>();
    ArrayList<Integer> r = new ArrayList<Integer>();
    ArrayList<Integer> m = new ArrayList<Integer>();
    for (Integer i : a)
      if ( i < a.get(0) ) l.add(i);
      else if ( i > a.get(0) ) r.add(i)
      else m.add(i);
    ArrayList<Integer> l_s = quickSort(l);
    ArrayList<Integer> r_s = quickSort(r);
    return l_s.addAll(m).addAll(r_s);
}
```

Figure 1. Original (sequential) Java version of Quicksort example

```
public static ArrayList<Integer>
  quickSort(ArrayList<Integer> a) {
    if (a.isEmpty()) return new ArrayList<Integer>();
    ArrayList<Integer> l = new ArrayList<Integer>();
    ArrayList<Integer> r = new ArrayList<Integer>();
    ArrayList<Integer> m = new ArrayList<Integer>();
    for (Integer i : a)
      if ( i < a.get(0) ) l.add(i);
      else if ( i > a.get(0) ) r.add(i)
      else m.add(i);
    final ArrayList<Integer> l_f = l, r_f = r;
    FutureTask<ArrayList<Integer>> l_t =
      new FutureTask<ArrayList<Integer>>(
        new Callable<ArrayList<Integer>>() {
          public ArrayList<Integer> call()
            { return quickSort(l_f); } } );
    FutureTask<ArrayList<Integer>> r_t =
      new FutureTask<ArrayList<Integer>>(
        new Callable<ArrayList<Integer>>() {
          public ArrayList<Integer> call()
            { return quickSort(r_f); } } );
    new Thread(left_t).start();
    new Thread(right_t).start();
    ArrayList<Integer> l_s = l_t.get();
    ArrayList<Integer> r_s = r_t.get();
    return l_s.addAll(m).addAll(r_s);
}
```

Figure 2. Parallel Java version of Quicksort example using `j.u.c`'s FutureTask

As in anonymous inner classes, the task has access to all variables declared in its enclosing lexical scope. However, the benefit of this language approach is that a simple keyword can be used, while the compiler takes care of lower-level code generation such as creating a new anonymous inner class.

As an example, Figure 1 shows a sequential Java implementation of the quicksort algorithm taught in the introductory programming class (COMP 211) in the Spring 2010 semester at Rice University. Figure 2 shows the corresponding Java Concurrency parallel version which was also taught in the same class with the intention of introducing students to simple examples of parallel programming. While the logic of the parallelism was easy for students to understand, all the additional syntax needed for `j.u.c`'s library interface in Figure 2 was a major source of confusion. (The example in Figure 2 will be even more complicated if an `Executor` or `ForkJoinTask` framework is used instead of creating a separate thread for each task.) In contrast, students learning HJ in the COMP 322 course at Rice University found it was much easier to understand the HJ version[4] in Figure 3 than the `j.u.c` version.

```
public static ArrayList<Integer>
  quickSort(ArrayList<Integer> a) {
    if (a.isEmpty()) return new ArrayList<Integer>();
    ArrayList<Integer> l = new ArrayList<Integer>();
    ArrayList<Integer> r = new ArrayList<Integer>();
    ArrayList<Integer> m = new ArrayList<Integer>();
    for (Integer i : a)
      if ( i < a.get(0) ) l.add(i);
      else if ( i > a.get(0) ) r.add(i)
      else m.add(i);
    final future<ArrayList<Integer>> l_t =
      async<ArrayList<Integer>> {return quickSort(l);};
    final future<ArrayList<Integer>> r_t =
      async<ArrayList<Integer>> {return quickSort(r);};
    ArrayList<Integer> l_s = l_t.get();
    ArrayList<Integer> r_s = r_t.get();
    return l_s.addAll(m).addAll(r_s);
}
```

Figure 3. Parallel Java version of Quicksort example using Habanero-Java

3.2 Task Scheduling

As highlighted in the previous section, in HJ, tasks are not explicit objects that the programmer can access or manipulate (*e.g.,* to change execution policies). An async is created and scheduled at the point it is declared in the program, but the programmer has no handle over it and should not make assumptions as to when the task will be executed and on which specific worker. (HJ instead offers the option of using hierarchical *places* [23] to support locality management policies, for advanced users interested in that level of control.)

This is a radically different approach from `j.u.c` which requires programmers to create tasks, store them and possibly modify them before submitting them to the scheduler. This responsibility is likely to become a source of bugs, either by omitting to execute some tasks or allowing a task representation to outlives its execution, which may lead to unexpected side effects if it was executed several times.

Although programmers used to parallelism can endeavor to follow good programming practices when using the `j.u.c` package, there are plenty of opportunities for mainstream programmers to shoot themselves in the foot. One strength of the language approach is that it can choose whether or not to expose task creation, whereas a library approach has to put some burden on the programmer to manage a task representation of some form. The HJ language design is motivated by usability for mainstream users rather than advanced users and tries to reduce opportunities for the programmer to make mistakes.

3.3 Task Execution

Both the `j.u.c` and HJ provide work-sharing and work-stealing task execution strategies. The `j.u.c` provides two sets of API that define various kind of work-sharing executors and a fork-join framework [15] based on work-stealing. The HJ language makes no assumptions on how the runtime will schedule tasks. With language

[4] The publicly available version of the HJ compiler does not fully support generic types as shown in Figure 3, but does support type declarations of the form `future<T>`. Full support for generic types is currently a work in progress for the HJ compiler.

support, all finish-async programs can be compiled either for work-sharing or work-stealing execution without requiring any changes to the source code [6, 9, 11].

The language approach results in improved productivity as programmers do not have to rewrite their code to take advantage of one or the other task execution strategy. This is possible because the language defines general purpose constructs and lets the compiler and runtime take care of implementation details. Although it would probably be possible for a library implementation to provide a more unified approach, the `j.u.c` API chose to clearly separate two sets of API, meant to do two different things, keeping them consistent and usable.

Another critical aspect for a task execution framework is to be able to handle tasks that block and ensure there is enough active workers for the program to continue to make progress. There are two key aspects to tackling this problem: how can a task let the runtime know it is going to block, and what can the runtime do about it.

Regarding work-sharing, the `j.u.c` allows programmers to adapt the size of the thread pool but does not provide any automated mechanisms for leveraging blocked task. HJ relies on language semantics to determine when a task is going to block (for example on a finish or a next operation) and generates code so that the runtime can take actions to ensure progress is possible; for example by creating new threads. Regarding work-stealing, both the `j.u.c` and HJ provide support for blocked tasks, but differ in the way they handle them.

The `j.u.c` provides a ManagedBlocker interface that defines a *block* method. Whenever a task needs to block, it notifies the runtime by providing an instance of ManagedBlocker. This indirection allows the runtime to decide which actions to take in face of this task being blocked before calling the *block* method. In this situation the fork-join implementation is going to try to ensure parallelism is maintained by eventually forking a new thread to compensate for the blocked task.

In HJ, when a task blocks on a *finish* and no further progress can be achieved by the current worker, the remaining code to be executed (continuation) and the current context (locals) are set aside. The worker tries to steal pending tasks from other workers to make the computation progress. The continuation may then be resumed or stolen by another worker. Although this kind of code could be written by programmers to interface with a task library implementation, it is application-specific and is in practice cumbersome to write and error-prone. This is a situation where a language implementation can really benefit the programmer as it can rely on compiler analysis and code transformation to output code that manages the creation and execution of continuations as well as saving and restoring execution contexts.

The HJ compiler goes one step further by implementing code generation for both help-first policies and work-first policies [9, 11]. Choosing one over the other is heavily dependent on the nature of the parallelism present in the code. The help-first policy is more efficient for flat and shallow loop parallelism whereas the work-first policy is more suitable for recursive parallelism. A third policy, named adaptive [11], allows the dynamic selection at runtime of the more efficient policy depending on the nature of the application.

4. Task Synchronization

Phasers are dynamic synchronization constructs [4]. Tasks can register and deregister dynamically on the phasers. Interestingly enough, both the `j.u.c` and HJ implement such feature. In fact, the Java concurrent utility Phaser class has also been inspired in part by the HJ phaser construct [16].

Viewed as synchronization objects, the HJ implementation is not far different from the `j.u.c` one. In both cases a phaser is represented as an object the programmer can interact with using its API. The main difference between the two implementations is that HJ includes language and compiler support (such as the `phased` clause in an `async` construct, and the `next` statement with an optional "single" computation) that makes it easier to use.

4.1 Managing Phasers

Conceptually, all tasks need to register on a phaser in order to use it and deregister when synchronization is no longer required. In HJ there is no need for a task to explicitly register or deregister on some phasers. When an *async* has an empty phased clause, the child task inherits all active phaser registrations from the parent task. When the *async* terminates, it automatically deregisters from all the phasers that it was registered on. If the task needs to be registered only with certain phasers, they can be specified in the phased clause.

In contrast, the `j.u.c` implementation requires tasks to explicitly invoke a method call to register on a particular Phaser object. Therefore it is the responsibility of the programmer to make sure registration and de-registration are done correctly, which is likely to become a source of bugs. For instance it is very easy to write a program that deadlocks if a task does not deregister.

This is an interesting example of how a language approach can reduce management chores. Figure 4 and Figure 5 show how `j.u.c` and HJ phasers can be used to act as a barrier. In the `j.u.c` version, the programmer has to call the register method for each task and also make sure the current thread deregisters itself by calling *arriveAndDeregister* (would deadlock otherwise) which allows the barrier to make progress. On the other hand the HJ implementation for the same code just needs to declare that the asynchronous execution of the loop body has to collaborate with phasers by specifying the phased keyword. HJ is able to hide the implementation details for several reasons. The runtime execution of an HJ program can be represented as a tree of tasks, the root node being a task executing the main of the program. Hence, when a phaser is created, it is always safe for the runtime to register it with the "current" executing task. Additionally, since the user has no handle over a task, i.e. a task cannot be scheduled for execution twice, it is always safe to deregister a phaser when a task terminates. Similarly to *arriveAndDeregister* when an activity reaches a finish it automatically deregisters from every phaser it is registered with.

In this scenario, the language approach can hide implementation details of a phaser because it enforces an execution model that the runtime can rely on.

4.2 Hierarchical Phasers

Both the `j.u.c` and HJ support organization of phasers into hierarchical trees [19]. This is a critical structure to achieve performance and scalability as the number of tasks involved in the synchronization grows.

The `j.u.c` implementation allows users to create trees of phasers by construction. The constructor of a Phaser can take a parent Phaser as argument. Although this approach is very flexible it can induce errors if the programmer makes mistakes such as introducing cycles or building a forest. The programmer must also ensure that each task will be registered on the correct phaser when building the tree.

The HJ implementation took a different approach by not allowing teh programmer to access the internal nodes of the tree. A hierarchical phaser [19] is initialized with two additional arguments, the number of tiers and the number of degrees. The number of tiers corresponds to the number of levels in the tree. The degree is the maximum number of tasks that can be registered at each leaf phaser.

```
public void run(final int N, final float[][] data) {
  final Phaser phaser = new Phaser() {
    protected boolean onAdvance(int phase, int nb) {
      mergeRows(data);
      return cond();
    }
  };
  phaser.register();
  for (int i = 0; i < N; ++i) {
    final int rowId = i;
    phaser.register();
    new Thread() {
    public void run() {
      do {
          processRow(data[rowId]);
          phaser.arriveAndAwaitAdvance();
        } while(!phaser.isTerminated());
      }
    }.start();
  }
  phaser.arriveAndDeregister();
}
```

Figure 4. Parallel `j.u.c` version of matrix chunking example using Phasers

When a new task is registered with the hierarchical phaser, it is going to automatically register with the next phaser slot available.

The hierarchical phaser API of HJ seeks a trade-off between usability and flexibility. For instance a HJ program that uses phasers can be converted to use hierarchical phasers simply by providing additional information to its constructor.

4.3 Single Statements

It is convenient to allow a single worker to execute a piece of code during a barrier synchronization when all tasks involved in a barrier reach it, but before proceeding to the next phase of a computation. For example, the code in question could involves checks for some properties or performing a reduction.

The approach taken by the `j.u.c` phaser is to allow programmers to extend the *onAdvance* method defined by the Phaser class whereas the HJ language defines the *single* construct as a block of code that post-fix a *next* statement, allowing the programmer to provide the code to be executed directly at the synchronization point.

Figure 4 and Figure 5 show how a barrier is expressed using `j.u.c` and HJ phasers. The strength of the HJ approach is that it keeps the code snippet at the exact point where the synchronization is performed. This approach increases the programmer productivity as the code in the *single* statement can access all variables accessible from the current lexical scope, which can be very convenient when checking whether the computation has reached some threshold value or when performing a reduction. An extra burden of the `j.u.c` approach is that programmers have to write and manage one Phaser implementation for each kind of barrier their program need. Additionally, usage of an anonymous inner class to override the *onAdvance* method can be error-prone as the programmer need to explicitly see the phaser declaration to determine if some extra code is going to be executed after calling the barrier.

HJ also features *accumulators* [20] that integrate with phasers to support parallel and dynamic reductions. Several tasks can contribute to a parallel reduction using the accumulator send operation. When tasks are synchronized, for instance in a single statement, the result of the accumulator can then be retrieved.

```
public void run(final int N, final float[][] data) {
  finish {
    phaser ph = new phaser();
    for (int rowId = 0; (rowId < N); ++rowId) {
      async phased {
       do {
          processRow(data[rowId]);
          next single { mergeRows(data); }
        } while (cond());
      }
  } }
}
```

Figure 5. Parallel (verbose) HJ version for matrix chunking example using phasers

```
public void run(final int N, final float[][] data) {
  forall(point [rowId] : [0:N-1]) {
do {
      processRow(data[rowId]);
      next single { mergeRows(data); }
    } while (cond());
  }
}
```

Figure 6. Forall parallel HJ version for matrix chunking example

5. Loop parallelism

Loop parallelism is especially important when it comes to handling large sets of data in parallel. A typical way to take advantage of such parallelism is to partition the data to be processed and create one computational task to process each chunk of data [21].

The HJ language defines a *forall* construct to perform parallel processing of arrays. The *forall* loop is similar to a regular for loop except that all iterations of the loop are potentially running asynchronously with respect to each other. The *forall* construct is implicitly wrapped in a *finish* by the compiler. Additionally, the *forall* construct also defines an implicit phaser allowing programmers to get more control over synchronization using the *next* construct. Figure 6 shows how the matrix chunking example from Figure 5 can be simplified using the *forall* construct.

The `j.u.c` implementation requires the programmer to be involved in extracting parallelism from the loop by creating a set of tasks, associating them with data and scheduling them for execution. Although this approach is very flexible and can be fine-tuned, it requires more investment from the programmer and a deeper knowledge of parallel programming than the *forall* approach.

Overall, the *forall* construct provides a very simple and practical approach to parallelism. Once some of the basic parallel concepts are understood, programmers can use the *foreach* construct that can take a phased clause as a parameter which gives a finer level of control to the programmer. These for loop constructs defined by the HJ language manage most of the parallelism for the programmer.

6. Improving Habanero-Java Usability

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, the Java language is a subset of the HJ language. The omission of these constructs in HJ makes porting of parallel Java program somewhat harder, but there are strong reasons for that. Usage of the synchronized keyword would mean allowing the use of wait-notify

primitives which are difficult to use correctly, and erroneous code that uses them can be difficult to detect and debug.

Since its creation from X10 v1.5, The HJ language has been used as an introductory language to parallelism at Rice University and we have been able to port various programs from other languages to HJ. In this section we present several language enhancements created to improve the usability of the language.

The async constructs follows the same restrictions Java anonymous inner classes enforce. In the body of an async, any variables referred to from the enclosing lexical scope have to be declared final and therefore cannot be modified. This makes the semantics of the programming model very clear, however, it makes porting code from Java to HJ more complicated. It also complicates modifying code from sequential to parallel and back. For instance, programmers need to constantly introduce final variables to make a value accessible to the body of an async, which makes programs very verbose. Judging by the feedback we received from programmers [3], this requirement was viewed as being unduly restrictive. We have removed this restriction from the HJ language. Now programmers may or may not declare an inherited variable as final. However, it is still a compilation error to try to assign a new value to an inherited variable. This change doesn't incur any runtime overhead as the final nature of a local variable is not propagated down to the generated bytecode. Therefore, there are no variable duplication from the programmer non-final variable to some compiler generated variables. The new generated code is identical to the previous one.

Although HJ generates Java bytecode, it is a language that used to have its own type system independent of Java. For example, when a programmer instantiated a new Integer object, the implementation package was `hj.lang` rather than java.lang. This was a major source of confusion for external collaborators and a major source of compilation problems to transform any existing Java code to HJ or use Java libraries and API from a newly written HJ program. In order to further ease porting from Java, the HJ type system has been relaxed to use primitive types classes from the *java.lang* package in place of those defined by HJ.

Regarding the integration of HJ in the Java ecosystem, it should be noted that since HJ produces 100% pure Java bytecode, programmers can take advantage of all the Java libraries by linking their programs to any existing class files and jars. Hence some of the `j.u.c` features can complement the primitives offered by the HJ Language. Atomic variables, concurrent map, deque and queue structures are particularly usefulto enable scalable data sharing among several tasks.

7. Conclusion

In this paper, we compared the usability of the `j.u.c` library approach with the Habanero-Java (HJ) language approach to task parallelism. We have focused our study on comparing two approaches that provide programmers with tasks creation and scheduling as well as loop parallelism and high-level synchronization constructs. We have shown a general purpose library-based approach to task parallelism can be verbose and requires a lot of meticulous micro management whereas a language-based approach provides enough flexibility to hide complexity as well as reduce the possibilities of flawed designs and errors in programs. The HJ language offers simple and safe high-level parallel constructs for usability and productivity. It provides programmers with a simple task-oriented programming model, deadlock-free synchronization constructs and parallel loop processing capabilities, which makes it appealing to main-stream users.

Our conclusion is that a language approach such as HJ is more suitable for mainstream users who lack expertise in parallelism, whereas a library approach such as the `j.u.c` is appropriate for more advanced users.

References

[1] G. Blelloch. NESL: A Nested Data-Parallel Language. Technical Report CMU-CS-92-103, Carnegie Mellon University, January 1992.

[2] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[3] K. Ebcioglu et al. An experiment in measuring the productivity of three parallel programming languages. In *Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, 2006.

[4] 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.

[5] K. Yelick et al. Productivity and performance using partitioned global address space languages. In *Proceedings of the international workshop on Parallel symbolic computation*, pages 24–32, New York, NY, USA, 2007.

[6] M. Frigo et al. The implementation of the Cilk-5 multithreaded language. In *PLDI'98*, pages 212–223, June 1998.

[7] 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.

[8] R. D. Blumofe et al. CILK: An efficient multithreaded runtime system. *PPoPP'95*, pages 207–216, July 1995.

[9] Y. Guo et al. Work-first and help-first scheduling policies for async-finish task parallelism. In *IPDPS'09*, April2009.

[10] B. Goetz et al. *Java Concurrency In Practice*. Addison-Wesley, 2007.

[11] Y. Guo et al. SLAW: a scalable locality-aware adaptive work-stealing scheduler. In *IPDPS '10*, April 2010.

[12] R. Gupta. The fuzzy barrier: a mechanism for high speed synchronization of processors. In *ASPLOS '89*, pages 54–63, New York, USA, 1989.

[13] Habanero. Habanero Java. http://habanero.rice.edu/hj, Dec 2009.

[14] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[15] D. Lea. A Java fork/join framework, 2000.

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

[17] R. Milne and C. Strachey. *A Theory of Programming Language Semantics*. Halstead Press, New York, 1976.

[18] V. Sarkar. Synchronization Using Counting Semaphores. In *Proceedings of the International Conference on Supercomputing*, pages 627–637, July 1988.

[19] J. Shirako and V. 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.

[20] J. Shirako et al. Phaser accumulators: A new reduction construct for dynamic parallelism. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[21] J. Shirako, J. Zhao, V. K. Nandivada, and V. 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.

[22] X10. Release 1.5 of X10 system dated 2007-06-29. http://sourceforge.net/project/showfiles.php?group_id=181722 &package_id=210532&release_id=519811, 2007.

[23] Y. Yan et al. 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.