Habanero-Java extensions for Scientific Computing

Vincent Cavé
Rice University
vc8@rice.edu

Jisheng Zhao
Rice University
jz10@rice.edu

Zoran Budimlić
Rice University
zoran@rice.edu

Vivek Sarkar
Rice University
vsarkar@rice.edu

James Gunning
Australian Commonwealth Scientific and Industrial Research Organisation
James.Gunning@csiro.au

Michael Glinsky
Australian Commonwealth Scientific and Industrial Research Organisation
Michael.Glinsky@csiro.au

ABSTRACT

Mainstream object-oriented languages such as Java and C#, through the use of object-oriented abstractions and managed runtimes (virtual machines), have significantly improved productivity and portability in multiple application domains. However, despite many attempts in the past, the effect of these improvements on high-performance numeric computations has been limited. In this report, we describe the results and lessons learned from a one-year joint study between researchers in an industrial company (BHP Billiton) and an academic institution (Rice University) to port Dipole1D, an open source Fortran 90 application for 1D forward modeling of an arbitrarily located and oriented electric dipole transmitter, to Java with a goal of gaining efficient sequential and multicore implementations. Our primary conclusions from this study are as follows: 1) a standard library-based implementation of Fortran 90 primitives in Java (especially complex arithmetic and complex variables) results in unacceptably large performance overheads, 2) the Java bytecodes generated from this translation include large methods for which current JIT compilers generate surprisingly inefficient code, 3) hand splitting of the large methods removes much of this inefficiency, and 4) after building the sequential extended-Java version of the Dipole1D application, the path to multicore execution is greatly simplified. The resulting Habanero-Java version of Dipole1D is in the process of being made available publicly, and sheds light on missed opportunities from the past, such as JSR 84 (Floating Point Extensions) which has been withdrawn in 2002.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]

General Terms

Languages

SPLASH '10 October, Reno, NV.

1. HABANERO JAVA

The Habanero Java (HJ) language [3] developed at Rice University is derived from X10 v1.5 [9], which in turn was based on the Java v1.4 language specification. HJ has made multiple modifications to X10 v1.5, including an extension of X10's *clocks* called *phasers*, relaxation of X10's *atomic* construct to *isolated* and others, discussed later in this paper. 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 is included below.

async: Async is the HJ construct for creating or forking a new asynchronous task (as in X10 v1.5). 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 HJ 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, as in X10 v1.5). Operationally, each instruction executed in an HJ task has an unique *Immediately Enclosing Finish* (IEF) statement instance [8].

Besides termination detection, the `finish` statement plays an important role with regard to exception semantics. As in X10, 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* [1] formed from the collection of all exceptions thrown by all abruptly-terminating tasks in the IEF. (In contrast, in the Java model an exception is simply propagated from a thread to the top-level console.)

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 future task completes and the result of `Expr` become 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 [5], 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 [8] 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. In addition, a *next* statement for phasers can optionally include a *single* statement which is guaranteed to be executed exactly once during a phase transition [10]. 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 [2, 6], counting semaphores [7], and X10's clocks [1].

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 [10, 8]. 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[1]. 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 `async`'s 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 [2]. 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.

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

[2]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*⟩".

2. COMPLEX NUMBERS

2.1 Existing Options

Java programmers have two limited options when it comes to using complex numbers in their programs: implementing complex as a pair of primitive real numbers or implementing complex numbers as Java objects.

Since complex arithmetic is quite verbose, the solution based on pairs of primitive real numbers makes programs quickly difficult to read and understand. Implementing complex arithmetic using functions does not help either since functions cannot return pairs of data. In this situation the object-oriented approach sounds more appealing. However, programmers now need to write method calls in place of operators to perform arithmetic operation, which makes the code verbose and difficult to read. Regarding performance, the major drawback of the object-oriented approach is the need to instantiate new complex objects, an expensive operation compared to the cost of performing simple arithmetic operations. Therefore, to be as efficient as possible, several different method implementations must be offered for each arithmetic operation, some performing in-place computation and some returning a new complex object. Although the API can circumvent object creation in various ways by passing around results and parameters to methods, the syntax becomes more complicated and error prone.

2.2 Primitive Type-based Complex Numbers

We have implemented complex numbers in the Habanero-Java language by introducing two new types: *complex32* (32 bits single-precision) and *complex64* (64 bits double precision).

These types can be used as any other regular primitive types. The compiler provides support for declaration, assignment, arithmetic operations, conversions, promotions and arrays of complex. Declarations can be constants, local variables and field of objects. A method can take complex as argument as well as return value.

A complex number is declared by specifying a pair representing the real and the imaginary part. Accessing the real and the imaginary part of a complex is done through the *real* and *imag* functions that take a complex number as an argument. Figure 1 shows an example of the syntax for declaring a complex variable, a complex array and print a complex value. Complex literals and variables are transformed to a String for printing and whenever a conversion to the String type is required.

The syntax for arithmetic operations involving complex numbers is identical to operations involving regular primitive types. The compiler provides support for addition, subtraction, multiplication and division as well as compound assignment for these operations. Additionally, Habanero-Java provides a library that implements mathematical operations to compute the exponential, the square root and the absolute value of a complex number. Figure 2 shows some examples of using arithmetic operators and examples of implicit type promotions in presence of complex numbers.

```
complex32 cx = (1.0f, 2.0f);
float r = real(cx);
float i = imag(cx);

complex64 [] a = new complex64[size];
for (int k=0; k < a.length; k++) {
  a[k] = a[k] * cx;
  System.out.println(a[k]);
}
```

Figure 1: Examples of complex variables and arrays declarations and usages.

```
complex32 cx = (1.0f, 2.0f);
complex32 add = cx + (1.0f, 2.0f);
complex32 sub = cx - (2.0f, 2.0f);
complex32 mul = cx * cx;
complex32 div = cx / (2.0f, 2.0f);

complex64 cy = (1.0, 2.0);
cy += 1;
cy -= (2.0, 2.0);
cy *= cx;
cy /= (2.0, 2.0);

cx = cabs((2.0,2.0));
cx = csqrt(cy);
cx = cexp(cy);
```

Figure 2: Examples of arithmetic operation and implicit type promotions involving complex numbers.

Providing programmers with a support for complex number as a primitive type offers better productivity compared to custom implementations using pairs of reals or objects. It also makes porting from legacy code supporting complex (such as Fortran 90) to Habanero-Java much more straightforward as the syntax used is similar.

2.3 Implementation

The compiler translates all complex numbers to pairs of Java primitive floating-point numbers, keeping the generated bytecode in pure Java. Code generation of simple operations such as declarations and access to real or imaginary parts is straightforward. For instance, a complex variable declaration is simply split into two variable declarations of the corresponding primitive type, *float* for *complex32* and *double* for *complex64*. The compiler takes care of transforming complex constants, variables as well as fields of objects to pairs of real numbers. Getting the real or the imaginary part of a complex is rewritten as a read of the corresponding variable. Arithmetic operations are slightly more complicated since the compiler needs to generate proper arithmetic expressions and use the appropriate real and imaginary variables for each complex involved in the operation.

The compiler also supports complex arguments and return values. If the method signature contains a complex as a parameter it is rewritten as a pair of parameters in order to represent both the real and the imaginary parts of the complex. If the function returns a complex, the default strategy is to inline the call. The compiler also supports another code

```
float cx_r = 1.0f;
float cx_i = 2.0f;
float r = cx_r;
float i = cx_i;

double [] a = new double[size*2];
for (int k=0; k < (a.length/2); k++) {
  a[k] = (a[k]*cx_r) - (a[k+1]*cx_i);
  a[k+1] = (a[k+1]*cx_r) + (a[k]*cx_i);
  System.out.println("(" + a[k] +
    "," + a[k+1] + ")");
}
```

Figure 3: Code generated by the Habanero-Java compiler from the example shown in Figure 1.

generation strategy where the primitive complex to be returned is boxed into an object. Note that the compiler takes care of renaming methods that had their signature changed to avoid any collision with methods already defined.

Figure 3 shows how the example shown in Figure 1 looks like after the compiler transformed complex declaration and uses. The first complex32 declaration is split into two float variables and calls to the real and imag accessors are transformed to directly refer to those variables. Arrays length is doubled so as to be able to store pairs of real and imaginary. In order to keep loop traversal consistent the compiler then needs to divide by two the value returned when getting the length of an array from its length field. The loop body shows how the multiplication operation which is simply written using the star operator is expanded as the complex multiplication by getting real and imag parts from both a complex array element and a complex variable. The last statement of the loop shows how a complex array element is transformed to a String by reading both its real and imaginary part. The transformation of complex numbers to pairs of variable multiply by two the number of variables declared in a program. However, further compiler passes can greatly optimize this code by mean of classic compiler optimization such as constant propagation, code motion and load elimination.

2.4 Performance

We have developed a set of micro-benchmarks in order to evaluate the performance of our implementation of primitive complex. The micro-benchmarks evaluate the time it takes to perform arithmetic operations on complex numbers, these include simple arithmetic operations such as add, subtract, multiply, divide and some more advanced arithmetic operations such as square root, exponential and absolute value. In the context of the Dipole1D application we particularly focus on performing complex number operations on arrays of complex numbers declared as fields of objects. Every micro-benchmark applies an operator and stores the result to an array of complex declared as a field of an object. The micro-benchmarks are divided in four sets, each of them apply addition, subtraction, multiplication and division. The first set applies operators to complex arrays using a compound statement, the second set to a complex from an array and a real variable and the third set to complex numbers taken from two different arrays. A last set of micro-benchmarks evaluates the advanced arithmetic operations mentioned earlier.

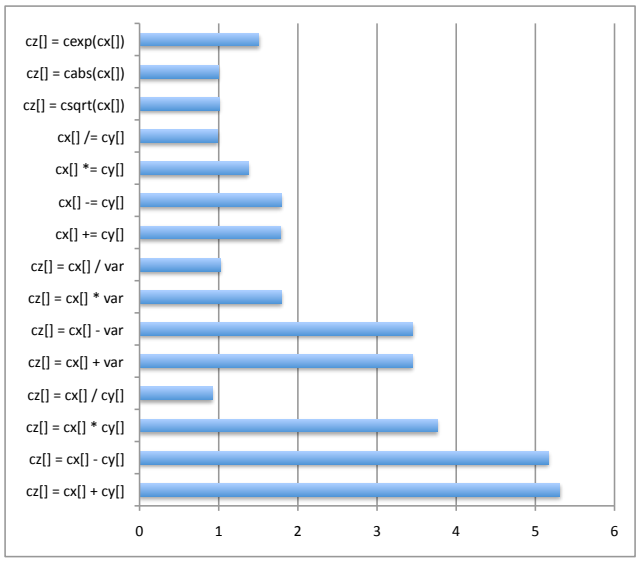

Figure 4: Speed-up using the Habanero-Java primitive-based complex implementation compared to the Java object-oriented implementation for various arithmetic operations.

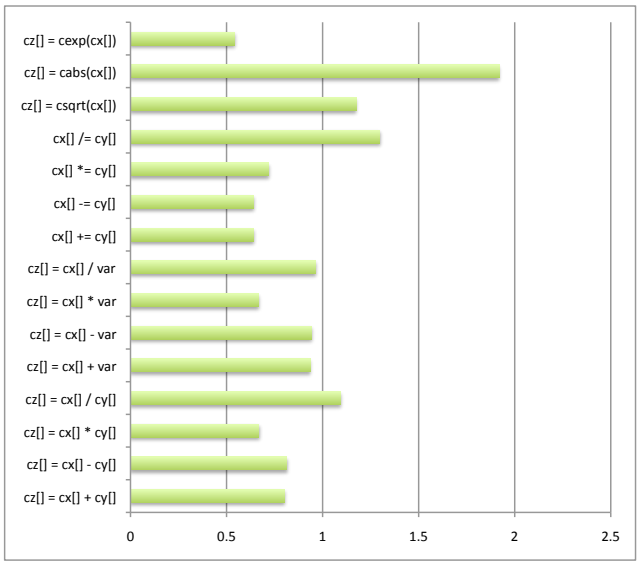

Figure 5: Speed-up using the Habanero-Java primitive-based complex implementation compared to Fortran 90 for various arithmetic operations.

We have developed three versions of the micro-benchmarks for Fortran 90, Java and Habanero-Java. The Java version uses complex implemented in an object-oriented fashion whereas the Habanero-Java version uses complex as a primitive type. The Fortran 90 version has been compiled with the O3 optimization flag, Java and HJ implementations are ran with the 1.6 SUN Java virtual machine using the -server option.

Figure 4 shows micro-benchmarks speed-up gained when using the primitive-based complex number implementation as opposed to the object-based one.

Not surprisingly we can observe the primitive-based complex implementation matches or outperforms the object one in all scenarios. The speed-up ranges from similar performance to a factor of five. The largest difference is seen when the object-based implementation has to instantiate new complex objects. When the object-based implementation does in-place updates, the advantage of the primitive-based implementation is still up to a factor of two, due to the elimination of costs introduced by method calls and field accesses.

Figure 5 compares our primitive-based complex implementation to native Fortran 90. The Habanero-Java implementation is roughly twenty percent slower than the Fortran 90 one.

3. USE CASE DIPOLE 1D

Dipole1D [4] is the kernel of an Inversion Program for Generating Smooth 1D Models from Controlled-Source Electromagnetic and Magnetotelluric Data (OCCAM1DCSEM). It computes the frequency–domain electromagnetic fields produced by a point dipole source embedded in a layered medium, together with the derivative (or sensitivity/Jacobian) matrix of those fields with respect to the resistivity of the layers in the medium. This problem is essential to new remote-sensing methods for oil/gas exploration using towed deep-marine dipole sources and seabed receivers, usually called "Controlled Source Electromagnetics (CSEM)", or "Sebed logging (SBL)".

We have implemented three versions of the Dipole1D application using Habanero-Java: a direct port from Fortran 90 using complex numbers as objects, a direct port using complex numbers as primitive types, and a hand-optimized version of the primitive-based implementation.

3.1 Experiments

Our experiments are based on running the Dipole1D kernel using six different use-case inputs that exercise various parts of the code and varies the number of transmitters, frequencies, layers and receivers. We have ran these benchmarks through the three implementations of dipole 1D and compared results with the original Fortran 90 implementation. All experiments measure the duration of Dipole1D core computation and do not take into account neither the application setup phase nor the result saving phase. The Fortran 90 version of Dipole1D has been compiled with GFortran 4.1.2 and the O3 optimization flag set. The Habanero-Java version of Dipole1D uses Habanero-Java version 1.1 running with Java 1.6.0_14.

Our starting point is to compare the Fortran 90 version of Dipole1D with the Habanero-Java implementation that relies on complex numbers implemented as objects. On average, benchmarks show that this implementation of Dipole1D is forty-nine times slower than its Fortran 90 counterpart.

Micro-benchmarks showed that complex numbers implemented as a primitive type achieve very good performance compared to an object-based implementation. This result is repeated for the Dipole1D application, which heavily relies on complex number arithmetic. Going from an object-oriented complex implementation of Dipole1D to a primitive-based increases performance on average by a factor of eight.

However, the primitive-based Dipole1D implementation is still a factor of ten slower than native Fortran 90.

3.2 Optimizations

Micro-benchmarks presented in Figure 5 have indicated that complex arithmetic in Habanero-Java should be within a factor of two to a native Fortran 90 implementation, which is clearly not the case for the direct port of Dipole1D. This section presents an analysis of these performance problems and solutions to them.

Method Size.

Profiling the application showed that most of the time is spent in large methods that perform intensive arithmetic computations involving complex numbers. We came to the conclusion that the just-in-time compiler is not designed to optimize large functions, being optimized to deal frequently with a large number of relatively small methods rather than few large monolithic methods.

For instance the Java HotSpot Just-In-Time (JIT) compiler has a heuristic that disables method inlining if the caller method is too large, regardless of how short the callee method is, or how important would its inlining be to overall performance. This affects the Dipole1D application, since its methods are very large, preventing the JIT compiler from inlining even the simplest and shortest method calls, including the manipulation of complex numbers.

The solution was to refactor the Dipole1D code by decomposing all large methods into smaller ones containing between twenty and forty lines of code. This transformation can be done automatically by the compiler.

Inlining versus boxed return value.

Inlining is a typical optimization for object-oriented languages, as it can reduce the overhead for saving/restoring the calling context. Since our compiler implementation forces inlining of every method that returns complex numbers, inlining can become detrimental to performance as it can greatly increase the size of some methods. For such methods we use boxing to return complex value as the JIT may then freely decide if it's worth inlining the method call or not. In this case one must be careful when to declare and how to reuse these boxed objects to amortize the cost of their creation.

Usage of the final keyword.

Using the final keyword can greatly help the JIT compiler in many situations including field accesses, conditional branches and loop boundaries. The Dipole1D code uses a lot of branch conditions to tests whether current iteration of the computation is of a certain kind (linversion for example). We have declared as final the majority of these boolean variables to allow the JIT to eliminate most of the branch conditions. We have also declared as final arrays that are part of an object, to allow the compiler optimize array accesses since it then knows the reference to the array is not going to change. Another optimization was to declare as final some integer fields that are extensively used as upper bounds in loops. This helps the JIT in array bounds check elimination.

3.3 Results

Experiments show that we gain a factor of four to thirteen compared to the non-optimized Habanero-Java version.

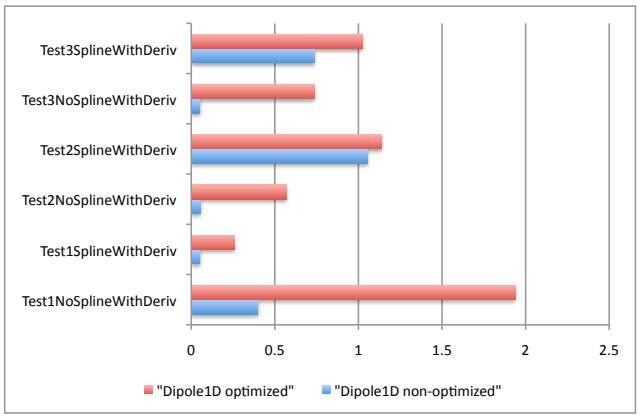

Figure 6: Speed-up of Dipole1D non-optimized and optimized Habanero-Java implementations compared to the Fortran 90 implementation.

If we compare these results to native Fortran 90 implementation they show a much more competitive performance. Three tests show similar or better performance than Fortran 90, two other tests show less than 50% overhead, and only one case (Test1SplineWithDeriv) has an overhead factor greater than two. The culprit is inefficient standard Java implementation of the the StrictMath.log10 function on which the Test1SplineWithDeriv spends two thirds of its execution time.

4. PARALLELIZATION OF DIPOLE1D

Since Dipole1D is a basic routine for similar (but much more computationally demanding) 2D and 3D problems, its parallelization is essential for the performance of those higher-dimensionality problems. Dipole1D code offers several possibilities for parallelization. In the computation core there is an embarrassingly parallel outer loop over transmitters, as well as a loop over frequencies, which can also be parallelized.

4.1 Using HJ Constructs in Dipole1D

Although Dipole1D exhibits embarrassingly parallel loops in the computation core, a preliminary refactoring of the Dipole1D code is necessary. Since there is no concurrency in the sequential version, data structures are all initialized once at the beginning of the application and reused by overwriting them from one iteration to another. This can be done because there are no data dependencies between iterations. However, exploiting the parallelism present in the application introduces race-conditions on those data structures. Hence, the first step of the parallelization process is to refactor the code so that each loop iteration can be executed in parallel without interfering with others. Worthy notice that in this situation there is an inevitable trade-off between parallelism and memory requirements.

Regarding the parallelization itself, one straightforward way of taking advantage of an embarrassingly parallel loop is to execute in parallel every iteration of the loop independently from each other. This can be done very easily in HJ, by wrapping the body of the loop in an async, which causes

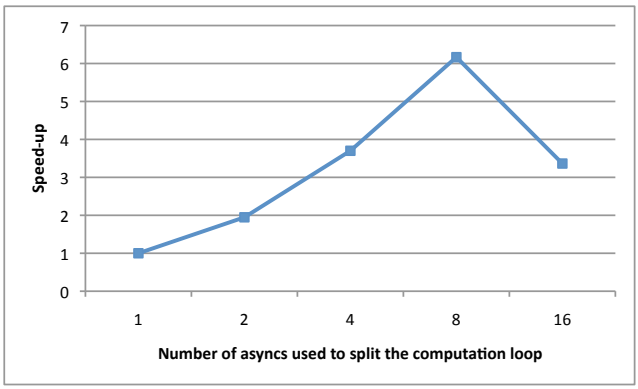

Figure 7: Dipole1D parallel version speed-up with respect to sequential execution when varying the number of asyncs to split the computation loop.

each iteration of the loop to be executed concurrently to others. However doing so is only effective if the amount of work carried by each loop iteration is large enough to compensate for the overhead of creating and managing asyncs. One classical approach to leverage this problem is to divide the iteration space of the loop into chunks. Such technique allows to create coarser grain tasks and also allows memory reuse across iterations within a same chunk. A finish block must enclose the loop to ensure iterations executed asynchronously have ran to completion before proceeding further.

4.2 Parallel Dipole1D Experiments

This section presents performance results of the parallel Dipole1D implementation on a 16-core Intel Xeon SMP using Java version 1.6. In order to be worth using, the parallel version of Dipole1D must process an input configuration that exhibits enough parallelism. To assess performance of the parallel version of Dipole1D we use a new use-case with an increased number of transmitters and receivers. The number of transmitters allows to control the number of iterations the outermost loop of the core computation. This is the above mentioned embarrassingly parallel loop that we have parallelized and divided into chunks. The number of receivers influences how coarse each iteration within a chunk is.

Figure 7 shows speed-up achieved when varying the number of asyncs relative to the sequential version. We can see the speed-up curve stays close to the ideal speed-up when the loop is divided in two to four asyncs, achieving respectively 1.9x and 3.7x speed-up compared to the sequential execution. When eight asyncs are used the speed-up still increases (6.1x) but notably goes below the ideal speed-up. When more than eight asyncs are used, speed-up starts to decrease.

Since the test-bed machine has 16 cores, we can conclude the observed performance decrease when going above eight asyncs is not caused by computing resource starvation, but rather by the increase in memory pressure as every async needs to access arrays of complex stored in the heap concurrently to perform computations. Therefore even though the Dipole1D offers plenty of parallelism to be exploited, the scaling of a parallel implementation is bounded by the memory bandwidth.

5. CONCLUSIONS

We have developed an extension for the Habanero-Java language to support complex numbers as a primitive type. From a programmability point of view, it allows users to perform basic arithmetic operations with complex numbers similar to using other primitive types. From a performance point of view, experiments showed that arithmetic operations are performed two to eight times faster than using an object-based solution, which makes them comparable in performance to optimized Fortran 90. We have implemented two versions of the Dipole1D application using Habanero-Java with complex numbers as a primitive type. The first version was shown to be very inefficient, between four and thirteen times slower than the Fortran 90 implementation. Experiments have shown that the method size is a major source of overhead, since the JIT compiler is having a hard time optimizing them. We have implemented a hand-optimized version of Dipole1D, which had its large methods split into smaller ones. This version shows a radical performance improvement and has an execution overhead within a factor of two compared to the optimized Fortran 90 implementation. Finally, we presented results of a parallelized version of the Dipole1D application, which achieved speed-up by exploiting the embarrassingly parallel nature of the main computation loop using finish and async constructs.

We believe these language extensions are a first step to widen acceptance of modern object-oriented languages into the scientific application community by providing developers with features that improve productivity while retaining acceptable performance compared to their original implementations.

6. REFERENCES

[1] 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.

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

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

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

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

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

[7] V. Sarkar. Synchronization Using Counting Semaphores. In *ICS'88*, pages 627–637, July 1988.

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

[9] Release 1.5 of X10 system dated 2007-06-29, 2007.

[10] K. Yelick et al. Productivity and performance using partitioned global address space languages. In *PASCO'07*, pages 24–32, New York, NY, USA, 2007. ACM.