Java for Scientific Computation: Prospects and Problems

Henk J. Sips and Kees van Reeuwijk

Delft University of Technology, the Netherlands
{sips,vanReeuwijk}@its.tudelft.nl

Abstract. Fortran is still a very dominant language for scientific computations. However it lacks modern language features like strong typing, object orientation, and other design features of modern programming languages. Therefore, among scientists there is an increasing interest in object oriented languages like Java. In this paper, we will discuss a number of prospects and problems in Java for scientific computation.

1 Introduction

Thusfar, Fortran has been the dominant language for scientific computation. The language has been modernized several times, but backward compatibility has made it necessary for modern constructs to be omitted. Nevertheless, scientists and engineers would like to use features that are only available in modern languages such as C, C++, and Java. Although it is tempting to abandon Fortran for a more modern language, new languages must successfully deal with a number of features that have proved to be essential for scientific computation. These features include multi-dimensional arrays, complex numbers, and, in later versions, array expressions (Fortran95, HPF, OpenMP). Any language that is to replace Fortran will at least have to efficiently support the above mentioned features. In addition, experience with scientific programs in Fortran has shown that support for structured parallel programming and for specialized arrays (block, sparse, symmetric, etc.) is also desirable.

In the paper, we describe a number of approaches to make Java suitable for scientific computation as well as a number of problems that still have to be solved. The approaches vary in their "intrusiveness" with respect to the current Java language definition.

2 Array Support

2.1 Multi-dimensional arrays as basic data structure

Language support for handling arrays is crucial to any language for scientific computation. In many languages, including Java, it is assumed that it is sufficient to provide one-dimensional arrays as a basic data structure. Multi-dimensional arrays can then be represented as arrays of arrays, also called the *nested array* representation. However, the Java array representation has some drawbacks:

- Memory layout for nested arrays is determined by the memory allocator applied. As a result, the rows of an array may be scattered throughout memory. This in turn deteriorates performance through poor cache behavior.
- For nested arrays, a compiler must take into account array aliasing (two array rows are the same within an array, or even between arrays) and ragged arrays (array rows have different lengths). This complicates code optimization.
- Garbage collection overhead for nested arrays is larger, since all rows of the array are administrated independently.
- Nested arrays are difficult to optimize in data-parallel programming. Extensive analysis is required to generate efficient communication code.

Therefore, the one-dimensional array support that Java currently offers is considered to be insufficient to support large scale scientific computations and many researchers have proposed improvements on the array support in Java. We will discuss a number of these approaches in the order of intrusiveness of the Java language.

The most elegant solution is to add true multi-dimensional data structures to the Java core language. Two such solutions are proposed in the Spar/Java project [14] and the Titanium project [16]. For example, a two-dimensional array in Spar/Java is declared and used as follows:

```
int a[*,*] = new int[10,10];
for( int i=0; i<a.GetSize(0); i++ )
   for( int j=0; j<a.GetSize(1); j++ )
     a[i,j] = i+j;
```

In general, arrays are indexed by a list of expressions instead of a single expression. Similarly, in an array creation expression a list of sizes is given instead of a single size. These features are straightforward generalizations of existing Java language constructs.

The `GetSize(int)` shown in the example is a method that returns the size of the array in the given dimension. This is an implicitly defined method on the array, similar to the `clone()` method that is defined on arrays in standard Java.

Titanium [16] resembles Spar/Java in the sense that it also provides a set of language extensions to develop Java into a language for scientific computations. It provides support for multi-dimensional arrays similar to Spar, although the concrete syntax is different. For example, the following Titanium code declares a two-dimensional array:

```
Point<2> l = [1,1];
Point<2> u = [10,20];
RectDomain<2> r = [l:u];
double [2d] A = new double[r];
```

As a simple illustration of the costs of multi-dimensional versus nested arrays, consider the following loop in Spar/Java:

```
for( int i=0; i<M; i++ )
   for( int j=0; j<M; j++ )
     A[i][j] = B[j][i];
```

which copies the transpose of array `B` into array `A`. The variant using multi-dimensional arrays simply replaces the statement in the inner loop by

```
A[i,j] = B[j,i];
```

We measured the execution times of these programs for `int` arrays of 2197×2197 elements, for 40 runs of this loop. We also measured the execution times of the analogous programs working on three-dimensional arrays of $169 \times 169 \times 169$ elements, again for 40 runs of the loop. We compiled both versions of the program with our Spar/Java compiler, called *Timber*, and measured the execution time of the resulting programs. For comparison we also measured the execution time of the Java variants of these programs using the Java HotSpot 1.3.0 Client VM. The programs were executed on a 466 MHz Celeron with 256 MB of memory running Linux. The shown execution times are in seconds.

Array type - compiler	2D array	3D array
Nested - Timber	64.9	123.7
Nested - Hotspot	60.6	84.6
Multidim. - Timber	6.3	7.2

The significantly larger execution times of the programs with nested arrays is caused by several factors. An indication of the overhead of one factor, bounds checking, can be found by disabling the generation of bounds checking code in the Timber compiler, and measuring the execution times again. In that case the results are (bounds checking of the HotSpot compiler cannot be disabled):

Array type - compiler	2D array	3D array
Nested - Timber	44.4	70.5
Multidim. - Timber	6.3	7.1

As these results indicate, for the multi-dimensional array representation the overhead of bounds checking is limited. For the nested array representation the overhead of bounds checking is larger, but there are other significant factors that contribute to the larger execution times, such as null pointer checks, memory layout issues, and more complicated array index calculations.

2.2 Multi-dimensional arrays as libraries

An important disadvantage of the support for multi-dimensional arrays described in the previous section is that an extension of the Java language is necessary. This is considered undesirable by many people. For this reason, a number of people have proposed to provide support for multi-dimensional arrays in the form of library functions. Basically there two approaches to libraries of this kind: as compiler known functions or as an independent library. Examples of both approaches are the Ninja and JAMA libraries, respectively.

In the Ninja project [1, 4] a compiler has been developed for pure Java. To provide support for array operations, a set of 'special' classes is defined that

represent multi-dimensional arrays and complex numbers. These classes can be handled by all standard Java compilers, but the Ninja compiler recognizes these special classes, and generates efficient code for them. However, since their notation for access to multi-dimensional arrays is quite awkward, they are advocating language extensions for at least multi-dimensional array access. Based on this work, a proposal has been made through the Java Community Process to add multi-dimensional arrays to Java [10].

A number of Java packages for linear algebra have been proposed, see for example JAMA [3]. These packages often also introduce multi-dimensional arrays, but usually only in a restricted form.

All the above proposals have as a drawback that they impose restrictions on the element type and rank of the supported arrays. Moreover, the notation of array types and array access is not very elegant. For example, in Spar/Java the main statement in a matrix multiplication is as follows:

```
c[i,j] += a[i,k]*b[k,j];
```

while using the Java Community proposal for multi-dimensional arrays, all array references have to go through `get()` and `set()` method invocations, like

```
c.set(i,j,c.get(i,j)+a.get(i,k)*b.get(k,j));
```

3 Specialized array representations

Many languages only support rectangular arrays as primitive data types. However, in real scientific applications frequently specialized array representations occur, such as block, symmetric, and sparse arrays. Because there is little unification in their representation, it makes no sense to make them a primitive data type. However, there are a number of possible language extensions to Java that would greatly contribute to their support in application programs. These extensions are: (i) Parameterized classes, (ii) Overloading of the subscript operator, (iii) Tuples and vector tuples, and (iv) Method inlining.

Parameterized classes allow generic implementations of specialized arrays. In particular, the implementations can be generic in the element type and the number of dimensions. Overloading of the subscript operator greatly improves the readability of the manipulation of specialized arrays. To be able to express manipulations on arrays with different numbers of dimensions generically, it is necessary to introduce vector tuples. Finally, to ensure that the use of specialized arrays is as efficient as the use of standard arrays, it is necessary that some methods are always inlined, in particular methods that access array elements.

Parameterized classes. In its simplest form support for specialized arrays can be provided by simply designing a standard Java class. An example of such an approach is the class `java.util.Vector`. However, in this approach it is not possible to abstract from parameters such as the element type, rank, or from 'tuning' parameters such as block sizes or allocation increments.

Support for some form of class parameterization is therefore highly desirable. A number of proposals have been made to add class parameterization to Java [8, 12, 15]. Also, there is a proposal in the Java Community process [7] to add proposal [8] to standard Java.

To avoid having to extend the existing JVM definition–which would render all existing JVM implementations obsolete–most proposals only allow reference classes as parameters. With this restriction, parameterized classes can be rewritten as operations on an unrestricted version of the class, and a number of casts and assertions. However, this makes these proposals less suited to support specialized arrays, since for this case parameterization with primitive types (e.g. for element types of the specialized arrays) and with numeric values (e.g. for numbers of dimensions) is required.

The Spar/Java language provides a different class parameterization mechanism, based on template instantiation. Using this approach, very efficient class instantiation is possible. Moreover, arbitrary type parameters and value parameters can be supported. For example, Spar/Java provides a typed vector in `spar.util.Vector`, which is implemented as follows (simplified):

```
final class Vector(| type t |)  {
     protected t elementData[] = null;
     public Vector(){}
     public Vector( int initCap ){
         ensureCapacity( initCap ); }
     // Etc.
}
```

The sequence `(| type t |)` is the list of parameters of the class. The list of parameters can be of arbitrary length. Parameters can be of type `type`, and of primitive types. Actual parameters of a class must be types, or evaluate to compile-time constants. For every different list of actual parameters a class instance is created with the actual parameters substituted for the formal parameters. Class `spar.util.Vector` can be used as follows:

```
// Create a new instance of an int vector with initial
// capacity 20.
Vector(| type int |) v = new Vector(| type int |)( 20 );
```

Vector tuples. To allow generic implementations of specialized arrays, it is necessary to allow a list of subscript expressions to be treated as a single entity, regardless of its length (and hence regardless of the rank of the subscripted array). This is easily possible by considering subscript lists as *tuples*. Thus, an ordinary array index expression such as `a[1,2]` is considered as the application of an implicit index operator on an array (`a`), and a tuple (`[1,2]`).

For example, Spar/Java generalizes this concept by allowing tuples as 'first class citizens' that can be constructed, assigned, passed as parameters, and examined, independent of array contexts. Spar/Java also provides an explicit array subscript operator '`@`'. The following code shows tuples and the `@` operator in use:

```
[int^2] v = [1,2];          // Declare, init. tuple
int a[*,*] = new int[4,4]; // Declare, init. array
a@v = 3;                    // Assign to a[1,2]
```

Subscript operator overloading. If specialized arrays are constructed using standard Java classes, then access to elements of these arrays must be done using explicit methods. For example, to swap elements 0 and 1 of a `java.util.Vector` instance `v` requires the following code:

```
Object h = v.elementAt(0);
v.setElementAt(v.elementAt(1),0);
v.setElementAt(h,1);
```

Such a notation is acceptable for occasional use, but is not very convenient for frequent use. For this reason, Spar/Java supports overloading of the index operator. If an index operator is used on an expression of a class type, this expression is translated to an invocation to a method `getElement` or `setElement`, depending on the context. For example, assuming '`v`' is a class instance, the statement `v[0] = v[1]` is translated to `v.setElement([0],v.getElement([1]))`. Obviously, the class must implement `getElement` and `setElement` for this convention to work.

At first sight it seems more obvious to choose an existing pair of functions instead of `setElement` and `getElement`. Unfortunately, the standard Java library is not consistent on this point: `java.util.Vector` uses `setElementAt` and `elementAt`, `java.util.Hashtable` uses `get` and `put`, etc. Moreover, for reasons of generality the methods `getElement` and `setElement` take a vector tuple as parameter, which makes them incompatible with any Java method anyway.

4 Complex numbers

Complex numbers are frequently used in scientific programs. Hence, it is very desirable to have a compact notation and efficient support for them. Complex numbers can be easily constructed by using a new class that represents complex numbers and the manipulations on them. This approach has been proposed, among others, by the Java Grande Forum [2] and for use with the IBM Ninja compiler [1, 11]. However, this approach has some drawbacks: complex numbers are stored in allocated memory, manipulations on complex numbers must still be expressed as method invocations, and the complex class is still a reference type, which means that values can be aliased. To a certain extent these problems can be reduced by a smart compiler, especially if it is able to recognize the complex number class and exploit its known properties. Nevertheless, it is not likely that such optimizations will be successful in all cases.

Spar/Java uses a more robust solution: it introduces a new primitive type `complex`. The operators `*`, `/`, `+`, and `-` are generalized to handle complex numbers; and narrowing and widening conversions are generalized. Also, a wrapper class `java.lang.Complex` is added, similar to e.g. `java.lang.Double`. The class

`java.lang.Complex` also contains a number of transcendental functions similar to those in `java.lang.Math`. To simplify the notation of complex constants, a new floating point suffix '`i`' has been added to denote an imaginary number. Together these additions allow code like `complex c1 = 1.0+2.0i` to be used.

Philippsen and Günthner [13] propose to add a `complex` type to Java in a way that is very similar to Spar/Java. Instead of the imaginary floating point suffix '`i`' in Spar/Java, they use a new keyword '`I`' that represents $\sqrt{-1}$. The Spar/Java approach uses syntax that was previously illegal, and therefore does not break existing programs.

5 Parallel processing

Parallel processing is another important aspect for scientific computation. Java has threads as a basic mechanism for concurrency and its is tempting to use threads for this purpose. There are, however, a number of problems with the standard notion of Java threads. First of all, the Java thread model has a very complicated memory model. This inhibits many optimizations or requires sophisticated analysis. Secondly, there is no standard mechanism to spawn threads in parallel. Thirdly, parallelism with threads is very explicit and hence suffers from all the classical programming problems, such as deadlock prevention.

In its simplest form, parallel processing can be done using a library of support methods build on top of standard Java threads. Such a library is described, for example, by Carpenter et al. [9]. The fact that standard Java can be used makes this approach attractive, but expressiveness is limited, and it is difficult for a compiler to generate efficient parallel code in this setup.

For this reason, many proposals extend Java with language constructs for parallelization. Note that many parallelization approaches require multi-dimensional arrays, so a language extension is required anyway, as discussed in Section 2.

In Javar [5], a parallel loop is identified with a special annotation. Since Javar annotations are represented by special comments, Javar programs are compatible with standard Java compilers.

Blount, Chatterjee, and Philippsen [6] describe a compiler that extends Java with a `forall` statement similar to that of HPF. To execute the `forall` statement, the compiler spawns a Java thread on each processor, and the iterations are evenly distributed over these threads. Synchronization between iterations is done by the user using the standard Java synchronization mechanism. No explicit communication is performed; a shared-memory system is assumed. Due to the dynamic nature of the implementation, they can easily handle irregular data and nested parallelism.

Spar [14] provides a `foreach` loop that specifies that the iterations of the loop can be executed in arbitrary order, but once an iteration is started, it must be completed before the next iteration can be started. Arrays can be annotated with HPF-like distribution pragmas to indicate on which processor the data must be stored. Additionally, Spar allows code fragments to be annotated with distribution information to indicate where that code must be executed.

6 Conclusions

In this paper a number of prospects and problems in using Java for scientific computation have been identified. It has been shown that Java has potential to serve the scientific community. However, much research still has to be done. A number of language extensions would make Java much more attractive for scientific computation, in particular support for multi-dimensional arrays, complex numbers, and efficient support for template classes.

References

1. Ninja web site. url: `www.research.ibm.com/ninja`.
2. Class `com.isml.math.complex`. url: `http://www.vni.com/corner/garage/-grande/complex.htm`.
3. JAMA: Java matrix package. url: `math.nist.gov/javanumerics/jama`.
4. P. V. Artigas, M. Gupta, S. P. Midkiff, and J. E. Moreira. High performance numerical computing in Java: Language and compiler issues. In *Proceedings of the 12'th Workshop on Language and Compilers for Parallel Computers*, Aug. 1999.
5. A. J. Bik and D. B. Gannon. Automatically exploiting implicit parallelism in Java. *Concurrency, Practice and Experience*, 9(6):579–619, June 1997.
6. B. Blount, S. Chatterjee, and M. Philippsen. Irregular parallel algorithms in Java. In *Irregular'99: Sixth International Workshop on Solving Irregularly Structured Problems in Parallel*, Apr. 1999.
7. G. Bracha. JSR 000014 – add generic types to the Java programming language, 2000. url: `java.sun.com/aboutJava/communityprocess/jsr/-jsr_014_gener.html`.
8. G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. GJ specification. Technical report, Bell Labs, May 1998. url: `www.cs.bell-labs.com/who/wadler/pizza/gj/-Documents/index.html`.
9. B. Carpenter, Y.-J. Chang, G. Fox, D. Leskiw, and X. Li. Experiments with "HPJava". *Concurrency, Practice and Experience*, 9(6):633–648, June 1997.
10. J. E. Moreira. JSR 000083 – Java multiarray package, 2000. url: `java.sun.com/-aboutJava/communityprocess/jsr/jsr_083_multiarray.html`.
11. J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence. Java programming for high performance numerical computing. *IBM Systems Journal*, 39(1):21–56, 2000.
12. A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for Java. In *Proceedings of the 24th AMD Symposium on Principles of Programming Languages*, pages 132–145, Jan. 1997.
13. M. Philippsen and E. Günthner. Complex numbers for Java. *Concurrency: Practice and Experience*, 12(6):477–491, May 2000.
14. C. v. Reeuwijk, F. Kuijlman, and H. Sips. Spar: an extension of Java for scientific computation. In *Proceedings of the Joint ACM Java Grande - ISCOPE 2001 Conference*, June 2001.
15. K. Thorup. Genericity in Java with virtual types. In *European Conference on Object-Oriented Programming, LNCS 1241*, pages 444–471. Springer Verlag, 1997.
16. K. Yelick, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: a high-performance Java dialect. In *ACM Workshop on Java for High-Performance Network Computing*, pages 1–13, Feb. 1998.