RC 21369 (96233) 21DEC98 Computer Science/Mathematics

Research Report

A Standard Java Array Package for Technical Computing

José E. Moreira, Samuel P. Midkiff, Manish Gupta
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - T. J. Watson - Tokyo - Zurich

This page intentionally left blank.

A Standard Java Array Package for Technical Computing

José E. Moreira Samuel P. Midkiff Manish Gupta
IBM T. J. Watson Research Center
Yorktown Heights NY 10598-0218 USA
{jmoreira,smidkiff,mgupta}@us.ibm.com

Abstract

Java is almost universally recognized as a good object-oriented language for writing portable programs. However, it still lags behind Fortran and C in performance, particularly for computationally intensive numerical programs. In this paper we present a true multidimensional Array package, and related compiler support, that can bring Fortran-like performance to Java numerical codes. We discuss the main design principles for our Array package, and illustrate the benefits from its use through several examples.

1 Introduction

Industry and academia alike have recognized the potential of Java[1] as an excellent environment for developing large-scale applications [10]. We can list several features of Java in support of this statement: (i) its unprecedented level of cross-platform portability; (ii) its clean object-oriented model; (iii) its built-in support for complicated operations such as networking and graphics; and (iv) its rapidly expanding programmer base. Nevertheless, Java must provide performance comparable to what is achieved with more conventional languages (*e.g.*, Fortran and C) in order to become the true environment of choice.

Despite the substantial gains in performance exhibited by the newer implementations of Java, there are still some unresolved issues. In particular, the Numerics Working Group of the Java Grande Forum (JGNWG) has identified five critical Java Language and Java Virtual Machine issues related to the performance of numerical computing in Java [10]:

1. *Complex arithmetic:* Complex numbers are essential in many areas of science and engineering. They must be properly supported in Java.

2. *Lightweight objects:* It is important to efficiently support classes that implement alternative arithmetic systems (*e.g.*, complex, interval, and decimal arithmetic) without paying the penalty usually associated with Java objects.

3. *Operator overloading:* Notational convenience is important for people operating with complex numbers and other alternative arithmetic systems.

4. *Use of floating-point hardware:* Application performance and accuracy can benefit from the exploitation of unique features of a machine. Examples include the 80-bit format in the IA-32 architecture and the `fma` (fused multiply-add) instruction in the POWER architecture.

[1]Java is a trademark of Sun Microsystems, Inc.

5. *Multidimensional arrays:* These are the most common data structures in technical computing. (One- and two-dimensional arrays are used to represent vectors and matrices in linear algebra. Physical structures are typically discretized and represent by arrays of dimension three and higher.) Operations on multidimensional arrays of numerical types must be performed efficiently in Java.

This paper is devoted to the last topic, multidimensional arrays. (However, the issue of lightweight objects is also touched on.) Our goal is to develop a Java package (a collection of classes) that provides the functionality and performance commonly associated with Fortran 90 arrays [1]. We start in Section 2 by describing the standard way to represent multidimensional arrays in Java, and the problems associated with it. In Section 3 we discuss the design principles of our Array package both from a usability (interface) as well as a performance perspective. Section 4 reports some experience and performance results with the Array package in linear algebra, PDE, and data mining codes. Finally, we present our conclusions and directions for future work in Section 5.

2 Java Arrays

The Java Language and the Java Virtual Machine directly support only one-dimensional arrays. Two-dimensional array are simulated with a one-dimensional array whose elements themselves are one-dimensional arrays. Higher dimensional arrays are constructed using a one-dimensional array whole elements are arrays of the next lower dimension. This approach allows great flexibility, and supports complicated data structures as shown in Fig. 1(a). In that figure, X and Y are arrays of references to rows of elements, declared as double[ ][ ] X, Y. $X[0]$ and $X[1]$ are two references to the same row, as are $X[4]$ and $Y[3]$.

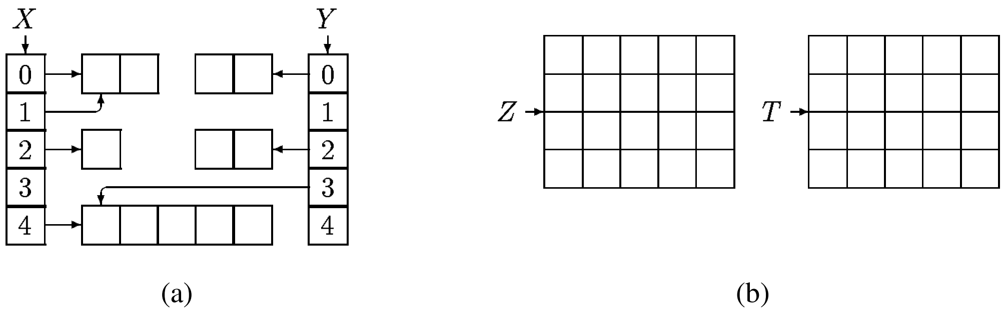


FIG. 1. *Examples of different array types: (a) array of arrays, (b) rectangular two-dimensional array.*

The flexibility provided by Java arrays of arrays (hereafter called Java arrays) translates into a nightmare for optimizing compilers. Most of the time, technical application programmers want to represent rectangular data structures, as shown by Z and T in Fig. 1(b). However, when faced with Java arrays, a compiler has to be prepared for the possibility that they represent something more like Fig. 1(a) than Fig. 1(b). (There has been some compiler work in automatically recognizing that Java arrays are being used to represent rectangular structures [7].) Java arrays present two main difficulties to optimizing compilers: (i) shape volatility and (ii) aliasing disambiguation.

Shape volatility arises from the possibility of dynamically changing the structure of an array of arrays. For example, in Fig. 1(a), an assignment $X[4] =$ **new** double[3] would at the same time break the aliasing between $X[4]$ and $Y[3]$ and change the length of row $X[4]$ from 5 to 3. Note that this change in shape can happen in any thread that has access to X, and will be reflected to all threads using X. We can illustrate the impact of shape volatility in one important Java optimization: the elimination of run-time bounds checks. Consider the matrix-multiply code in Fig. 2. According to the Java specification, all references to $c[i][j]$ have to be tested to make sure that i and j are valid indices for their respective arrays. (The discussion also extends to arrays a and b.) The compiler can eliminate these tests from the loop body if it can show that c has at least m rows and at least p columns. (This is typically done at run-time and one of two loop versions, with or without tests, is chosen for execution.)

```
void matmul(double[ ][ ] a, double[ ][ ] b, double[ ][ ] c, int m, int n, int p ) {
  for (i = 0; i < m; i++)
    for (j = 0; j < p; j++)
      for (k = 0; k < n; k++)
        c[i][j] += a[i][k] * b[k][j];
}
```

FIG. 2. *Matrix-multiply code with Java arrays.*

The number of rows of c is c.**length**, and can be obtained inexpensively at run-time. Finding the number of columns of c is a different matter. There is no guarantee that all the rows have the same length, and no guarantee that another thread is not dynamically replacing some of the rows of c with rows of different length. (Note that we are not concerned with the values of the elements of c, just its shape.) To enable bounds checking optimization in this case, it is necessary to privatize c: a local copy is created by copying the array of references to rows. This local copy can be created with code as follows.

(1)

```
double[ ][ ] c' = new double[c.length][ ];
c_cols = ∞;
for (r = 0; r < c.length; r++) {
  c'[r] = c[r];
  c_cols = min(c_cols, c[r].length);
}
```

The number of columns of c, c_{cols}, can be computed during privatization as the length of its shortest row.

The matrix-multiply code of Fig. 2 is an example of a code that modern Fortran compilers would attempt to optimize with high-order loop transformations (*e.g.*, loop interchange, tiling, unroll-and-jam) [18]. These transformations typically change the order in which computations are performed in a loop nest and care must be taken to guarantee that the new order remains legal. To apply these loop transformation to the Java code of Fig. 2 the compiler must guarantee that (i) the loop nest is free of exceptions (*i.e.*, it will execute from beginning to end) and (ii) all elements of c are independent (no intra-array aliasing) and separate from the elements of a and b (no inter-array aliasing). Aliasing disambiguation is a key analysis to enable high-order transformations. In Fortran, to show that $a(i, j)$ is not aliased to $b(k, l)$ it suffices to show that a and b refer to different arrays or that the expression $(i \neq k) \vee (j \neq l)$ is **true**. The same does not apply to Java: in Fig. 1(a) $X[0][1]$ is the same element as $X[1][1]$ and $Y[3][0]$ is the same element as $X[4][0]$.

Just as the shape of Java arrays can be determined with an array traversal code like (1), a disambiguation test can also be performed at run-time. This disambiguation, however, is a more expensive operation, as we have to show that no two rows of the same or different arrays are aliased. Whereas the privatization and shape determination is of complexity $O(n)$, where n is the array extent, disambiguation is of complexity $O(n^2)$. (This can be reduced to $O(n \log n)$ if we are able to order the references. This cannot be done at the Java source code or bytecode level, but can typically be done inside the Java Virtual Machine.)

3 The Array Package

We developed the Java Array package to overcome the difficulties associated with representing multidimensional rectangular structures through Java arrays. We refer to the rectangular multidimensional arrays from the Array package as **array**s. In designing the Array package we adopted the following design principles:

1. The package must provide Fortran 90-like array functionality. This includes the ability to manipulate regular and irregular array sections.
2. The package, as defined by its reference implementation, must work on existing JVMs and yet be general enough to be useful for a long time.
3. The Array package must be able to support non-primitive numerical types, such as complex and decimal numbers.
4. Good performance in array operations is achieved through compiler technology (in particular, *semantic inlining* [20] discussed below). The reference implementation is kept clean and general.
5. The internal layout of data in the Array package is not exposed, and it cannot be inferred by any test. This allows more optimization at the implementation level.
6. Array operations follow a transactional model. Each operation either terminates with an exception, in which case no visible data is modified, or it completes successfully.
7. Array operations are defined as to facilitate optimization (*e.g.*, through high-order loop transformations) and parallelization.
8. Access to array elements and array sections can be implemented efficiently using existing Java technology. (At least for primitive data types.)

An **array** from the Array package is characterized by three immutable properties: (i) its *rank* (number of dimensions or axes), (ii) its elemental data *type* (all elements of an **array** are of the same type), and (iii) its shape (the extents along its axes). Immutable rank and type are important properties for effective code optimization using existing techniques developed for Fortran and C. Immutable shape is an important property for the optimization of run-time bounds checking according to recent techniques developed for Java [14, 16].

The rank and type of an **array** are defined by its class. The **array** classes in the Array package are of the form <*type*>**Array**<*rank*>. Supported types include all Java primitive types (boolean, byte, char, short, int, long, float, and double), as well as alternative arithmetic systems (*e.g.*, **Complex** and **Decimal**) themselves implemented through classes. Supported ranks include **0D** (scalar), and **1D** (one-dimensional) through **7D** (seven-dimensional). Note that rank 7 is the current standard limit for Fortran.

The shape of an **array** is specified at its creation. As an example,

$$\textbf{doubleArray2D}\ A = \textbf{new doubleArray2D}(m, n)$$

```
void matmul(doubleArray2D a, doubleArray2D b, doubleArray2D c, int m, int n, int p) {
  for (i = 0; i < m; i++)
    for (j = 0; j < p; j++)
      for (k = 0; k < n; k++)
        c.set(i, j, c.get(i, j) + a.get(i, k) * b.get(k, j));
}
```

FIG. 3. *Matrix-multiply code with the Array package.*

creates an $m \times n$ two-dimensional array of doubles. The shape parameters (m and n) for an **array** must be nonnegative integer expressions (*i.e.*, they must evaluate to a value greater than or equal to zero and less than or equal to 2147483647). If an **array** of the specified shape cannot be created because of memory limitations, then an `OutOfMemoryError` must be thrown. **Array**s can also be created as sections of another **array**. An example is:

doubleArray3D $A =$ **new doubleArray3D**(m, n, p);
doubleArray2D $B = A$.**section**(**new Range**$(0, m-1)$, **new Range**$(0, n-1), k)$;
doubleArray1D $C = B$.**section**(**new Range**$(0, m-1, 2), j)$;

The first statement creates an $m \times n \times p$ three-dimensional array A. The second statement extracts an $m \times n$ two-dimensional section B corresponding to the k-th plane of A. The third statement extracts all even-indexed elements of the j-th column of B, which corresponds to the j-th column of the k-th plan of A.

Elements of an **array** are identified by their indices along each axis. Let a k-dimensional array A of elemental type T have extent n_j along its j-th axis, $j = 0, 1, \ldots, k-1$. Then, a valid index i_j along the j-th axis must be greater than or equal to 0 and less than n_j. An attempt to reference an element $A(i_0, i_1, \ldots, i_{k-1})$ with any invalid index i_j causes an `ArrayIndexOutOfBoundsException` to be thrown.

Fig. 3 shows a matrix-multiply written using the Array package routines. Three two-dimensional **array**s of doubles, a, b and c, are declared. The computations occurs on the line:

$$c.\mathbf{set}(i, j, c.\mathbf{get}(i, j) + a.\mathbf{get}(i, k) * b.\mathbf{get}(k, j));$$

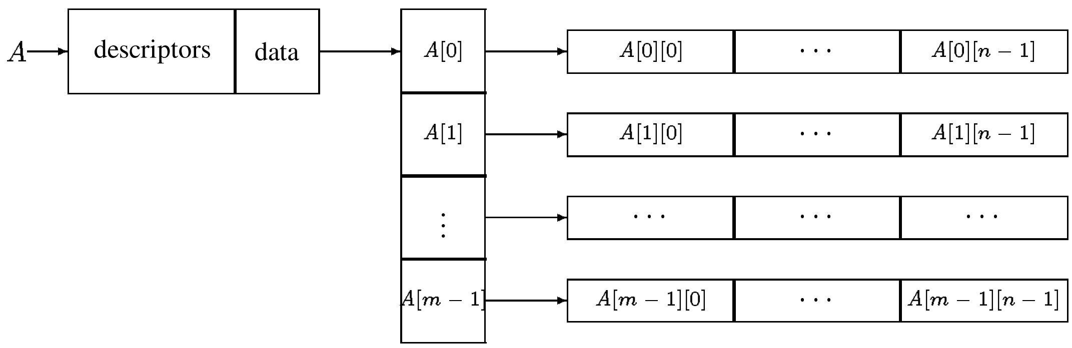
The **get** methods retrieve an element of the **array**, taking as arguments subscript expressions for each dimension. Although not shown here, each subscripts may be an integer expression, a *range* or a *sequence* of elements. A range is semantically equivalent to a Fortran 90 triplet, whereas a sequence is an arbitrary list of integers. Both are represented by objects, a range through the **Range** class and a sequence through the **Index** class. If all the indices are integer expression, then **get** returns only the value of an element. If any of the indices is a range or a sequence, **get** returns the corresponding **array** of values. (Note that **get** returns a new **array**, not a section of the original **array**.) The **set** method takes, in addition to the subscript functions, an additional parameter specifying the value for the corresponding **array** elements. In addition to these basic accessor methods, the Array package also provides routines to perform aggregate operations on arrays, including element-by-element relational, logical, and arithmetic operators. The rest of this section describes the issues involved in designing and implementing a standard Array package.

3.1 Reference and conforming implementations

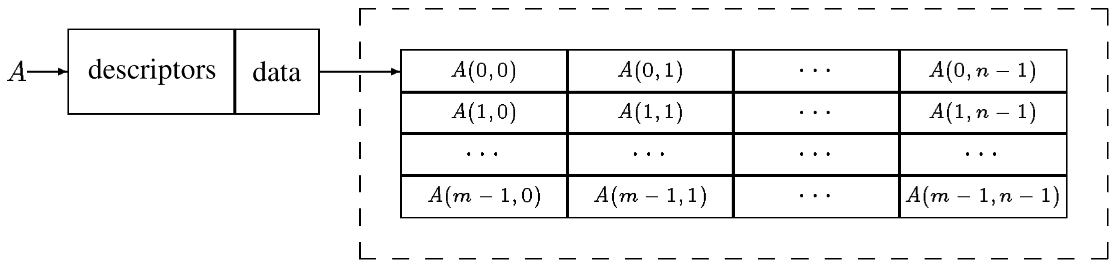
When discussing the standard array package, a distinction must be drawn between the *reference* implementation of the package, and a *conforming* implementation. The purpose of the reference implementation to is exactly represent the functional semantics of the Array package in a Java implementation. The implementations of those attributes of the Array package that are not visible from outside the package (*e.g.*, the layout of storage) are not significant, whereas those attributes that are visible (*e.g.*, referencing elements by their coordinates in a Cartesian space) are significant. The goal of the reference implementation is to provide a clear representation of the Array semantics, and to provide an executable standard to test compliance of conforming implementations.

The reference implementation of the Array package is written entirely in standard Java. As a consequence, it may be run on any JVM that supports the corresponding JDK level as used in its development (JDK 1.1.6). The straightforward reference implementation uses an n-dimensional Java array to represent an n-dimensional rectangular **array**. This is necessary to properly implement the Array package semantics which allows each dimension of an Array to have an extent of up to $2^{31}-1$. A 2-dimensional array in the reference implementation is shown in Fig. 4(a).

A conforming implementation of the Array package takes advantage of the fact that the layout of data is not visible and that the available memory on a machine executing the methods from the Array package is environment-dependent. This allows for a higher performance version of the Array package (at least for arrays of Java primitives), that maps an n-dimensional **array** into a 1-dimensional Java array. This provides a representation as shown in Fig. 4(b), with the constraint that the maximum number of elements in the array is $2^{31}-1$. Should the number of elements of the Array exceed $2^{31}-1$, an `OutOfMemoryError` exception is thrown.



(a) Reference implementation form of the object.



(b) Conforming implementation for the object.

FIG. 4. *Multidimensional array objects.*

A more sophisticated implementation can be accomplished in cooperation with a static or dynamic Java compiler and the technique of semantic inlining. This allows the number of elements of the Array to be constrained only by the available memory, while preserving contiguous storage [20].

3.2 Other features of the Array package

Integer and real types constitute the bulk of numerical programming data types, but other numerical types are also important. Complex numbers are pervasive in technical computing, and in the commercial world, various *decimal* types are important. With semantic inlining (discussed below) these non-primitive numeric types can be handled as efficiently as integer and floating point arithmetic types. The **set** and **get** accessor methods are defined to take and return references to objects of the non-primitive type.

One valuable feature of Fortran 90 arrays is the ability to use subsections of an array in most places an array can be used. Any changes to the subsections are reflected in the larger array. This support is provided in Fortran with almost no overhead. We follow a similar approach in the Array package to provide the same sort of support. As Fig. 4 shows, an **array** object consists of a set of descriptor fields and a pointer to data storage. The **array** descriptors contain array section information, in particular the offset of the first element and stride value along each axis. Because the base **array** and all its subsections point to the same storage, changes to one are reflected in all. The only difference between the base **array** and **array** sections are the values in the array descriptor fields.

3.3 Achieving performance in the Array package

Having an unspecified physical data layout gives reference and conforming implementations flexibility in using a layout that provides a clean implementation, good performance, or both. For reference implementations, the lack of a specified layout allows whatever layout is provided by the JVM that executes the reference implementation to be used. In other situations, simplicity of implementation is generally less important than performance, and the lack of a specified layout allows greater flexibility in the data representation used by the JVM. Thus, if an **array** is passed to a variety of native calls that expect a C-like row-major layout, the compiler can keep the data in row-major order. Likewise, if the **array**s are passed by native interfaces to programs expecting a Fortran-like layout, the arrays can be kept in column-major order. More importantly, new forms of data layout, like recursive partitioning [9] for dense structures or layouts that support irregular and sparse structures [2, 3], can be transparently incorporated into a sophisticated JVM while maintaining strict compatibility with earlier or less sophisticated JVM's. In fact, the layout can be chosen on an individual **array** basis, leading to better support for parallelization and memory hierarchy optimizations such as described in [11, 12].

Obviously, invoking a method for every access and operation on an array element will generally lead to poor performance. Therefore, we rely on compiler technology for performance. The most important compiler technology is *semantic inlining*. Semantic inlining is a compiler technology that treats calls to known methods on known data types as language primitives. Because the method is recognized, knowledge about its semantics can be embedded in the compiler. The compiler can then replace a method call by an intermediate language representation of its semantics. Thus the semantics, but *not* the actual logic and code of the method, are inlined. The practical advantage of this is that almost all of the overhead of accessing and operating on object fields can be eliminated.

As an example, consider a two-dimensional **array** A of doubles. The method call $A.\mathbf{get}(2{*}i, j)$ can be replaced (in the intermediate representation of the program) by instructions that test the legality of the access and then directly index into the specified element. Ignoring for the moment the overhead of the bounds check (which can be optimized away as shown in [14, 16]), the indexing operation on this **array** can be performed as cheaply as a Fortran indexing operation.

As another example, consider two-dimensional **array**s A and B of complex numbers. From the programmers point of view, each element of A is a complex number, whose elements have values assigned to them from **Complex** objects passed to the **set** method, and whose values are accessed via a **Complex** object returned by the **get** method. Consider the operation A.**set**(B.**get**(i, j).**plus**(A.**get**$(i, j)), i, j)$ (which corresponds to $A(i, j) = B(i, j) + A(i, j)$). A naive translation of this expression would create temporary objects for each of the **get** operations and for the result of the **plus** operation. With semantic inlining, it can be recognized that the consumer of A.**get** is a standard operation that only needs the values of the complex number, thus the object creation can be dispensed with. Likewise, no **Complex** object is needed to hold element (i, j) of B, since the **plus** operation only needs the complex values. Similarly, even though the **set** interface requires a **Complex** object, the compiler knows that only the values are really needed, and directly writes the value results of the complex addition to the specified element, without creating an intermediate temporary object. Thus, in evaluating the expression, a smart compiler can eliminate four method calls and three object creations.

Aggregate **array** operations follow a transactional model which facilitates high-order transformations and usability by application programmers. At the beginning of each standard aggregate **array** operation, a check is made to determine if any violations (*e.g.*, bounds and **null**-pointer) may happen during the actual computation. If so, an exception is thrown and the method immediately terminates without affecting any visible data. Otherwise, the computation proceeds knowing that it will execute to the end. (Things get slightly more complicated when integer divisions are present, but the transactional model is still supported.) This transactional model has two major benefits. First, application programs can be written knowing that an operation either completes successfully or no data values are changed. This allows for easier error recovery in the case of an exception. From the compiler perspective, this means that the only exceptions that can be thrown in the method implementing the operation are those exceptions thrown in the prologue of the operation. Java mandated *implicit* exceptions, such as bounds violations and null pointer exceptions at **array** element accesses, do not occur. JVM errors, such as `OutOfMemoryError`, also occur only at the prologue.

We illustrate the concept of transactional model for **array** operations through an example. Fig. 5 shows an implementation for the **plusAssign** method of the **doubleArray2D** class, which implements the operation $a = a + b$ for two conforming arrays a and b. First, the shape $(m \times n)$ of a is obtained and compared to the shape of b. If a and b do not have the same shape, the operation terminates immediately with an exception. If the operation can proceed, an **array** c is prepared to hold the result of $a + b$. In general, a and b could be aliased, referring to overlapping sections of a common base **array**. If that is the case, as determined by the **intersects** method, then a new **array** must be created for c. Otherwise, c can be made equal to a. The first loop nest computes $c = a + b$ and the second loop nest, if necessary, copies the result from c back into a. Because of the transactional model and the **array** semantics, the two loop nests can be extensively optimized and parallelized without changing the semantics of the **plusAssign** method.

In user code that directly manipulates **array** elements, the array bounds checking and **null**-pointer checking optimizations techniques described in [14] can be used to create safe regions and eliminate run-time tests. The Array package aids these optimization technique by constraining all aliasing to occur between references to **array**s, and not between different rows of **array**s. Also, the rectangular and immutable shape of **array**s makes the privatization step discussed in Section 2 unnecessary. Eliminating the possibility of run-time exceptions in loop nests opens the door to

```
doubleArray2D plusAssign(doubleArray2D b) throws NonconformingArrayException {

   doubleArray2D a = this;
   doubleArray2D c = null;

   int m = a.size(0);
   int n = a.size(1);
   if (m ≠ b.size(0)) throw new NonconformingArrayException();
   if (n ≠ b.size(1)) throw new NonconformingArrayException();

   if (a.intersects(b)) c = new doubleArray2D(m, n);
   else c = a;

   for (int i = 0; i < m; i++)
      for (int j = 0; j < n; j++)
         c.set(i, j, a.get(i, j) + b.get(i, j));

   if (c ≠ a) {
      for (int i = 0; i < m; i++)
         for (int j = 0; j < n; j++)
            a.set(i, j, c.get(i, j));
   }

   return a;
}
```

FIG. 5. *The method* **plusAssign***, which implements* $a = a + b$ *for two-dimensional* **array***s.*

much more aggressive reordering optimizations. In [16], speedups of well over 100 are reported for single node programs, and [13] shows that a Java application can break the 1 Gigaflop barrier.

4 Experiments

In this section we illustrate through examples some of the performance benefits from using the Array package. We first demonstrate the large performance gains that can be obtained through semantic inlining of operations with complex numbers. We proceed by showing that the freedom of data layout allowed by the Array package also improves the performance of numerical codes. Finally, we show the improvements achieved in a production-level data mining code when operations with Java arrays were replaced by operations with **array**s from the Array package.

We used two different machines in our investigations. Some experiments were performed on an RS/6000 model F50. This machine has four 332 MHz PowerPC 604e processors and 1 GB of main memory. Peak performance of each processor is 664 Mflops. We also performed experiments on an RS/6000 model 590. This machine has a single 67 MHz POWER2 processor, 512 MB of main memory, and a peak performance of 266 Mflops.

4.1 Complex arithmetic operations

Numerical codes with complex arithmetic are a real challenge for efficient execution in Java. Representing each complex number as an object results in a voracious rate of object creation and destruction during computation. Therefore, the performance of Java applications that use Java arrays of **Complex** objects is orders of magnitude less than equivalent Fortran applications. This problem can be solved by using a combination of **Complex** arrays in the Array package and semantic inlining optimizations.

A Java code that uses the **Complex** arrays in the Array package continues to perform operations on **Complex** objects. Direct execution of such code by a conventional JVM is also very slow. However, a compiler with semantic inlining optimization can replace the operations on **Complex** objects by operations on *complex values*. That is, it represents and operates on (for as long as possible) only the values of complex numbers. The conversion from values to actual **Complex** objects needs to be performed only when the compiler finds an operation outside its understanding. This is a very infrequent event during most numerical computations.

We analyze the impact of using the Array package and semantic inlining for complex arithmetic through three benchmarks: MICROSTRIP, CFD, and LU. MICROSTRIP [15] computes the electric potential field of a microstrip structure in the presence of sinusoidal voltages. It uses an iterative Jacobi solver for the PDE defining the field. The microstrip structure is discretized by a 1000×1000 grid. CFD is a computational fluid dynamics kernel. It performs convolutions on three pairs of two-dimensional functions. Each function is represented as a 256×256 array of complex entries. LU is a straightforward implementation of Crout's algorithm for performing the LU decomposition of a square matrix A [8] of complex numbers, with partial pivoting. The factorization is performed in place and A is of size 500×500.

Fig. 6 shows the results for our experiments with MICROSTRIP, CFD, and LU on an RS/6000 590. For each benchmark, results for three implementations are reported: (i) a Java implementation using Java arrays of **Complex** objects (the Java bar); (ii) a Java implementation using the Array package and semantic inlining (the Array bar); and (iii) a Fortran implementation compiled with the highest level of implementation (the Fortran bar). For each benchmark the height of the bars are normalized to the Fortran performance. The numbers on top of the bars represent actual achieved Mflops.

It is evident that the performance of versions with Java arrays is vastly inferior to Fortran. In fact, it would be extremely hard to justify using Java in these situations. However, the versions using the Array package and semantic inlining achieve between 60 to 90% of Fortran performance. In the case of MICROSTRIP, this represents a 75-fold speedup over the Java arrays approach. Overall, the Array package and semantic inlining make Java very competitive with Fortran for computations with complex numbers.

4.2 Impact of array layout

The implementor of the Array package has complete freedom for choosing the data layout of an array. In fact, the layout can be chosen on a per-array basis. Our particular implementation for these experiments uses a row-major dense layout that maps multidimensional arrays into a one-dimensional vector of data elements. (This corresponds to a C-like layout.) Our Java environment also stores each row of a Java two-dimensional array as a dense vector, but there is no predefined relation between the position of two rows. Accessing an element of a **doubleArray2D** requires only index arithmetic, while accessing an element of a double[ ][ ] also requires pointer chasing. The array layout and the access techniques impact the ability of the compiler in extracting good performance from the code.

Fig. 7 compares the performance, on a single processor of an RS/6000 F50, of two equivalent implementations of matrix-multiply (MATMUL): one with Java arrays and the other with the Array package. Both versions are optimized with tiling, unroll-and-jam, and scalar replacement. The plots

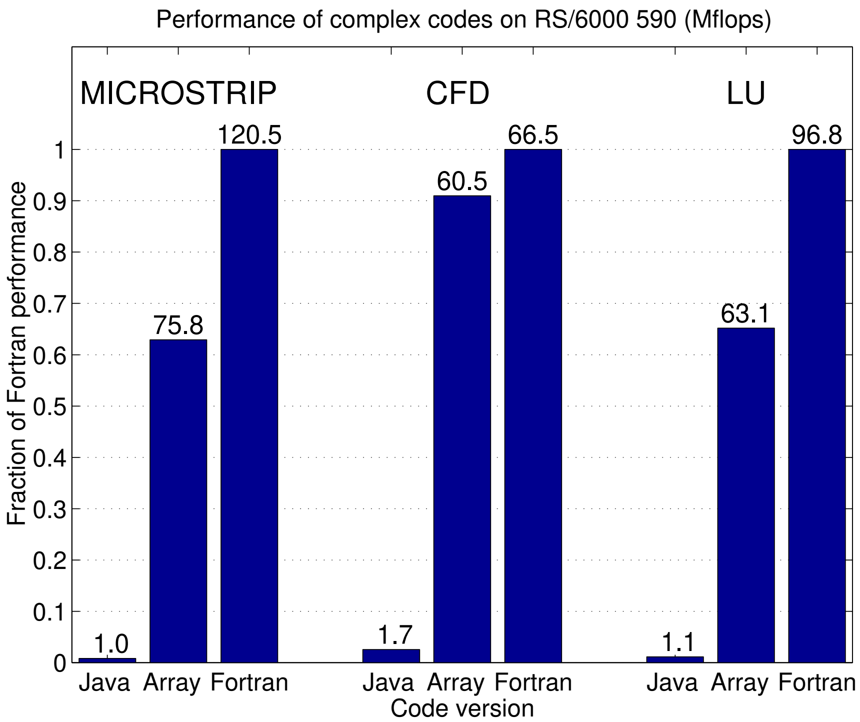


FIG. 6. *Complex arithmetic with and without the Array package.*

show the Mflops achieved by the two versions for different sizes of the problem matrices. Fig. 7(a) presents the performance when the *fused multiply-add* (`fma`) instruction in of the PowerPC *is not* used (as mandated by the current Java standard). Fig. 7(b) presents the performance when the `fma` instruction *is* used (as proposed by the Java Grande Forum).

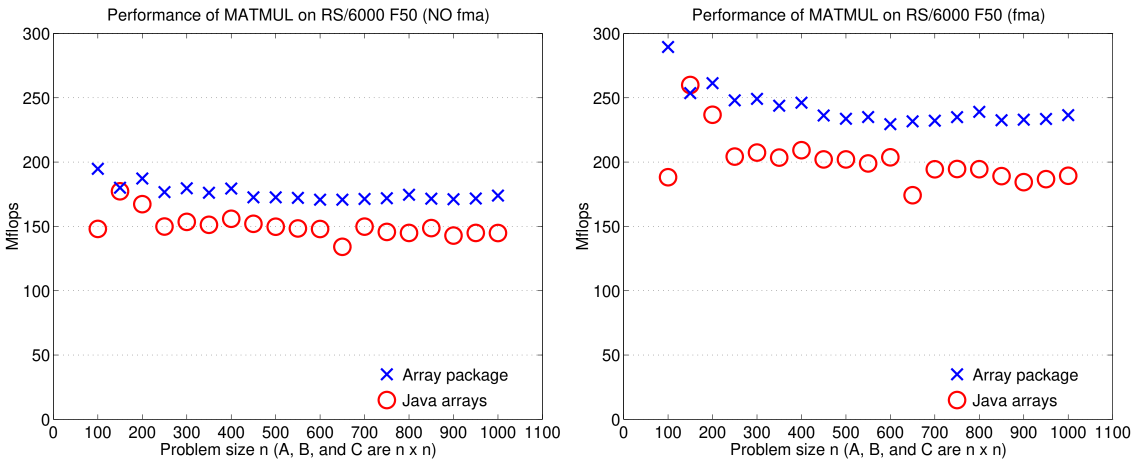


FIG. 7. *Matrix-multiply with Java arrays and the Array package.*

We observe that the dense, one-dimensional, row-major organization in the Array package results in consistently better performance when compared to the organization of Java arrays. This advantage is approximately 15% without `fma`'s and 20% with `fma`'s. (The Java array version also performs array privatization, but that has a performance impact of less than 1%.) The Array package version with `fma` achieves approximately 75% of the performance of the highly optimized matrix-multiplication routine in ESSL (290 Mflops for matrices in that size range).

Recent work on recursive blocking layout for arrays has shown the benefit of this approach as compared to the conventional row- and column-major approaches [9]. With the Array package, recursive blocking and other advanced layouts can be implemented transparently to the application programmer. The compiler can also aggressively adjust the organization of arrays for better memory behavior and parallelism [11, 12]. As opposed to Fortran and C, there is no risk of modifying the semantics of a program through changes in the organization of a multidimensional array from the Array package.

4.3 A data mining application

This last example illustrates our experience with developing a production-quality data mining application in Java [17]. The application is a recommendation code for suggesting new products to customers based on their previous spending behavior. The algorithm involves operations between two sparse matrices: an affinity matrix A and a spending matrix S. Matrix A has size 10350×2103 and $397,559$ non-zeros (1.8% of non-zeros). Matrix S has size 4800×2103 and $231,194$ non-zeros (2.3% of non-zeros). We experimented with two version of the application, one that operates directly on the elements of Java arrays and another that uses the multidimensional arrays and array operations from the Array package. Many of the operations in the Array package are parallelized, thus taking advantage of the multiple processors of the RS/6000 F50 where these experiments were performed. (The data mining code is still a sequential code, parallelism is exploited inside the Array package operations, totally transparent to the the application.) As a performance reference, we compare the Java codes to a version in Fortran.

Results from our experiments with the data mining application are shown in Fig. 8. The heights of the bars are relative to the best single-processor Java performance, and the numbers on top of the bars represent achieved Mflops for each version. The Java arrays version (Java bar) achieves 21 Mflops. The single-processor Java version with the Array package (Array x 1 bar) does significantly better, achieving 109 Mflops. Note that the Fortran version (Fortran bar) is only 10% better, at 120 Mflops. As we run the Array package version on more processors (2, 3, and 4 processors corresponding to the Array x 2, Array x 3, and Array x 4 bars, respectively), performance of the data mining application increases. We achieve a speedup of 2.7 on 4 processors, with no effort on the part of the application programmer. This shows that the parallelism inside the Array package can indeed be beneficial at the application level.

5 Conclusion

We have developed the Array package to overcome the difficulties associated with representing multidimensional rectangular structures through Java arrays. The flexibility of Java arrays results in additional complications for optimizing compilers due to possible shape volatility and difficult aliasing disambiguation. The true multidimensional arrays from the Array package have immutable shape and are always rectangular. These properties make operations with them amenable to optimization using mature techniques developed for Fortran and C.

The Array package implements array operations following a transactional model: operations either complete successfully or terminate through an exception with no other visible side effects. This approach supports extensive optimization an parallelization of the operations while preserving semantics. We have demonstrated, through several example applications, that Fortran-like performance on numerical codes, with both real and complex arithmetic, can be achieved in Java with the

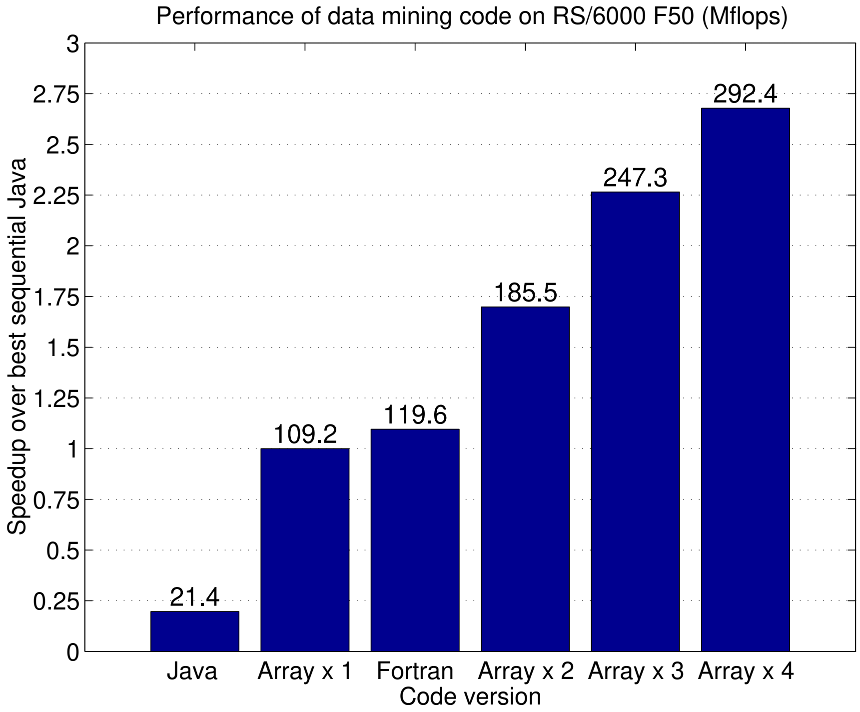


FIG. 8. *Experiments with a data mining code using Java arrays, the Array package, and Fortran.*

Array package.

Our Array package is implemented entirely in Java (no native code) and follows the orientation of the Java Grande Forum [10]. In fact, our Array package is a strawman proposal for possible standardization that has been released for comments. It can be found at `http://math.nist.gov/javanumerics/#proposals`, or through our project Web page at `www.research.ibm.com/ninja`.

Much work has been done in developing high-performance class libraries for numerical computing in Java, as described in [4, 5, 19]. The Array package fills one gap in Java support for technical applications. We are now starting to examine the interaction between Java libraries for linear algebra [6] and the Array package. We are confident that, through a combination of standard packages for numerical processing and appropriate compiler support, Java can become a high-performance environment for scientific and engineering computing.

References

[1] J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener, *Fortran 90 Handbook: Complete ANSI/ISO Reference*, McGraw-Hill, 1992.

[2] A. Bik, *Compiler Support for Sparse Matrix Computations*, PhD thesis, Leiden University, may 1996.

[3] A. Bik and H. Wijshoff, *Compilation techniques for sparse matrix computations*, in Proceedings of the 1993 International Conference on Supercomputing, 1993.

[4] B. Blount and S. Chatterjee, *An evaluation of Java for numerical computing*, in Proceedings of ISCOPE'98, vol. 1505 of Lecture Notes in Computer Science, Springer Verlag, 1998, pp. 35–46.

[5] R. F. Boisvert, J. J. Dongarra, R. Pozo, K. A. Remington, and G. W. Stewart, *Developing numerical libraries in Java*, in ACM 1998 Workshop on Java for High-Performance Network Computing, ACM SIGPLAN, 1998. Available at `http://www.cs.ucsb.edu/conferences/java98`.

[6] R. F. Boisvert, J. Hicklin, B. Miller, C. Moler, R. Pozo, K. Remington, and P. Webb, *JAMA: A Java matrix package*. URL: `http://math.nist.gov/javanumerics/jama/`, 1998.

[7] M. Cierniak and W. Li, *Just-in-time optimization for high-performance Java programs*, Concurrency, Pract. Exp. (UK), 9 (1997), pp. 1063–73. Java for Computational Science and Engineering - Simulation and Modeling II, Las Vegas, NV, June 21, 1997.

[8] G. H. Golub and C. F. van Loan, *Matrix Computations*, Johns Hopkins Series in Mathematical Sciences, The Johns Hopkins University Press, 1989.

[9] F. G. Gustavson, *Recursion leads to automatic variable blocking for dense linear algebra algorithms*, IBM Journal of Research and Development, 41 (1997), pp. 737–755.

[10] Java Grande Forum, *Report: Making Java work for high-end computing*. Java Grande Forum Panel, SC98, November 1998. URL: `http://www.javagrande.org/reports.htm`.

[11] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee, *A matrix-based approach to the global locality optimization problem*, in Proceedings of International Conference on Parallel Architectures and Compilation Techniques (PACT'98), Paris, France, October 1998.

[12] M. Kandemir, A. Choudhary, J. Ramanujam, N. Shenoy, and P. Banerjee, *Enhancing spatial locality via data layout optimizations*, in Proceedings of Euro-Par'98, Southampton, UK, vol. 1470 of Lecture Notes in Computer Science, Springer Verlag, September 1998, pp. 422–434.

[13] S. P. Midkiff, J. E. Moreira, and M. Snir, *Java for numerically intensive computing: From flops to Gigaflops*, Tech. Rep. 21351, IBM Research Division, December 1998. To appear in Proceedings of Frontiers'99.

[14] ———, *Optimizing bounds checking in Java programs*, IBM Systems Journal, 37 (1998), pp. 409–453.

[15] J. E. Moreira and S. P. Midkiff, *Fortran 90 in CSE: A case study*, IEEE Computational Science & Engineering, 5 (1998), pp. 39–49.

[16] J. E. Moreira, S. P. Midkiff, and M. Gupta, *From flop to Megaflops: Java for technical computing*, in Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing, LCPC'98, 1998. IBM Research Report 21166.

[17] J. E. Moreira, S. P. Midkiff, M. Gupta, and R. D. Lawrence, *Parallel data mining in Java*, Tech. Rep. 21326, IBM Research Division, 1998. Submitted for publication.

[18] V. Sarkar, *Automatic selection of high-order transformations in the IBM XL Fortran compilers*, IBM Journal of Research and Development, 41 (1997), pp. 233–264.

[19] M. Schwab and J. Schroeder, *Algebraic Java classes for numerical optimization*, in ACM 1998 Workshop on Java for High-Performance Network Computing, ACM SIGPLAN, 1998. Available at `http://www.cs.ucsb.edu/conferences/java98`.

[20] P. Wu, S. P. Midkiff, J. E. Moreira, and M. Gupta, *Improving Java performance through semantic inlining*, Tech. Rep. 21313, IBM Research Division, 1998. Submitted for publication.