RC 21481 (96939) 20MAY99 Computer Science/Mathematics

Research Report

Java Programming for High Performance Numerical Computing

J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, R. D. Lawrence
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

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

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center [Publications 16-220 ykt] P. O. Box 218, Yorktown Heights, NY 10598. email: `reports@us.ibm.com`

Some reports are available on the internet at `http://domino.watson.ibm.com/library/CyberDig.nsf/home`

This page intentionally left blank.

Java Programming for High Performance Numerical Computing

J. E. Moreira S. P. Midkiff M. Gupta P. V. Artigas M. Snir R. D. Lawrence
{*jmoreira,smidkiff,mgupta,artigas,snir,ricklawr*}*@us.ibm.com*

IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598-0218

Abstract

First proposed as a mechanism for enhancing Web content, Java has taken off as a serious general-purpose programming language. Industry and academia alike have expressed great interest in using Java as a programming language for scientific and engineering computations. Applications in these domains are characterized by intensive numerical computing, and often have very high performance requirements. In this paper we discuss programming techniques that lead to Java numerical codes with performance comparable to Fortran or C, the more traditional languages for this field. The techniques are centered around the use of high-performance numerical libraries, written entirely in Java, and on compiler technologies. Proper use of this package, and of other appropriate tools for compiling and running a Java application, results in code that is clean, portable, and fast. We illustrate the programming and performance issues through case studies in data mining and electromagnetism.

1 Introduction and motivation

Java is an attractive language for many fundamental reasons: clean and simple object semantics, cross-platform portability, security, and an increasingly large pool of adept programmers. The goal of this paper is to show that the major drawback to the use of Java as a language for numerical programming – performance – is not fundamental, and can be solved using techniques we describe.

The problem, stated bluntly, is that the performance of computationally intensive Java programs currently falls woefully short of Fortran performance. As shown in Section 5, the performance of many numerically intensive Java programs, compiled without the benefit of the techniques described in this paper, can be as low as 1% of the performance of equivalent Fortran programs. Although anecdotal evidence suggests that performance degradations of 10-30% relative to Fortran 90 might be tolerated in computationally intensive programs in order to gain the benefits of Java, a 100-fold performance slowdown is unacceptable.

The remainder of this section outlines what leads to these problems with current Java performance on numerically intensive computing. We focus on two aspects: multidimensional arrays and alternative arithmetic systems (particularly complex numbers). We then discuss why the corresponding Fortran 90 programs do not have these problems, and how the solutions employed by Fortran 90 can be adopted, while maintaining 100% pure Java and, almost as important, remaining within the philosophy and spirit of Java.

1.1 Java arrays

Although the specification of Java arrays does not mandate their exact organization, it places sufficient constraints that all implementations we know of appear as shown in Figure 1. The figure shows how a two-dimensional array would be laid out, and accessed, in Java.

Java does not directly support arrays of rank greater than one. Thus, a two-dimensional array is represented as an array-of-arrays: An array whose elements are, in turn, references to one-dimensional row arrays. Each array is represented as an *array object descriptor* followed by a chunk of storage which contains the elements of the array. The array object descriptor contains the fields that are present in all objects (lock bits, bits used by the garbage collector, type information, etc), as well as the upper bound of the array. If a two-dimensional array of type T is being represented, the elements of each row array are either of type T, if T is a Java primitive type, or references to objects of type T. In Figure 1, each element is a Java primitive of type `double`. Although the layout of the elements of the array are not specified, they are typically contained in a contiguous chunk of storage.

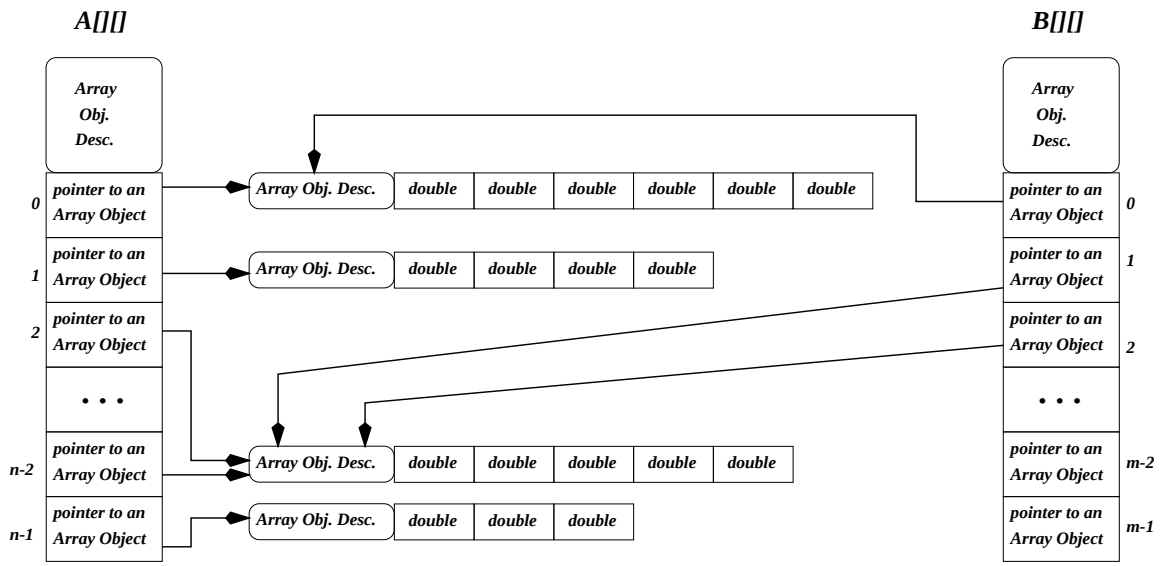


Figure 1: Example of a two-dimensional Java array.

The advantages of this layout for an array are that very general structures can be created, and that the necessary information to perform bounds checking is always available. As seen in Figure 1, it is not required that every row have the same length. Figure 1 is an example of a *ragged* array. It is not even necessary for every row to be unique – rows 2 and $n-2$ of array A point to the same row array object, and thus $A[3][4]$ and $A[n-2][4]$ name exactly the same element. Even more generality is possible with arrays of `Objects`. In that case, each element can be an arbitrary Java object, even another array of any rank and type. All this generality comes with a performance price – a price that is too high for numerical programs that do not need the supplied generality.

First, consider array bounds checking. The Java language specification requires that any access to an array element that is not in bounds must throw an exception. The job of checking for out-of-bounds exceptions exacts a significant toll on Java performance. In [22], it is shown that eliminating array bounds checks alone produced speedups of three to four on the IBM POWER and PowerPC processors. On some processors, this overhead is less. The real impact on performance, however, comes from having one or more potential exceptions for every data access. Because Java exceptions are precise, operations cannot be moved past a potential exception. This effectively precludes almost all optimizations traditionally applied to numerical programs [29, 25] – optimizations that are necessary for good performance.

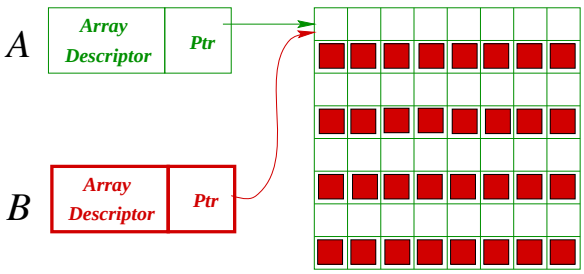
In [22], techniques for creating program regions free of array bounds exceptions are presented. Even with these techniques, the Java array model interferes with good optimization. The major remaining im-

pediment is the ability in Java to alias [25] rows of an array. Aliasing occurs when two or more apparently different variables or references actually refer to the same datum. An example of aliasing is shown in Figure 1, where $A[2]$ and $A[n-2]$ refer to the same row. Moreover, $B[1]$, $B[2]$ and $A[n-2]$ all refer to the same row. The problem with aliasing is that many optimizations rely on moving data reads and writes relative to one another. For this rearrangement to be legal, it is necessary that (i) all writes to a datum be kept in order; (ii) that no write to a datum be moved prior to a read of the same datum; and (iii) that no read from a datum be moved after a write to the same datum. Essential to determining the legality of a transformation is the ability to determine when two operations are to different data and, therefore, independent. Aliasing makes this more difficult because it is no longer possible to say that just because two variables have different names, or because different array elements have different coordinates, they must refer to different memory locations.

The final problem with Java arrays is the cost of accessing an element. Accessing the element $A[3][4]$ requires the following steps: (i) get the reference to the array object A; (ii) determine that A is not a `null` pointer; (iii) determine that 3 is an in-bounds index for A; (iv) get the reference to the array object that is row 3; (v) determine that this object reference is not a `null` pointer; (vi) determine that 4 is an in-bounds index for $A[3]$; and (vii) get the data element $A[3][4]$. In contrast, as we will see, Fortran 90 requires only two steps.

1.2 Fortran 90 arrays

The structure of a Fortran 90 two-dimensional array is shown in Figure 2. It has two major components: (i) a block of dense, contiguous storage that holds the data elements of the array, and (ii) a *descriptor* that contains the address of the block of dense storage, bounds information for each dimension, the size of individual data elements, and other information used to index the array. This indexing information is used to access array elements and implement array sections. (Logically adjacent elements of an array section may not be physically adjacent in storage.) Accessing an element of the array is straightforward: the base address of the storage is extracted from the descriptor, and the subscript, along with the indexing information contained in the descriptor, is used to compute an offset into the data storage.




```
real*8, dimension(8,8) :: A
real*8, dimension(:,:), pointer :: B
B => A(2:8:2,:)
```

Figure 2: The layout of a Fortran 90 array.

Because of the information contained in the descriptor, array sections (see array B in Figure 2, which corresponds to the shaded elements of A) can be implemented efficiently. A *section* of a Fortran 90 array A is a view of A that accesses some subset of the elements of A. The section has the following properties: (i) it (logically) utilizes the same storage as the original array, thus a change of an element value contained in A and the section is reflected in both; (ii) the section has rank less than or equal to the rank of A; (iii) the elements in the section are defined by enumerating element indices along each axis; and (iv) each element enumerated must be a valid, in-bounds element of the original array axis. Let A be an array of shape 8×8. Then $B \Rightarrow A(2:8:2,:)$ associates B with rows $2,4,6,8$ and all columns of the original array A.

The reference $B(3,4)$ accesses row 6 and column 4 of the original array A. This example is illustrated in Figure 2.

The design of Fortran 90 arrays has several advantages. First, because the storage is dense and contiguous, access requires a single pointer dereference and offset computation, regardless of the rank or dimensionality of the array being accessed. Second, the ability to take sections allow a great flexibility in passing portions of arrays to library functions. Third, a rich set of *intrinsic functions* (*i.e.*, built-in operators expressed syntactically as functions) that operate on arrays can be supported. Fourth, most primitive operations in the language (*e.g.*, $+$, $-$, $*$ and $/$) are overloaded and specify the operation applied element-wise over the entire array.

Fortran 90 arrays are not, however, without their faults. *Storage association* exposes to the programmer the fact that data elements are contiguous in memory, and that the next logical element of the array is also adjacent in storage to the previous element. This makes the benefits of bounds checking much more difficult to achieve if programmers do not exercise restraint. (It is not unusual to find Fortran programs that purposedly perform out-of-bounds accesses and rely on storage association to execute correctly.) It also means that optimizations that alter the layout of an array [16, 20, 21, 13] are globally visible. As a consequence, data typically has to be copied to and from the original layout. This makes these optimizations more fragile and less useful, since the overhead of copying the arrays twice must be factored into the cost model for determining if the optimization should be performed.

The lack of true, rectangular, multidimensional arrays in Java has motivated us to develop the Java Array package. This collection of 100% pure Java classes delivers (i) the efficiency of Fortran 90 arrays, (ii) the flexibility of data layout specified by Java arrays, and (iii) the safety and security features provided by the Java requirement for bounds checking.

1.3 Nonprimitive numerical types in Java and Fortran 90

Java does not support complex, decimal and other important numerical types as primitives. It is, of course, possible to define classes that implement these types, and this is what is usually done in Java. There is a large cost associated with this strategy, however. As seen in the MICROSTRIP benchmark results (Section 4, where a detailed analysis of this benchmark is given), applications using a Java `Complex` class (to represent complex numbers) can be 100 times slower than the equivalent applications in Fortran 90. These performance problems can be addressed by using the technique of *semantic expansion* [35]. With semantic expansion, `Complex` objects and operations on those objects are recognized by the compiler, and treated essentially as language primitives (largely as Fortran 90 would treat them). Semantic expansion can improve the performance of Java applications using complex arithmetic to approximately 90% of the performance of corresponding Fortran 90 programs.

Thus, semantic expansion can overcome the performance impact of representing numerical types as objects. What is not overcome by semantic expansion is the difficulty of writing, maintaining, and reading programs that make extensive use of classes to represent numerical types. Because Java does not support *operator overloading*, the $+$ operator cannot, for example, be used to represent the addition of two complex numbers. Instead, the expression $a = b + c$, where a, b, and c are all `Complex`, is coded as `a.assign(b.plus(c))`, which is more cumbersome. Although operator overloading is not a performance issue, it is fundamental for productivity and clarity of numerical codes with nonprimitive arithmetic systems.

1.4 Achieving high performance in numerically intensive Java programs

Achieving high performance in numerically intensive programs has two prerequisites: (i) basic tools provided by the programming language that allow the numerical operations to be performed with a minimum

of overhead, and (ii) skilled programmers able to employ these tools in an intelligent way. The reason why Fortran is the standard of comparison for numerical programming is that its design has always emphasized the ability to specify efficient numerical programs over all else. For Java to be similarly effective does not require violating the spirit of the language. Good numerical performance can be achieved by:

- Defining standard packages and classes, such as the Array package and classes for complex numbers described in this paper, that provide operations and data structures that are suited for important numerical computations.
- Developing optimizing compilers that recognize these standard packages and classes, and then efficiently map them onto the target hardware. Semantic expansion is one of the techniques for achieving this.
- Transforming programs to contain exception-free regions over which aggressive traditional optimizations (*e.g.*, as developed for Fortran) can be performed.
- Allowing modes of operations that map the semantics of well defined floating-point operations onto hardware. Examples include the *fused multiply-add* (`fma`) instruction of the IBM POWER and PowerPC processors, or the 80-bit floating-point registers of the Intel IA32 family of processors.

The remainder of this paper discusses our efforts in these areas, and the substantial performance gains achieved. Section 2 describes the details of the Java Array package, which supports Fortran 90-style programming in Java. We follow with two case studies that illustrate the benefits of programming with the Array package. Section 3 is a case study of a data mining application and Section 4 is a case study of an electromagnetics application. Section 5 summarizes performance results from a large set of numerically intensive benchmarks. It shows that Java can be performance competitive with the best Fortran implementations. Finally, Section 6 discusses some related work and Section 7 presents our conclusions, discusses the impact of our work, and elaborates on some future research.

The Array package is freely available from alphaWorks, at `http://www.alphaworks.ibm.com/tech/ninja`. Our project web-page, with additional information on the Array package and our research in general, is located at `http://www.research.ibm.com/ninja`.

2 The Java Array package

The goal of the Array package is to overcome the limitations inherent in Java arrays in terms of performance and ease of use, as compared to array implementations in other languages. A major design principle of the Array package is to define a set of classes implementing Fortran 90-like multidimensional arrays that are highly optimizable, amenable to compiler analysis techniques, and that contain the functionality necessary for numerically intensive computing. We have modeled the Java Array package along features of the Fortran 90 language and the A++/P++ libraries [26, 34]. Not surprisingly, the reader will also find similarities with APL, since that language undoubtedly influenced the design of Fortran 90. However, APL is a much more dynamic language, with strong support for polymorphism of data and functions. The design of the Java Array package, particularly as described in this paper, followed the more static nature of Fortran 90. It is important to emphasize that the Array package is written entirely in Java. Its use does not require any code preprocessing steps, just the appropriate `import` statements.

A word about notation: From now on, when we use the term *Array* (with a capital *A*) we refer to the constructs provided by the Array package. The standard arrays in the Java language will be referred to as *Java arrays* (with a small *a*).

The functionality provided by the Array package is implemented by a set of basic methods and operations that are highly optimizable. The class hierarchy inside the Array package was defined to enable

aggressive compiler optimization. As an example, we make extensive use of *final* classes, since most Java compilers (and, in particular, *javac*) are able to apply method inlining to methods of final classes. In the most commonly used methods of the Array package (*i.e.*, simple element-wise operations) the cost of a method call dominates the cost of the computation done by the method. Therefore, inlining those methods is essential for good performance. Even in compilers capable of inlining nonfinal methods, the overhead of analysis is reduced by having the methods declared as final.

In contrast to Java arrays discussed in the previous section, Arrays defined by the Array package have properties (*e.g.*, a nonragged rectangular shape and constant bounds over their lifetime) that are easy to detect and use by an optimizing compiler. Thus, for problems that map well to rectangular multidimensional structures, the Array package provides both functionality and performance and is the obvious choice over Java arrays.

Arrays are n-dimensional rectangular collections of elements. An Array is characterized by its *rank* (number of dimensions or axes), its elemental data *type* (all elements of an array are of the same type), and its *shape* (the extents along its axes). Rank, type, and shape are immutable properties of an Array. All Array classes are named using the following scheme: <*type*>`Array`<*rank*>`D`. As an example, `floatArray3D` is a three dimensional array of single precision floating point numbers. In the current version of the Array package, *type* can be any of the Java primitive types (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double`), `Complex` (for complex numbers), or `Object` for general objects. The currently supported values for *rank* are 0, 1, 2, and 3.

All operations (methods and constructors) of the Array package have, in the absence of JVM failure, transactional behavior. That is, there are only two possible outcomes of an array operation: (i) the operation completes without an exception being thrown, or (ii) an exception is thrown before any data is changed. If a JVM failure occurs (*e.g.*, it runs out of stack space) during the execution of an Array package operation, then the program enters an unrecoverable error state, and the results of the partial operation are observable only by post-mortem tools. Array operations also implement *array semantics*: In evaluating an expression (*e.g.*, $A = A * B + C$), all data is first fetched (*i.e.*, A, B, and C are read from memory), the computation is performed (*i.e.*, $A * B + C$ is evaluated), and only then the results is stored (*i.e.*, A is modified). Of course, a high-performance system should implement the array semantics while trying to avoid unnecessary copying.

2.1 Class hierarchy

All Array classes are derived from a base abstract class `Array`, which defines the common methods for all types of Arrays. Arrays are further subdivided according to their (elemental) type and rank. We also define the `Index` and `Range` helper classes for indexing and sectioning an Array (see Section 2.3 for more details). Figure 3 shows the class hierarchy of the Array package.

Various exception classes are also provided by the Array package for reporting errors that occur during array operations. They have descriptive names that make the cause of the exception clear to the user. All exception classes are derived from the `RuntimeException` class. In Java, exceptions derived from the `RuntimeException` class are not required to be caught by user code. Hence, it is not an error for a piece of code to invoke an Array package method but not declare (using a `throws` clause) or catch an exception that may be thrown by that method. In current numerical algorithms, conditions that would raise an exception typically result from programming bugs. The expectation is that the programmer will fix the bug rather than add exception-handling code to handle it at run-time. Nevertheless, if the programmer would like to catch the exception, it can still be done, and so no limitations have been placed on exception handling.

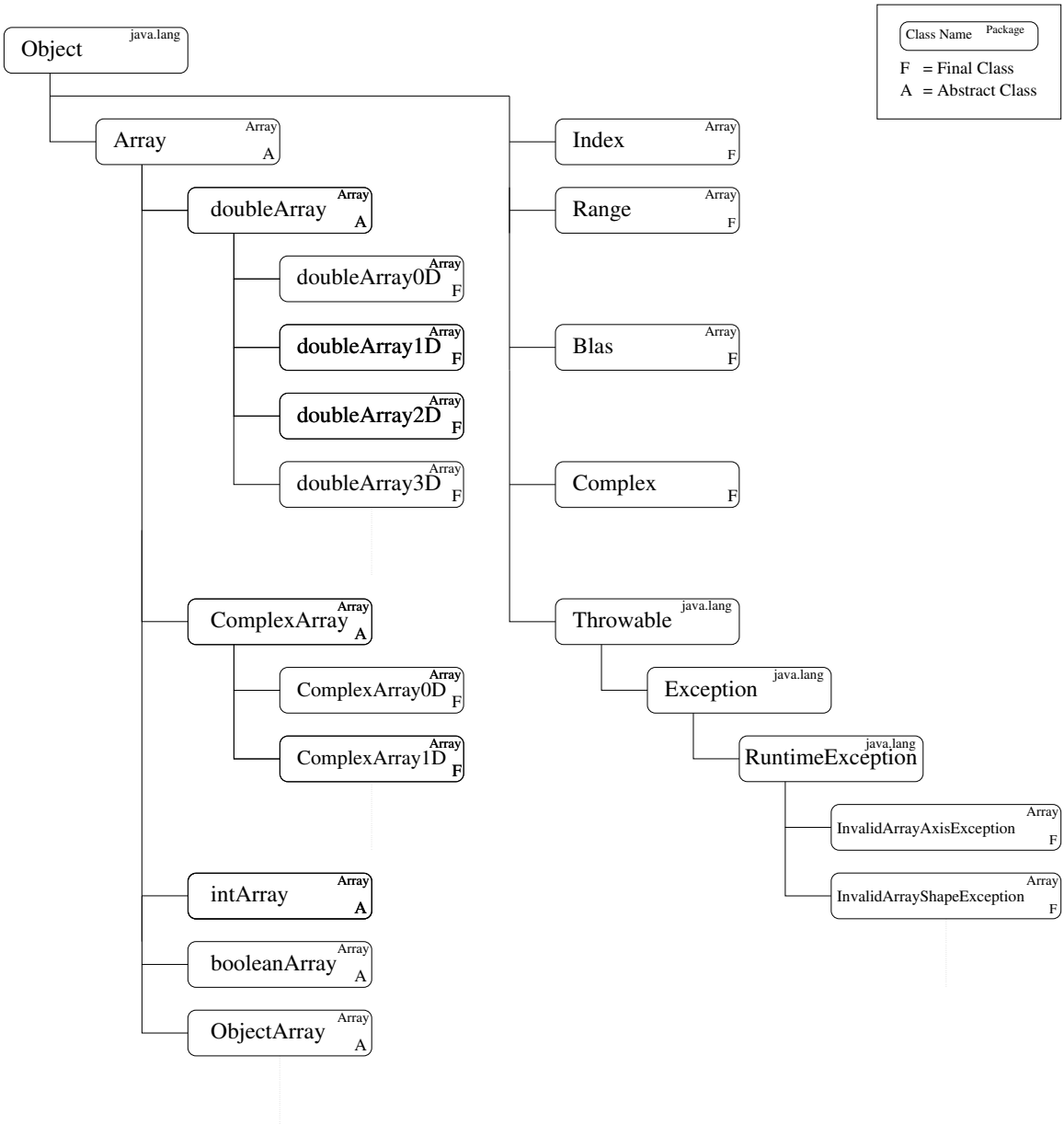


Figure 3: Simplified Class Hierarchy Chart

2.2 Array constructors

The shape of an array is specified at its creation through a constructor, and it is immutable. The Array package defines three constructors for each Array class, illustrated in Figure 4. The first constructor shown builds a new Array whose shape is specified by the parameters. The second constructor shown builds a new Array that is a copy of the Array input parameter. The new Array has its own storage and shares no data with the input Array. Finally, the third constructor shown builds a new Array (with its own storage) with elements copied from a conventional Java array. If the Java array is multidimensional, it must be rectangular.

```
// Simple 3 x 3 array of integers
intArray2D A = new intArray2D(3,3);

// This new array has a copy of the data in A,
// and the same rank and size.
intArray2D A2 = new intArray2D(A);

// Java array of bytes
byte[] buffer = { 0x1A, 0x20, 0xCA, 0xFE, 0x01 };

// New byte array with the same data
byteArray1D B = new byteArray1D(buffer);
```

Figure 4: Simple Array construction operations.

2.3 Indexing elements of an array

Elements of an array are identified by their indices (coordinates) along each axis. Let a k-dimensional array A of elemental type T have extent n_j along its j-th axis, $j = 0, \ldots, k-1$. Then, a valid index i_j along the j-th axis must be greater than or equal to zero 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.

The Array package allows an axis of an Array to be indexed by either an integer, a `Range` or an `Index`. Indexing with an integer specifies a single element along an axis. Indexing with a `Range` object is like addressing using triplets in Fortran: a first element, a last element and an optional stride (which defaults to one) specify a regular pattern of elements. An `Index` object is a list of numbers which is then used by the Array package method to select the enumerated elements along the given axis. `Index` objects are convenient for scatter/gather operations and for operations on sparse data in general. When indexing with an `Index` or `Range` object, all the indices must be valid (*i.e.*, within bounds), or an `ArrayIndexOutOfBoundsException` is thrown.

Indexing operations are used in *accessor methods* that read/write data from/to an array, and in *sectioning methods* that create views of an array. Accessor methods support all possible combinations of integer, `Range`, and `Index` indices. That is, the subscript along any given dimension can be either an integer, a `Range` object, or an `Index` object. Sectioning methods only support combinations of integer and `Range` indices. Supporting sections defined with `Index` objects would require a complicated and slow descriptor for the array, and impair efficient access to array elements in the common cases. Array accessors can be implemented efficiently even with `Index` objects.

2.4 Defined methods

The Array package defines four groups of methods that are always present in any Array package class: Array Operation, Array Manipulation, Array Accessor, and Array Inquiry. A group of methods is just a collection of methods with similar functionality. The complexity of operations is, in most cases, similar for methods of a group.

2.4.1 Array operations

Array operations are scalar operations applied element-wise to a whole Array. The methods in this category include assignment, arithmetic and arithmetic-assign operations (analogous, for example, to += and *=

operators in Java), comparison operations, and logic operations (where appropriate for the elemental data type). These operations are further subdivided as *Scalar-Array* and *Array-Array* operations. Scalar-Array operations apply the operation to a scalar and each element of the Array. Array-Array operations apply the operation to the equivalently (scalar) elements of two Arrays. If both Arrays in an Array-Array operation do not have the same shape, a `NonconformingArrayException` is thrown. These operations are highly optimizable and parallel in nature.

The semantics of these operations imply that the operations are performed on Arrays as they would be performed on scalars. That is, the result is computed before assigning into the target array. Therefore, execution of an Array operation logically follows the sequence: (i) all operands are fetched from memory, (ii) all computation is performed, and (iii) the result is stored to memory. The ordering between steps (ii) and (iii) implies that, in general, the result of the operation must be first stored in a temporary Array and then copied to the final target. The result of the operation can be written directly to the target Array (thus eliminating the need for the temporary Array and the copy operation) if doing so does not overwrite any memory location before its use as an operand. The Array package uses a form of dynamic *dependence analysis* [4] to determine when this condition holds. It then bypasses the unnecessary temporary Array and copy.

In Figure 5 we compare some elementary Array operations described using a math notation and the notation defined by the Array package. When there is more than one Array package version for the expression, the first expression is the most straightforward and intuitive form. The alternative forms for an operation lead to better performance, as they avoid creating (at least some) intermediate objects. (The forms are semantically equivalent, if the whole operation is legal.) It is unrealistic to expect users of the Array package to write all Array expressions in the less intuitive but more efficient form. An intelligent compiler should be able to automatically translate the Array expressions to the best performing version.

Math notation	Java Array package notation
$A \leftarrow B$	`A.assign(B);`
$A \leftarrow A + B$	`A.assign(A.plus(B));` `A.plusAssign(B);`

$A \leftarrow B + C \times D$	`A.assign(B.plus(C.times(D)));`
	`A.assign(C).timesAssign(D).plusAssign(B);`
	`C.times(D,A).plusAssign(B);`

$A \leftarrow 0.25 \times (B + C + D + E)$	`A.assign(B.plus(C).plus(D).plus(E).times(0.25));`
	`A.assign(B).plusAssign(C).plusAssign(D).plusAssign(E).timesAssign(0.25);`
	`B.plus(C,A).plusAssign(D).plusAssign(E).timesAssign(0.25);`

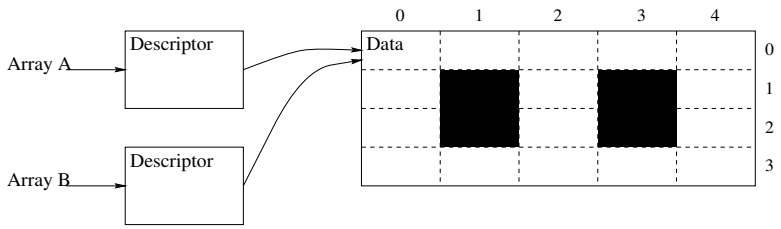
Figure 5: Array operations in Array package and math notations.

2.4.2 Array manipulation

Methods in this group include `section`, `permuteAxis` and `reshape`. These methods operate on the array as a whole, creating a new view of the data and returning a new Array object expressing this view. A new Array object is needed because an Array object always has constant properties, and the new Array view may have different basic properties than the original array, (*e.g.*, a different rank or shape). This new Array object shares its data with the old object when possible to avoid the overhead of data copying.

Sectioning an Array means obtaining a new view to some elements of a given Array. A section is a new Array object that shares its data with the original Array, as a Fortran 90 sections share its data with the base array. Since data is shared, only a descriptor needs to be created, resulting in a fast sectioning operation.

Figure 6 illustrates how a sectioning operation works. For two-dimensional Arrays, the first index applies to rows and the second index applies to columns. This example creates a section `B` corresponding to rows 1 and 2 and columns 1 and 3 of `A`. Because sections are first class Array objects, operating on an Array section is indistinguishable from operating on any other Array object.




```
longArray2D A = new longArray2D(4,5);

Range R0 = new Range(1,2); // from 1 to 2 in steps of 1
Range R1 = new Range(1,3,2); // from 1 to 3 in steps of 2

longArray2D B = a.section(R0,R1); // now A and B share elements
```

Figure 6: Example of an array section. The operation creates a 2×2 Array `B` associated with rows 1 and 2 and columns 1 and 3 of Array `B`.

The `permuteAxis` method implements generic axis interchange. Although `permuteAxis(1,0)` returns a transposed two-dimensional array, `permuteAxis` is more general than a simple transpose because any axis can be replaced by any other axis, as long as all axes that exist in the original Array also exist in the permuted Array. If this condition is not met, the method call is illegal and an exception is thrown. For example, `B = A. permuteAxis(2,1,0)` exchanges axes 0 and 2 of a three-dimensional Array `A`. That is, `B(i,j,k)` = `A(k,j,i)`. The expression `B = A.permuteAxis(1,1,0)` is illegal, and throws a runtime exception, since axis 1 appears twice and axis 2 is missing. Permuting the axis of a given array is a fast operation that only creates a new descriptor. No data copy is needed as the data is shared with the original array. If the user wants data copy to take place, the copy constructor always guarantees that the data is actually copied, as in the example in Figure 7.

```
// Create a 5x4 Array A of longs
longArray2D A = new longArray2D(5,4);

// Create an Array B that is a transpose of A. This is just a
// descriptor operation. The internal data layout of
// A is not changed. The data in A is shared with B.
longArray2D B = A.permuteAxis(1,0);

// Use the copy constructor to guarantees that the data is copied.
// C has a copy of the data in A, organized in a different
// layout.
longArray2D C = new longArray2D(A.permuteAxis(1,0));
```

Figure 7: Permuting axis, with and without data copy.

The `reshape` method returns a new Array with the same data as the old Array but with a different

shape. Every element in the old Array is mapped to an element of the new Array. Therefore, the new Array has the same number of elements as the old Array. The algorithm used to map Array elements is the following. The data is first (logically) linearized by visiting all axes in order, and all elements within an axis in index order. The logically linearized elements are then copied into an Array with the shape specified in the `reshape` call by copying them into the new Array. Figure 8 gives examples of `reshape` method calls. The reshape method throws an exception if the specified shape does not have the right number of elements.

```
short[][] JavaArray = { {1,2},{3,4},{5,6} };

shortArray2D A = new shortArray2D(JavaArray);

shortArray2D B = A.reshape(2,3);
// B is now  { {1,2,3},{4,5,6}}

shortArray2D C = B.reshape(6);
// C is now { 1,2,3,4,5,6 }

shortArray2D D = C.reshape(3,2);
// D is equal to A, the reshape never changes
// the defined total order among elements
```

Figure 8: Reshaping an array.

Because Array objects have immutable shape, reshaping an Array implies creating a new Array object. A copy of data is needed because our descriptor, which is designed to be fast for common operations, is not rich enough to express the effect of a reshape. We choose not to have a complex descriptor to support fast reshaping for two reasons: First, the complex descriptor would slow down most other methods. Second, a common purpose of an Array reshape operation is to reorganize the data in order to obtain faster access, and so data copying is a desirable consequence of the reshape operation.

2.4.3 Array accessor methods

The methods of this group are `get`, `set` and `toJava`. These are methods that allow the user to extract and change the array data. Figure 9 contain examples of uses of the methods in this group.

The `get` and `set` methods are straightforward. The programmer is provided indexing methods (see Section 2.3) to select one or more elements. A `get` returns the specified elements, and a `set` updates the specified elements.

The method `toJava` allows a user to extract a Java array from an Array package Array. This feature, together with constructors that convert Java arrays into Array package Arrays, allows pieces of code that work with Java arrays to co-exist with (*i.e.*, exchange data with) pieces of code that use the Array package.

As the basic purpose of this group of methods is to move data in and out of Arrays, data copy is always performed. The common operations are fast – for example, extracting a single element from an array. Extracting an entire subarray (*e.g.*, a row of a two-dimensional array) using a single Array class operation is typically faster than extracting one element at a time. A compiler is more likely to optimize the operations within a method with known semantics than across methods contained in code whose semantics must be inferred.

```
doubleArray2D A = new doubleArray2D(5,5);

// Simple element wise operations
double firstElement = A.get(0,0);
A.set(4,4,firstElement);

// The row1 Object has a copy of the data in the first row of
// array A. It is not a section of Array A because it does
// not share data with A.
doubleArray1D row1 = A.get(1,new Range(0,A.last(1)));

// The second row of array A now equals the first (values, not aliased)
A.set(2,new Range(0,A.last(1)),row1);

// Extract elements of A into a double[][] structure
double[][] AJava = A.toJava();
```

Figure 9: Simple Array accessor operations.

2.4.4 Array inquiry methods

This group contains methods `last`, `size`, `rank` and `shape`. They are fast, descriptor-only operations which return information about the constant properties of the Array.

The method `last` (used as an example in Figure 9) returns the last valid index along a given axis. The method `size`, when invoked without arguments, returns the number of elements in the Array; if invoked with an `int` argument it returns the number of elements along this given axis. The method `rank` returns the rank of the Array (*i.e.*, the number of axes or dimensionality of the Array). Finally, the `shape` method returns a Java array of integers whose elements are the sizes of the Array along each individual axis.

2.4.5 Comparing array operations in Fortran 90 and with the Java Array package

As previously mentioned, we have patterned our design of the Array package along the lines of the Fortran 90 language. Figure 10 provides a side by side comparison of equivalent Fortran 90 and Java Array package constructs. Readers familiar with Fortran 90 will notice an almost one-to-one correspondence between the two approaches. This correspondence has both performance as well as usability implications. From a performance perspective, optimization techniques developed for Fortran 90 can be applied to the Array package. From a usability perspective, programmers used to Fortran 90 features will find corresponding functionality in the Array package.

2.5 Internal optimizations

To provide good performance, the supplied Array operations avoid data copying – a costly operation – as much as possible. For example, consider the code in Figure 11. A naive implementation would implement the `plusAssign` method as two operations, with a new object being created to store the sum of the two arrays. Thus, `row1.plusAssign(row2)` would be logically implemented as:

```
new_object = row1 + row2
      row1 = new_object
```

Fortran 90	Java Array package

`INTEGER :: m, m, p`
`ALLOCATABLE, REAL*8 :: a(:,:),b(:,:),c(:,:)`
`ALLOCATE(a(0:m-1,0:p-1))`
`ALLOCATE(b(0:p-1,0:n-1))`
`ALLOCATE(c(0:m-1,0:n-1))`
 	
`c(i,j) = c(i,j)+a(i,k)*b(k,j)`
 	
`c(0:10,0:p-1) = b(10:20,0:p-1)`
 	
`INTEGER, DIMENSION(10) :: i, j`
`REAL*8, DIMENSION(0:9,0:19) :: x, y`
`i = (/ 2, 3, 5, 7 /)`
`j = (/ 1, 2, 4, 8 /)`
`x(i,0:19) = y(j,0:19)`
	`int m, n, p;`
	`doubleArray2D a, b, c;`
	`a = new doubleArray2D(m,p);`
	`b = new doubleArray2D(p,n);`
	`c = new doubleArray2D(m,n);`
	
	`c.set(i,j,c.get(i,j)+a.get(i,k)*b.get(k,j));`
	
	`Range pm1 = new Range(0,p-1);`
	`Range R010 = new Range(0,10);`
	`Range R1020 = new Range(10,20);`
	`c.section(R010,pm1).assign(b.section(R1020,pm1));`
	
	`Index i, j;`
	`doubleArray2D x, y;`
	`i = new Index(...);`
	`j = new Index(...);`
	`x = new doubleArray2D(10,20);`
	`y = new doubleArray2D(10,10);`
	`x.set(i,new Range(0,19),y.get(j,new Range(0,19));`

Figure 10: Side by side comparison of Fortran 90 and Java Array package constructs.

The contents of the new object would then be copied back to the first object (`row1`). The unnecessary copying degrades performance. Our implementation avoids this extra copying by checking if the two objects represent different Arrays or nonoverlapping sections of the same Array. It performs a run-time data dependence test to verify if storing directly to the target Array will preserve array semantics.

```
floatArray2D x;
floatArray1D row1 = x.section(1,new Range(0,x.last(1)));
floatArray1D row2 = x.section(2,new Range(0,x.last(1)));

row1.plusAssign(row2);
```

Figure 11: Example of array operation internally optimized through dynamic data dependence analysis.

2.6 BLAS routines

The Basic Linear Algebra Subprograms (BLAS) [12] are the building blocks of efficient linear algebra codes. BLAS implements a variety of elementary vector-vector (level 1), matrix-vector (level 2), and matrix-matrix (level 3) operations. We have designed a BLAS class as part of the Array package. It provides the functionality of the BLAS operations for the multidimensional Arrays in that package. Note that this is a 100% Java implementation of BLAS, and not an interface to already existing native libraries. Therefore, we have a completely portable parallel implementation of BLAS that works well with the Array package. The code in the BLAS class is optimized to take advantage of the internals of the Array package implementation. Application code making use of these methods will have access to tuned BLAS routines with no programming effort.

3 Case study: data mining

In this section, we present a case study, using a data mining application, that illustrates some of the pitfalls and opportunities in using Java to develop a numerically intensive application. Data mining [32] is the process of discovering useful and previously unknown information in very large data bases, with applications that span scientific computing (*e.g.*, image processing) and commercial computing (*e.g.*, fraud detection). Although typical data mining applications access large volumes of data from either flat files or relational data bases, such applications often involve a significant amount of computation above the data access layer. High-performance data mining applications, therefore, require both fast data access and fast computation.

While Java is an attractive vehicle for developing and deploying data mining applications across different computing platforms, a major concern has been its performance. We show that this concern is well-founded, but can be dealt with by using well-designed class libraries, such as those from the Array package described in Section 2. In particular, we show that a plain Java implementation suffers from the following major performance problems, both of which are exacerbated by its lack of support for true multidimensional arrays:

- *Bounds checking and null pointer checks on array accesses*: The overhead of checking each array access for a `null` pointer or out-of-bounds access violation takes its toll on performance. To make matters worse, a single reference to a d-dimensional array in Java translates into d references to 1-D arrays, requiring d `null` pointer checks and d out-of-bounds checks. The algorithm to optimize these checks away in a compiler by creating *safe* regions [24] is significantly simpler for a true multidimensional array because its shape is constant along all axes and there is no need to copy arrays to a thread's *working memory* for thread-safety of the transformation.

- *Poor data locality and instruction scheduling*: Loop transformations, such as interchange, tiling and outer loop unrolling, typically used by optimizing compilers to improve data locality and instruction scheduling, cannot be readily applied to loops in which an exception may be thrown, because of the need to support *precise* exceptions in Java. Furthermore, different arrays constituting a multidimensional array in Java are not necessarily stored together in a single contiguous block of memory, leading to inherently poor data locality.

Using the Array package allows us to deal with these problems quite effectively. The methods in our implementation of this package perform all checks for possible exceptions before the main part of the computation. As a result, the main computation can be optimized in the same manner as equivalent Fortran or C++ code, while still remaining strictly Java-compliant. The main loops in the methods of the BLAS class are transformed manually using well-known techniques [12] to get better uniprocessor performance though improved data locality, instruction scheduling, and register utilization. Furthermore, the Array package internally uses contiguous memory for the entire data of a multidimensional Array, leading to improved data locality and more efficient indexing of elements.

3.1 The data mining computation

The specific data mining problem considered here involves recommending new products to customers based on previous spending behavior. As input, we have available both detailed purchase data for the customer set, as well as a product taxonomy which generates assignments of each product to spending categories. In the specific application discussed here, we utilize approximately two thousand such categories. Recommendations are drawn from a subset of eligible products. For each customer, a personalized recommendation list is generated by re-sorting this list of eligible products according to a customer-specific score computed for each product. In some commercial situations, there may be an incentive to preferentially recommend some products over others due to overstocks, supplier incentives, or increased profit margins. This information

can be introduced into the recommendation strategy via product-dependent scaling factors, with magnitudes reflecting the relative priority of recommendation. We now give a more precise description of the scoring algorithm.

Let there be m products eligible for recommendation, p customers, and n spending categories. Each product i has associated with it an *affinity vector* A_i, where A_{ik} is the affinity of product i with spending category k. These affinities can be interpreted as the perceived appeal of this product to customers with participation in this spending category. They are determined by pre-computing associations [2] at the level of spending categories. The collection of affinity vectors from all products forms the $m \times n$ affinity matrix A. This matrix is sparse and organized so that products with the same nonzero patterns in their affinity vectors are adjacent. This is shown in Figure 12. Because of this organization, matrix A can be represented by a set of γ groups of products. Each group i is defined by f_i and l_i, the indices of the first and last products in the group, respectively. All products with the same nonzero pattern belong to the same group. This nonzero pattern for the products in group i is represented by an index vector I_i, where I_{ik} is the position of the k-th nonzero entry for that group. Let $\text{len}(I_i)$ be the length of index vector I_i. Also, let $g_i = l_i - f_i + 1$ be the number of products in group i. Then the affinity matrix G_i for group i is the $g_i \times \text{len}(I_i)$ dense matrix with only the nonzero entries or rows f_i to l_i of A. This representation of A is shown in Figure 12.

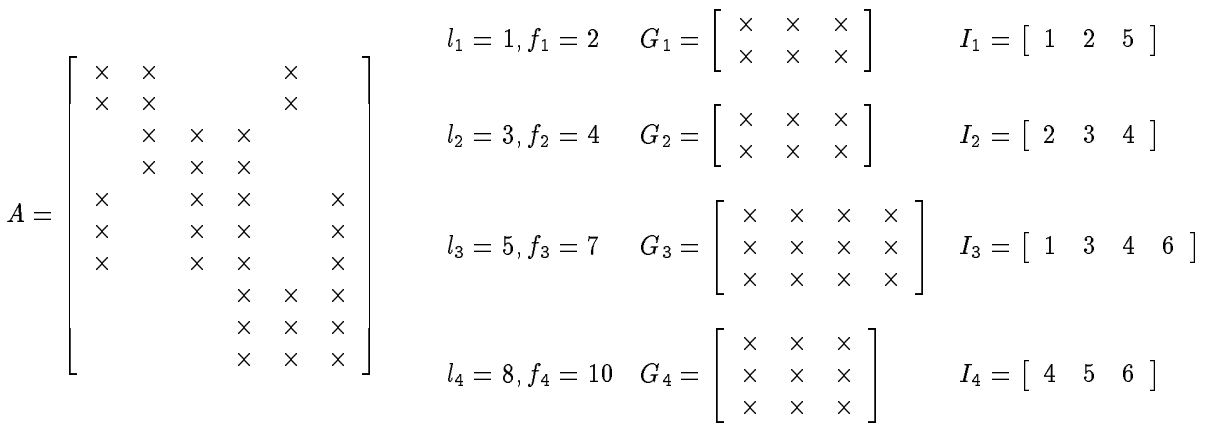


Figure 12: Structure and representation of the affinity matrix A.

Each customer j has an associated *spending vector* S_j, where S_{jk} is the normalized spending of customer j in category k. The collection of spending vectors of all customers forms the $p \times n$ spending matrix S. This matrix is also sparse, but customers are not organized in any particular order. (The affinity matrix is relatively static and can be built once, whereas the spending matrix may change between two executions of the scoring algorithm.) We also define a $p \times n$ normalizing matrix N, which has the same nonzero pattern as S, but with a 1 wherever S has a nonzero:

$$N_{jk} = \begin{cases} 1 & \text{if } S_{jk} \neq 0 \\ 0 & \text{if } S_{jk} = 0 \end{cases} . \tag{1}$$

The structure of matrices S and N are shown in Figure 13. The nonzero pattern of row j of matrix S is represented by an index vector J_j, where J_{jk} is the position of the k-th nonzero entry for that row (customer). The actual spending values are represented by a dense vector σ_j, of the same length as J_j, which has only the nonzero entries for row j of S. This representation of S is shown in Figure 13. Matrix N does not have to be represented explicitly.

$$S = \begin{bmatrix} \times & & \times & & \times & \\ \times & \times & & & \times & \\ \times & \times & \times & \times & & \\ & \times & \times & \times & & \\ \times & \times & & \times & & \times \\ \times & & \times & \times & & \times \\ & & \times & \times & & \times \\ \times & & & \times & \times & \end{bmatrix} \quad N = \begin{bmatrix} 1 & & 1 & & 1 & \\ 1 & 1 & & & 1 & \\ 1 & 1 & 1 & 1 & & \\ & 1 & 1 & 1 & & \\ 1 & 1 & & 1 & & 1 \\ 1 & & 1 & 1 & & 1 \\ & & 1 & 1 & & 1 \\ 1 & & & 1 & 1 & \end{bmatrix}$$

$$\sigma_1 = [\ \times\ \times\ \times\ ] \qquad J_1 = [\ 1\ 3\ 5\ ]$$
$$\sigma_2 = [\ \times\ \times\ \times\ ] \qquad J_2 = [\ 1\ 2\ 5\ ]$$
$$\sigma_3 = [\ \times\ \times\ \times\ \times\ ] \qquad J_3 = [\ 1\ 2\ 3\ 4\ ]$$
$$\sigma_4 = [\ \times\ \times\ \times\ ] \qquad J_4 = [\ 2\ 3\ 4\ ]$$
$$\sigma_5 = [\ \times\ \times\ \times\ \times\ ] \qquad J_5 = [\ 1\ 2\ 4\ 6\ ]$$
$$\sigma_6 = [\ \times\ \times\ \times\ \times\ ] \qquad J_6 = [\ 1\ 3\ 4\ 6\ ]$$
$$\sigma_7 = [\ \times\ \times\ \times\ ] \qquad J_7 = [\ 3\ 4\ 6\ ]$$
$$\sigma_8 = [\ \times\ \times\ \times\ ] \qquad J_8 = [\ 1\ 4\ 5\ ]$$

Figure 13: Structures of the spending matrix S and normalizing matrix N, and representation of matrix S.

Let ρ_i be the priority scaling factor for product i as discussed earlier in this section. The *score* Φ_{ij} of customer j against product i is computed as:

$$\Phi_{ij} = \rho_i \frac{\sum_{k=1}^{n} A_{ik} S_{jk}}{\sum_{k=1}^{n} A_{ik} N_{jk}}. \tag{2}$$

Note that $\sum_{k=1}^{n} A_{ik} S_{jk}$ is the dot-product of row i of A by column j of S^T. Similarly, $\sum_{k=1}^{n} A_{ik} N_{jk}$ is the dot-product of of row i of A by column j of N^T. Let $\Lambda_p(\rho_1, \rho_2, \ldots, \rho_m)$ denote an $m \times p$ matrix Λ with $\Lambda_{ij} = \rho_i, 1 \le i \le m, 1 \le j \le p$. Then, Equation (2) can be rewritten as a matrix operation:

$$\Phi = \Lambda_p(\rho_1, \rho_2, \ldots, \rho_m) * (A \times S^T) \div (A \times N^T) \tag{3}$$

where $\times$ denotes matrix multiplication, $\div$ denotes two-dimensional array (or element-by-element) division, and $*$ denotes two-dimensional array (element-by-element) multiplication.

Using the representations of A and S discussed previously, the scores for a customer j against all products in a product group i can be computed in the following way. First, two vectors ε and η of size n are initialized to 0:

$$\begin{aligned} \varepsilon[1:n] &= 0, \\ \eta[1:n] &= 0. \end{aligned} \tag{4}$$

Then, the compressed vector σ_j is expanded by assigning its values to the elements of ε indexed by J_j. (This is a scatter operation.) Similarly, an expanded representation of the j-th row of N is stored into η:

$$\begin{aligned} \varepsilon[J_j] &= \sigma_j, \\ \eta[J_j] &= 1. \end{aligned} \tag{5}$$

Now, the scores of customer j against products f_i to l_i can be computed by:

$$\Phi[f_i : l_i, j] = \Lambda_1(\rho_{f_i}, \rho_{f_i+1}, \ldots, \rho_{l_i}) * (G_i \times \varepsilon[I_i]) \div (G_i \times \eta[I_i]) \tag{6}$$

where $\times$ now denotes matrix-vector multiplication, and $\div$ and $*$ denote one-dimensional array (element-by-element) division and multiplication respectively. Note that we have two gather operations, in $\varepsilon[I_i]$ and $\eta[I_i]$. If we group several customers together, forming a block from customer j_1 to customer j_2, the operations in Equation (6) become dense matrix multiplications, array division, and array multiplication:

$$\Phi[f_i : l_i, j_1 : j_2] = \Lambda_{j_2 - j_1 + 1}(\rho_{f_i}, \rho_{f_i+1}, \ldots, \rho_{l_i}) * (G_i \times \varepsilon[I_i, j_1 : j_2]) \div (G_i \times \eta[I_i, j_1 : j_2]). \tag{7}$$

Matrix multiplication can be implemented more efficiently than matrix-vector multiplication. This is particularly true on cache-based RISC microprocessors, since matrix multiplication can be organized through blocking to better exploit the memory hierarchy.

3.2 The data mining implementations

This section discusses five different implementations of the scoring algorithm described in Section 3.1. The first version is a *plain* Java implementation of the scoring algorithm. That is, it does not make use of the Array package described in Section 2. Instead, it uses standard Java arrays. The second implementation is our *enhanced* Java code, which implements the scoring using classes from the Array package. Then we describe our support for parallelism in the Array package, that allows us to trivially obtain an *implicitly* parallel version of the application from the enhanced Java code. We also derive an *explicitly* parallel version of the application, which explicitly uses the multithreading facilities in Java together with the sequential version of the Array package. Finally, we describe a Fortran 90 version of the scoring algorithm. The Fortran 90 version serves as a reference: it helps us evaluate how well Java is doing relative to a language known for high performance.

The plain Java implementation: In the plain Java implementation of the scoring algorithm, each G_i matrix is represented as a `double[][] G` and each I_i index vector is represented as an `int[] I`. The compressed spending vector σ_j for customer j is expanded into vectors ε and η according to Equation (5), with these vectors represented by a `double[] E` and `double[] N`, respectively. Scores for customer j against the products in group i are computed according to Equation (6) by the code shown in Figure 14. In that code, `m` and `n` are the number of rows and columns of `G`, respectively. The index f_i of the first product in group i is represented by `f[i]` and the scaling factor ρ_q for product q is represented by `rho[q]`. Results are stored in a `double[][] Phi` structure, which represents Φ.

```
for (int l=0; l<m; l++) {

    double score = 0.0;
    for (int k=0; k<n; k++) {
        score += G[l][k]*E[I[k]];
    }

    double norm = 0.0;
    for (int k=0; k<n; k++) {
        norm += G[l][k]*N[I[k]];
    }

    Phi[f[i]+l][j] = rho[f[i]+l]*score/norm;
}
```

Figure 14: Main computations in the plain Java implementation. It compute $\Phi[f_i : l_i, j]$.

The enhanced Java implementation: In this implementation, each G_i matrix is represented by an object `G` of class `doubleArray2D`, and each I_i by an `Index` object `I`. Scoring is computed according to Equation (7), which operates on a block of customers, from customer j_1 to customer j_2. In that case, ε and η are matrices, also represented by `doubleArray2D` objects `E` and `N`, respectively. Each column of matrix ε is obtained by expanding the compressed vector σ_j for customer j ($j_1 \leq j \leq j_2$). The matrix η is obtained similarly. We operate on one block of customers at a time to conserve memory: we only have to keep expanded spending vectors for the customers in the block. The matrix Λ is stored as a `doubleArray2D` object `Lambda`, with all the elements of row k set to ρ_k. The computation of $\Phi[f_i : l_i, j_1 : j_2]$ is performed by the code of Figure 15. First, an array section `results` is created to represent the region of Φ being

computed. That region is initialized with the matrix product $G_i \times \varepsilon$, after extracting the relevant rows of the ε matrix according to the I index. The result of $G_i \times \eta$ is similarly stored in an auxiliary array `normal`. Matrix products are computed by the `dgemm` method in our BLAS class. We complete the computation of $\Phi[f_i : l_i, j_1 : j_2]$ with an array multiplication by Λ and an array division by the result of $G_i \times \eta$. The `timesAssign` and `divideAssign` methods implement the operators `*=` and `/=` for arrays.

```
/*
 * Extract the section of the Phi matrix corresponding
 * to group i and customers j1 to j2.
 */
results = Phi.section(new Range(f[i],l[i]),new Range(j1,j2));

/*
 * Compute the compressed matrix of customer spending
 * in "compressed". Multiply the group affinity matrix
 * by it, producing the unnormalized result.
 */
compressed = E.get(I,new Range(j1,j2));
Blas.dgemm(NoTranspose,NoTranspose,1.0,G,compressed,0.0,results);

/*
 * Compute the compressed matrix of "1"s.
 * Multiply the group weights matrix by it,
 * producing the normalization matrix.
 */
compressed = N.get(I,new Range(j1,j2));
Blas.dgemm(NoTranspose,NoTranspose,1.0,G,compressed,0.0,normal);

/*
 * Multiplying the unnormalized results by the scaling
 * matrix Lambda and then dividing by the normalization
 * matrix produces the final, normalized results.
 */
results.timesAssign(Lambda).divideAssign(normal));
```

Figure 15: Main computations in the enhanced Java implementation.

The implicitly parallel Java implementation: We first describe how the Array package provides support for implicit parallelism. This allows a user program written in a completely sequential style to obtain the performance benefits of parallelism transparently. The user program does retain control over the extent of parallelism desired, by calling an initializer method of the parallel Array package that sets the number of parallel threads to be used for the application.

The parallel Array package employs a master/slave mechanism in the implicitly parallel routines, and uses the multithreading facilities available in Java. Our design principles in developing this mechanism were: (i) create threads only once and reuse them in different operations; and (ii) allow the master to control the assignment of work to helper threads. The first principle eliminates the cost of creating new threads for each parallel operation. The second principle supports better exploitation of data locality between operations.

Figure 16 shows the basic actions performed when an operation is executed in parallel. Solid boxes indicate mutex operations performed on per-thread data structures, while dashed boxes indicate mutex operation

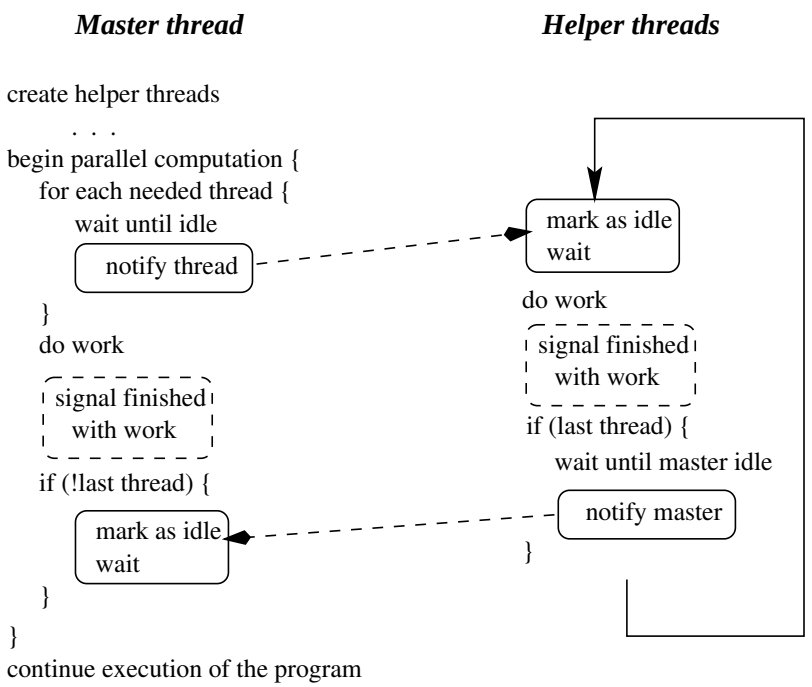


Figure 16: Overview of the thread scheduling operations.

performed on a global shared data structure. Solid arcs indicate flow of control, and dashed arcs indicate notify events. The master thread initially creates the number of helper threads specified by an initialization call from the application program. It then begins executing the program. Each newly created helper thread marks itself as idle, and waits for work on a synchronization object associated with it. This allows the Java `notify` method to be used to restart a specific thread.

When the master thread encounters a parallel operation, it computes T, the desired number of threads to perform the operation. It then notifies helper threads $1, \ldots, T-1$ to start work. All of the threads, including the master, perform their part of the computation. When each thread finishes work on the parallel operation, it signals that by decrementing a shared counter. If the master thread is not the last thread to finish work (as can be determined by examining if the shared counter is zero) it marks itself as idle and waits on its synchronization object. If a helper thread is the last thread to finish, it notifies the master to continue. All helper threads, when finished with the operation, mark themselves as idle, and wait for the next parallel operation.

The implicitly parallel Java version is identical to the enhanced Java version, except for a call at the beginning to an initializer method of the parallel Array package, indicating the number of threads to be used. All parallelism is exploited internally in the parallel Array package in a manner totally transparent to the application.

The explicitly parallel Java implementation: In the explicitly parallel version, it is the data mining application itself that creates additional helper threads. The customers are partitioned into blocks, just as in the enhanced Java version, but the blocks themselves are statically distributed among the threads, following a cyclic distribution. In essence, this corresponds to a block-cyclic distribution of customers to threads. Since each thread operates on one block of customers at a time, it can take advantage of the matrix-matrix operations in BLAS. At the same time, the cyclic distribution of blocks helps achieve better load balance.

matrix	size (rows × columns)	number of non-zeros	fraction of non-zeros
A	10350×2103	397,559	1.83%
S	4800×2103	231,194	2.29%
Φ	10350×4800	20,759,569	41.79%

Table 1: Problem parameters for our data mining application.

The Fortran 90 implementation: The Fortran 90 implementation is a straightforward translation of the enhanced Java. The `doubleArray2D` objects of Java are represented by two-dimensional Fortran arrays of `real*8`. `Index` objects are represented by one-dimensional arrays of `integer`. Matrix multiplication is implemented through calls to the highly optimized `dgemm` routine in ESSL [17]. Array multiplication and division are performed by using the standard `*` and `/` operators of Fortran 90, applied to two-dimensional arrays.

3.3 Experimental results

We now describe the performance results obtained for the different implementations of the data-mining code on an IBM RS/6000 model F50 4-way SMP using 332 MHz PowerPC 604e processors. We show the performance of the scoring algorithm, which is the bulk of the computation in the code. We used the IBM XLF 5.1 compiler (with -O3 -qhot optimization level) for the Fortran 90 program. Results for Java represent the best obtained using both the IBM HPCJ compiler [31] (with -O optimization level) and JDK 1.1.6 with the IBM JIT. (The JDK with JIT delivered the better performance for Java arrays, while the IBM HPCJ compiler was better with the Array package.) The problem parameters are shown in Table 1, which describes the sizes and number of nonzeros in the three main matrices of our application. We remind the reader that the affinity matrix A is stored as a set of dense blocks, one for each group (93 in total). The S matrix is stored in compressed rows. Finally, the Φ matrix, because of its relatively large number of non-zeros, is stored in expanded form.

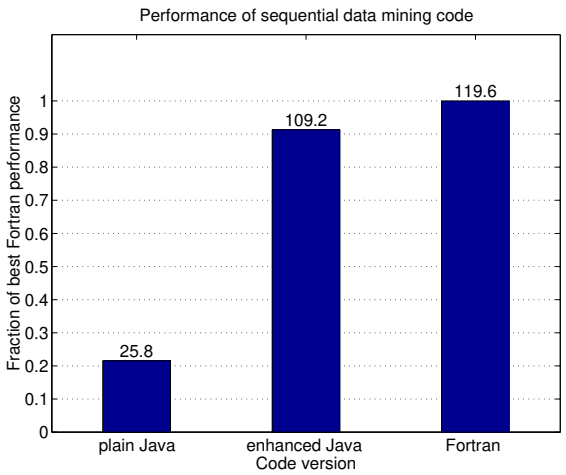


Figure 17: Performance of plain Java (Java arrays), enhanced Java (Array package), and Fortran 90 versions of the data mining code.

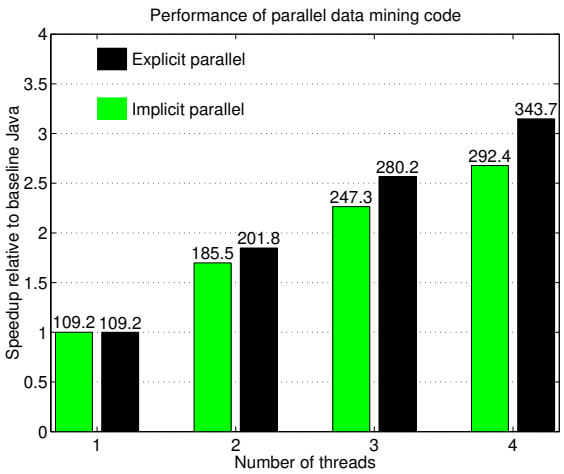


Figure 18: Performance of explicitly parallel and implicitly parallel versions of the data mining code.

Figure 17 shows the serial performance of the Java and Fortran 90 codes, as a fraction of the performance obtained by the Fortran 90 version. The plain Java program (with Java arrays) achieves 25.8 Mflops, which is 22% of the Fortran 90 performance. The enhanced Java version (with the Array package) performs significantly better, achieving 109.2 Mflops, or 91% of the Fortran 90 performance.

Figure 18 shows the performance of the parallel codes for different numbers of threads. The implicitly parallel code achieves a speedup of 2.7 with four threads, reaching a performance of 292.4 Mflops. The explicitly parallel code consistently delivers better performance, achieving 343.7 Mflops on four threads, which represents a speedup of 3.1 over the enhanced Java version.

3.4 Discussion

We now discuss the reasons for the performance differences we have seen in the various implementations of the data mining code. We also draw some general conclusions about writing numerically intensive applications using Java. Finally, we compare the two approaches to exploiting parallelism: (i) writing explicitly parallel code, and (ii) using a parallel subsystem to extract parallelism out of an application written in a completely sequential style.

3.4.1 Performance of serial versions

It is clear that the Array package provides a higher level of abstraction to the programmer than the primitive operations on arrays currently defined in Java. Our experimental results demonstrate that using this package also leads to better performance. There are two main reasons for the difference in performance between the plain Java version (that uses primitive array operations) and the enhanced Java version (that uses the Array package): (i) greater negative impact of array bounds checking on the performance of the plain Java version, and (ii) better data locality and instruction scheduling of the BLAS class library code, used by the enhanced Java version.

In the plain Java program, bounds checking is performed for all array accesses. Since Java supports precise exceptions, all code-reordering transformations must be disabled in loops that have bounds checking. An additional measurement showed that the performance of this program jumps to 36.5 Mflops if all array bounds checks are optimized away. As discussed earlier in Section 2, it is easier for the compiler to apply transformations [24] to deal with this problem if the program uses the Array package. In fact, for the

scatter and gather operations over sparse arrays in our data mining application, static techniques, such as those discussed in [24], are insufficient to optimize the bounds checks, as they cannot handle subscripted subscripts. On the other hand, when using the Array package, bounds checking on an array dimension accessed using an `Index` object is performed simply by comparing the minimum and maximum elements of the `Index` object with zero and the dimension length respectively. The minimum and maximum elements for each `Index` object are computed at creation time in its constructor, to enable efficient bounds checking later.

The BLAS class provides the benefit of making a tuned implementation of commonly needed operations, such as matrix-multiplication, available to the programmer. This is similar to the approach adopted by providers of high-performance libraries like LAPACK [3] and ESSL [17]. As with the implementation of the rest of the Array package, all checks for conditions leading to exceptions are performed at the beginning, so that compiler optimizations can be applied effectively to performance-critical sections of the code.

We observe that by using the Array package, which does not use any native libraries, the Java version of the data mining code achieves performance close to that delivered by the Fortran 90 version. Thus, we show that it is possible to obtain competitive performance even on a numerically-intensive production quality code with pure Java.

3.4.2 Performance of parallel versions

Our experiments with explicitly parallel and implicitly parallel implementations of the data mining code show that (i) it is possible to get the benefits of parallelism by simply using the parallel version of the Array package, and (ii) manual parallelization of an application can often achieve higher performance than a purely library-based approach. Manual parallelization is more amenable to global decisions that increase opportunities for exploiting parallelism and reduce the overhead of parallelization.

The support for implicit parallelism in our implementation of the Array package entirely hides the complexity of parallelization from the end-user. The user merely has to specify the degree of parallelism desired. On the other hand, the implicitly parallel code suffers from four main sources of performance degradation relative to the explicitly parallel code: (i) higher overhead of threading operations, (ii) greater load imbalance, (iii) more serial components, and (iv) poorer cache utilization and a greater amount of cache traffic due to interference in shared data. These factors are largely a consequence of the fact that the implicitly parallel code exploits parallelism at a finer granularity than the explicitly parallel code. The explicitly parallel version has a single dominant outer loop which can be parallelized, so that the overhead of waking up helper threads and waiting for them to finish is incurred only once, and a bigger section of the code is parallelized. Additionally, most of the temporary Arrays used in the explicitly parallel code are read and written by a single thread, so that this version attains very good cache utilization and avoids the problems of cache line invalidations due to false sharing. The implicitly parallel code, by its very nature, relies on library calls over shared data.

We observe that even though the implicitly parallel approach does not lead to the best possible performance, it consistently delivers good performance as the number of threads is increased. On four processors, it leads to a respectable speedup of 2.7 over the sequential code, as compared with a speedup of 3.1 obtained using the explicitly parallel approach. For many users, this performance level would be acceptable, given the ease with which it is obtained.

4 Case study: microstrip

Our second case study is the (greatly simplified) electromagnetics application MICROSTRIP [23]. This example is representative of many applications that deal with physical phenomena. It implements a numerical solver for the partial differential equation that describes the behavior of a physical system.

MICROSTRIP makes heavy use of complex arithmetic. We use it to illustrate some additional Java performance issues, not discussed in the data-mining case study:

- *Complex numbers:* Java does not support complex numbers as primitive types. The typical approach is to define a `Complex` class and then use `Complex` objects to implement complex numbers. We will see that this approach leads to enormous object creation/destruction overhead.
- *Storage efficiency:* This is related to implementing complex numbers as `Complex` objects. Each `Complex` object includes, in addition to the `double` fields for the real and imaginary parts, data that describes the object. This results in a 100% storage overhead. Also, Java arrays of `Complex` (*e.g.*, `Complex[]`, `Complex[][]`) are actually arrays of *references* to `Complex` objects. This leads to poor memory locality and poor memory system performance.

In the case of MICROSTRIP, we achieve high-performance through a combination of *semantic expansion*, discussed in Section 4.3, and features of the Java Array package. Semantic expansion is a compilation technique that replaces method calls by code that directly implements the semantics of those methods. This technique allows us to support complex numbers in Java as efficiently as in Fortran. The Array Package has a strong synergy with semantic expansion. Multidimensional Arrays of complex numbers (*e.g.*, `ComplexArray1D` and `ComplexArray2D`) deliver better locality than Java arrays of `Complex` objects. However, semantic expansion is absolutely necessary to allow efficient access to elements of these arrays.

4.1 The MICROSTRIP computation

In a shielded microstrip structure (see Figure 19(a)), n parallel microstrips run inside a homogeneous insulator. A conducting shield, at a reference potential $V_{\text{shield}} = 0$, completely encloses the structure. Each microstrip k is at a voltage V_k with respect to the shield. We want to compute the value of the potential field $\Phi(x, y)$ in each point of the structure. The potential field $\Phi(x, y)$ is described by a partial differential equation (PDE).

The first step in computing a numerical solution for the field $\Phi(x, y)$ is to discretize the problem domain through a rectangular *grid* of evenly spaced points. Figure 19(b) illustrates this discretization. The grid has shape $(w + 1) \times (h + 1)$ and s is the *grid step*, or space between consecutive grid points. Points in the structure can be identified by their coordinates in the grid. We use the notation $\Phi_{i,j}$ to denote the value of the potential field in point (i, j) of the grid. Using the discretization, and applying several simplifications to the Physics of the problem, we arrive at the following equation for the potential field:

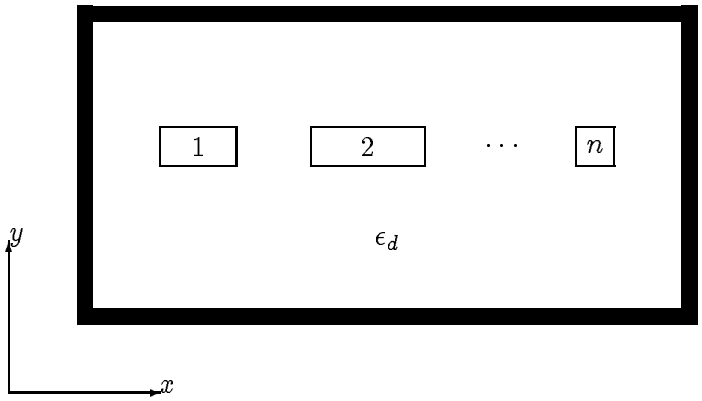
$$\Phi_{i+1,j} + \Phi_{i-1,j} + \Phi_{i,j+1} + \Phi_{i,j-1} - 4\Phi_{i,j} = 0 \tag{8}$$

for all points (i, j) of the insulator. This, in fact, corresponds to a system of equations in the unknows $\Phi_{i,j}$, for all i and j. One approach to solving such a system is through an iterative method called *Jacobi relaxation*. We start with some arbitrary initial approximation of the potential field, $\Phi^0_{i,j}$, for all points (i, j). (The values on the microstrips and the shield are fixed by the problem boundary conditions.) Then, for each point (i, j), we compute a better approximation of the solution through the expression

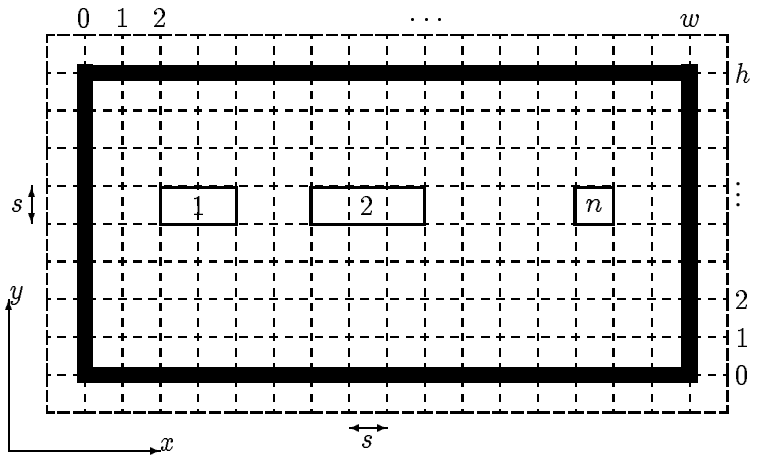
$$\Phi^1_{i,j} = \frac{1}{4}(\Phi^0_{i+1,j} + \Phi^0_{i-1,j} + \Phi^0_{i,j+1} + \Phi^0_{i,j-1}). \tag{9}$$

In the next iteration of the method we compute an even better approximation:

$$\Phi^2_{i,j} = \frac{1}{4}(\Phi^1_{i+1,j} + \Phi^1_{i-1,j} + \Phi^1_{i,j+1} + \Phi^1_{i,j-1}). \tag{10}$$



(a) continuous domain



(b) discretized domain

Figure 19: A shielded microstrip structure.

We continue this process until we converge to some satisfactory solution. (A typical condition is that we want the difference between Φ^k and Φ^{k+1} to be very small.) From a programming point of view, we represent the field by two arrays, `a` and `b`, of size $(w+1) \times (h+1)$. We can then write the relaxation step as

for $i = 1, 2, \ldots, w-1$
  for $j = 1, 2, \ldots, h-1$
    $b(i,j) = \frac{1}{4}(a(i+1,j) + a(i-1,j) + a(i,j+1) + a(i,j-1))$ (11)
  end
end

where `a` is used to represent the current value of the potential field Φ and `b` its next value. We are interested in the case where Φ is a complex-valued function. We can measure progress in our iterative solver by

computing the error between the previous and current approximation for Φ as follows:

$$
\begin{array}{l}
\text{error} = 0 \\
\textbf{for } i = 0, 1, \ldots, w \\
\quad \textbf{for } j = 0, 1, \ldots, h \\
\quad\quad \text{error} = \text{error} + \textbf{abs}(b(i,j) - a(i,j)) \\
\quad \textbf{end} \\
\textbf{end} \\
\text{error} = \text{error}/((w+1) \times (h+1))
\end{array}
\tag{12}
$$

The entire iterative solver is a combination of relaxation steps and error computation, as shown in Figure 20. For clarity, we omit the code that handles the boundary conditions at the microstrips. Although important, that code represents only a small fraction of the computation. The interested reader can obtain more detailed information from [23].

---

while $(error > \varepsilon)$

```
  (* relaxation step *)
  for i = 1, 2, ..., w - 1
    for j = 1, 2, ..., h - 1
      b(i,j) = 1/4 (a(i+1,j) + a(i-1,j) + a(i,j+1) + a(i,j-1))
    end
  end

  (* another relaxation step, to put things back into a *)
  for i = 1, 2, ..., w - 1
    for j = 1, 2, ..., h - 1
      a(i,j) = 1/4 (b(i+1,j) + b(i-1,j) + b(i,j+1) + b(i,j-1))
    end
  end

  (* error computation *)
  error = 0
  for i = 0, 1, ..., w
    for j = 0, 1, ..., h
      error = error + abs(b(i,j) - a(i,j))
    end
  end
  error = error/((w+1) x (h+1))
end
```

Figure 20: Iterative solver for MICROSTRIP.

4.2 The MICROSTRIP implementations

This section discusses three different implementations of the iterative solver for MICROSTRIP. The first version is a *plain* Java implementation that uses Java arrays of `Complex` objects. The second implementa-

tion is our *enhanced* Java code, which uses classes from the Array package. Each of the two Java versions can be compiled with or without the semantic expansion technique. As we shall see in Section 4.4, semantic expansion has a significant impact on performance. Finally, we describe a Fortran 90 version of the solver. This Fortran version is the yardstick by which we measure how good is the absolute performance of our Java versions.

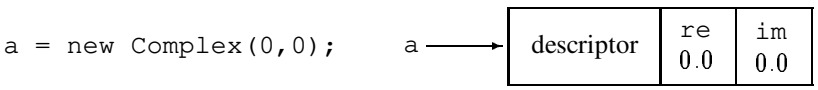
The plain Java implementation In the Fortran programming language, complex numbers are supported as primitive types. Therefore, the relaxation and error computation steps are coded very much as shown in the pseudo-code (11) and (12). Java, in contrast, does not support primitive complex types. The typical approach to performing complex arithmetic in Java is to use a `Complex` class. Some components of this class, which is being proposed for standardization by the Java Grande Forum [19], are shown in Figure 21. Objects of this class are used to represent complex numbers. Using this approach, the relaxation and error computation steps can be coded in Java as shown in Figure 22. Note that `a` and `b` in that code are arrays of *references* to `Complex` objects. That is why we need an initialization loop to create the individual objects. We node that a `Complex` object requires 32 bytes for representation: A (typically) 16-byte object descriptor (present in all Java objects) and two 8-byte fields for the real and imaginary parts.

```
public final class Complex {

    public double re, im;

    public Complex(double r, double i) { re = r; im = i; }
    public Complex assign(Complex z) { re = z.re; im = z.im; return this; }
    public Complex plus(Complex z) { return new Complex(re+z.re,im+z.im); }
    public Complex minus(Complex z) { return new Complex(re-z.re,im-z.im); }
    public Complex times(Complex z) {
        return new Complex(re*z.re-im*z.im,im*z.re+re*z.im);
    }
    public Complex times(double x) { return new Complex(re*x,im*x); }
    public Complex plusAssign(Complex z) { re += z.re; im += z.im; return this; }
}
```

(a) code for `Complex` class



(b) `Complex` object creation and representation

Figure 21: A standard Java class for complex numbers.

The enhanced Java implementation An alternative to representing `a` and `b` as Java arrays of `Complex` objects is to use the multidimensional Array classes for `Complex` from the Array package. Figure 23 shows some components of class `ComplexArray2D`, which implements two-dimensional Arrays of complex numbers. Inside the Array, values are stored in a packed form, with (*real*,*imaginary*) pairs in adjacent locations of the `data` array. This is similar to the Fortran style for storing arrays of complex numbers and only requires 16 bytes per complex value. The `get` operation returns a `Complex` object with the value of an Array element, and the `set` operation assigns the value of a `Complex` object to an Array element. It is

```
Complex[][] a = new Complex[w+1][h+1];
Complex[][] b = new Complex[w+1][h+1];
for (i=0; i<w+1; i++) {
   for (j=0; j<h+1; j++) {
      a[i][j] = new Complex(0,0);
      b[i][j] = new Complex(0,0);
   }
}


for (i=1; i<w; i++)
   for (j=1; j<h; j++)
      b[i][j].assign(a[i+1][j].plus(a[i-1][j]).plus(a[i][j+1]).plus(a[i][j-1]).times(0.25));

error = 0.0;
for (i=0; i<w+1; i++)
   for (j=0; j<h+1; j++)
      error = error + b[i][j].minus(a[i][j]).abs();
error = error / ((w+1)*(h+1))
```

Figure 22: Relaxation code in Java using arrays of references to `Complex` objects.

easy to create Arrays using the `ComplexArray2D` class:

```
ComplexArray2D a = new ComplexArray2D(w+1,h+1)
```

creates and initializes to zero a $(w+1) \times (h+1)$ Array of *complex values* (not complex objects). Using the multidimensional Arrays from the Array package, the relaxation and error steps can be coded as shown in Figure 24.

```
public final class ComplexArray2D {

   private double[] data;

   public ComplexArray2D(int m, int n) {
       data = new double[2*m*n];
   }

   public Complex get(int i, int j) {
       return new Complex(data[2*(i*n+j)],data[2*(i*n+j)+1]);
   }

   public void set(int i, int j, Complex z) {
       data[2*(i*n+j)] = z.re;
       data[2*(i*n+j)+1] = z.im;
   }
}
```

Figure 23: A (proposed) standard Java class for two-dimensional complex array.

The Fortran 90 implementation: The Fortran 90 implementation is a straightforward translation of the enhanced Java. The `ComplexArray2D` objects of Java are represented by two-dimensional Fortran arrays

```
ComplexArray2D a = new ComplexArray2D(w+1,h+1);
ComplexArray2D b = new ComplexArray2D(w+1,h+1);

for (i=1; i<w; i++)
  for (j=1; j<h; j++)
    b.set(i,j,a.get(i+1,j).plus(a.get(i-1,j)).plus(a.get(i,j+1)).plus(a.get(i,j-1)).times(0.25);

error = 0.0;
for (i=0; i<w+1; i++)
  for (j=0; j<h+1; j++)
    error = error + b.get(i,j).minus(a.get(i,j).abs();
error = error / ((w+1)*(h+1))
```

Figure 24: Relaxation and error steps using the Array package.

of `complex*16`. The code for the iterative solver in Fortran looks very much like the pseudo-code in Figure 20. This Fortran version achieves approximately 50% of the peak performance of our benchmark machine. (It achieves 120 out of a 266 peak Mflops.) Therefore, it constitutes a good reference point against which we compare our Java performance.

4.3 Semantic expansion

As previously mentioned, implementing complex numbers as Java objects leads to poor performance on programs written using the `Complex` class and Array package. Each `get` operation on a `ComplexArray2D` and each arithmetic operation on a pair of `Complex` objects creates a temporary `Complex` object. A more sophisticated compiler can use a technique called *semantic expansion* to achieve dramatically better performance.

Semantic expansion treats standard classes as language primitives. Unlike traditional procedure or method inlining, which (intuitively) treats the inlined method as a macro, semantic inlining uses the compiler's built-in knowledge of the class semantics to directly generate efficient, legal code. The arithmetic methods of class `Complex` are treated by semantic expansion as operating and returning *complex values* (pairs of (*real*, *imaginary*) `doubles`). If the consumers of theses values are other arithmetic methods of `Complex`, they are semantically expanded to directly use the complex values. Otherwise, the value is converted to a `Complex` object, with all the properties of a regular Java object. By treating complex numbers as values whenever possible, temporary `Complex` objects do not have to be created for arithmetic operations, thus reducing the allocation and garbage collection overhead. This approach delivers the same performance benefit for complex numbers as the proposed *value object* extension for Java [19], without requiring language changes. `Complex` objects continue to be used where required by program semantics.

Semantic expansion is also applied to `set` and `get` methods of the complex-valued arrays in the Array package. The method `get` is semantically expanded to return a complex value, which can directly be used by the (semantically expanded) arithmetic methods of `Complex`. Correspondingly, the `set` method is semantically expanded to accept a complex value, thus directly using the result of (semantically expanded) arithmetic operations on `Complex`.

4.4 Experimental results

We now discuss the performance results obtained for the different implementations of the MICROSTRIP code on an IBM RS/6000 model 590 workstation. This machine has a 67 MHz POWER2 processor with a 256 kB single-level data cache and 512 MB of main memory. The peak computational speed is 266 Mflops. We used the IBM HPCJ compiler (with -O optimization level) to compile Java programs, and the XLF 4.1

compiler (with -O3 optimization level) for the Fortran program. (For this benchmark, HPCJ always performs better than JDK 1.1.6 with the IBM JIT.)

Figure 25 shows five different results for the MICROSTRIP benchmark. The problem parameters are those described in [23]. The grid size is 1000×1000 and there are 4 microstrips, of cross section 100×10 grid steps, running through the insulator. The bars labeled plain and enhanced show the performance for the plain (Java arrays) and enhanced (Array package) Java implementations, respectively, *without* semantic expansion. Both versions achieve only 1 Mflop. This is the performance that would typically be delivered by current Java environments. The bars plain+exp and enhanced+exp show the performance for the plain and enhanced Java implementations, respectively, *with* semantic expansion. The plain Java version achieves 45.6 Mflops with semantic expansion, while the enhanced Java version achieves 75.8 Mflops. Finally, the bar labeled Fortran shows the performance of the Fortran implementation. The Fortran version achieves 120.5 Mflops, or 45% of the peak computational speed of the machine.

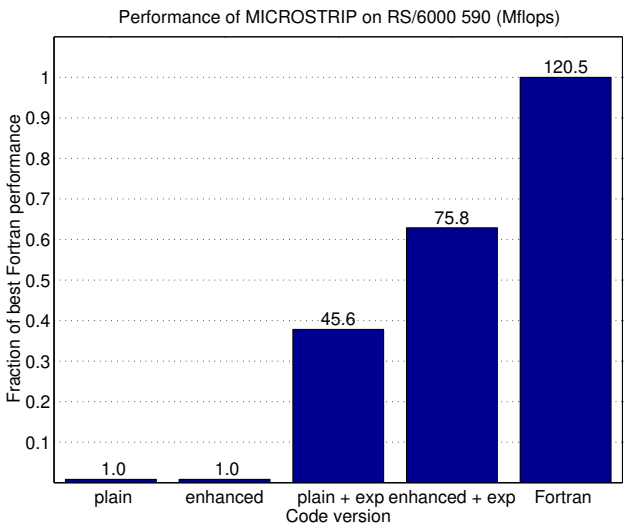


Figure 25: Summary of results for MICROSTRIP. In the plot, plain and enhanced refer to Java arrays and Array package, respectively. The +exp notation indicates use of semantic expansion.

4.5 Discussion

We now discuss why the performance of the Java versions without semantic expansion is so poor. We also discuss why, with semantic expansion, the performance of the enhanced Java version is better than the plain version. We note that the synergy between the Array package and semantic expansion leads to pure Java performance that is a respectable 60% of the best Fortran performance we can achieve.

4.5.1 Java without semantic expansion

Coding the relaxation step straightforwardly as shown in Figure 22 leads to very poor performance in current implementations of Java. The reason is that each arithmetic operation (methods `plus` and `times`) creates a new `Complex` object to represent the results. (See Figure 21.) Object creation is a very expensive operation, which causes the plain Java version of MICROSTRIP to be approximately 100 times slower than the reference Fortran version.

The `ComplexArray2D` class offers a significant storage benefit compared to `Complex[][]`, since it only stores complex values and not `Complex` objects (a 50% reduction in storage, and associated im-

provements in memory bandwidth and cache behavior). The performance of numerical codes that use `ComplexArray2D`, however, is just as bad as codes that use Java arrays of `Complex` objects. Execution continues to be dominated by object creation and destruction. In fact, every `get` operation on a `ComplexArray2D` results in a new temporary object. The performance of this enhanced version on current Java environments is also approximately 100 times slower than the reference Fortran.

4.5.2 Java with semantic expansion

The benefits of treating complex numbers as values rather than objects are enormous. We were able to increase the performance 45- and 75-fold for the plain and enhanced versions, respectively. Of particular importance is the synergy between the Array package and semantic expansion. By eliminating the cost of creating temporary objects, semantic expansion make the use of `ComplexArray2D` objects viable from a performance perspective. `ComplexArray2D`, in turn, delivers better memory behavior than Java arrays of `Complex` objects. The combination of the two approaches delivers a Java performance on a complex arithmetic application that is a respectable 60% of the best Fortran performance.

5 Additional experiments

We performed additional experiments to further illustrate the performance impact of our techniques for optimizing Java numerical codes. In this section we present a summary of results from eight benchmarks. Four of these benchmarks perform real arithmetic and the other four complex arithmetic. We compare the performance of Java and Fortran versions of the benchmarks. We perform all our experiments on an IBM RS/6000 model 590 workstation. This machine has a 67 MHz POWER2 processor with a 256 kB single-level data cache and 512 MB of main memory. The peak computational speed of this machine is 266 Mflops.

Fortran programs are compiled using version 4.1 of the IBM XLF compiler with the highest level of optimization (-O3 -qhot, which performs high-order loop transformations [29]). Results for two different Java versions of each benchmark are reported. One version, which we call *plain*, is a straightforward implementation of the benchmark using Java arrays. The numbers reported for this version correspond to the best obtained using both JDK 1.1.6 with the IBM JIT and IBM HPCJ. (The JIT, which incorporates many advanced optimizations such as bounds checking and `null` pointer checking elimination, performs better in the real arithmetic codes, while IBM HPCJ does better in most of the complex arithmetic codes.) The other Java version, which we call *best*, represents the best result we achieved from a pure Java implementation of the benchmark. It uses multidimensional Arrays from the Array package, aggressive compiler optimizations to generate and exploit safe regions of code (*i.e.*, regions without potential exceptions), semantic expansion, and the fused multiply-add operation of the POWER2 hardware. This version is compiled with our prototype research version of the IBM HPCJ compiler.

5.1 Real arithmetic benchmarks

The real arithmetic benchmarks are: CHOLESKY, BSOM, SHALLOW, and TOMCATV. CHOLESKY is a straightforward implementation of the Cholesky decomposition algorithm, as described, for example, in [15, 27]. It computes the upper triangular factor U of a symmetric positive definite matrix A such that $A = U^T U$. We implemented our code so that the factorization is performed in place, with the upper half of matrix A being replaced by the factor U. For our experiments, we factor a matrix of size 1000×1000. BSOM (Batch Self-Organizing Map) benchmark is a data-mining kernel representative of technologies incorporated into Version 2 of the IBM Intelligent Miner. (It was not used in the data mining application described in Section 3.) It implements a neural-network-based algorithm to determine clusters of input

records that exhibit similar attributes of behavior. We time the execution of the *training* phase of this algorithm, which actually builds the neural network. This phase consists of 25 *epochs*. Each epoch updates a neural network with 16 nodes using 256 records of 256 fields each. SHALLOW is a computational kernel from a shallow water simulation code from the National Center for Atmospheric Research (NCAR). The data structures in SHALLOW consist of 14 matrices (2-dimensional arrays) of size $n \times m$ each. The computational part of the code is organized as a time step loop, with several array operations executed in each time step. For our measurements we fix the number of time steps $T = 20$ and use $n = m = 256$. TOMCATV is part of the SPECfp95 suite (`www.spec.org`). It is a vectorized mesh generation with Thompson solver code. The benchmark consists of a main loop, which iterates until convergence or until a maximum number of iterations is executed. At each iteration of the main loop, the major computation consist of a series of stencil operations on two (coupled) grids, X and Y, of size $n \times n$. In addition, a set of tridiagonal systems are solved through LU decomposition. For our experiments, we use a problem size $n = 513$.

Results for these four benchmarks are summarized in Figure 26. The height of each bar is proportional to the best Fortran performance achieved in the corresponding benchmark. The numbers at the top of the bars indicate actual Mflops. For the plain Java version, two-dimensional arrays of `doubles` were represented using a Java `double[][]` array. The best Java version uses `doubleArray2D` Arrays from the Array package. The use of multidimensional Arrays from the Array package in turn enables bounds checking and other optimizations. In all cases we observe significant performance improvements between the plain and best Java versions. Improvements range from a factor of 18 (4.2 to 74.6 Mflops for TOMCATV) to a factor of 34 (3.8 to 129.9 Mflops for CHOLESKY) We achieve Java performance that ranges from 80% (SHALLOW) to 95% (CHOLESKY and BSOM) of fully optimized Fortran code.

5.2 Complex arithmetic benchmarks

The complex arithmetic benchmarks are: MATMUL, LU, FFT, and CFD. MATMUL computes $C = C + A \times B$, where C, A, and B are complex matrices of size 500×500. We use a dot-product version of matrix multiplication, with an i, j, k-loop nest. The i, j, and k loops are blocked and the i and j loops are unrolled, in all versions, to improve performance [29]. LU is a straightforward implementation of Crout's algorithm [15, 27] for performing the LU decomposition of a square matrix A, with partial pivoting. The factorization is performed in place and, in the benchmark, A is of size 500×500. FFT computes the discrete Fourier transform of a two-dimensional complex function, represented by an $n \times m$ complex array. We use the Daniel-Lanczos method described in [27] to compute the one-dimensional FFTs in the two-dimensional FFT. For our experiments we use $n = m = 256$. CFD is a kernel from a computational fluid dynamics application. It performs three convolutions between pairs of two-dimensional complex functions. Each function is represented by an $n \times m$ complex array. The computation performed in this benchmark is similar to a two-dimensional version of the NAS parallel benchmark FT. Again, for our experiments we use $n = m = 256$.

Results for these four benchmarks are summarized in Figure 27. Again, the height of each bar is proportional to the best Fortran performance achieved in the corresponding benchmark, and the numbers at the top of the bars indicate actual Mflops. For the plain Java version, complex arrays were represented using a `Complex[][]` array of `Complex` objects. No semantic expansion was applied. The best Java version uses `ComplexArray2D` Arrays from the Array package and semantic expansion. In all cases we observe significant performance improvements between the plain and best Java versions. Improvements range from a factor of 35 (1.7 to 60.5 Mflops for CFD) to a factor of 75 (1.2 to 89.5 Mflops for MATMUL). We achieve Java performance that ranges from 65% (MATMUL and LU) to 90% (FFT and CFD) of fully optimized Fortran code.

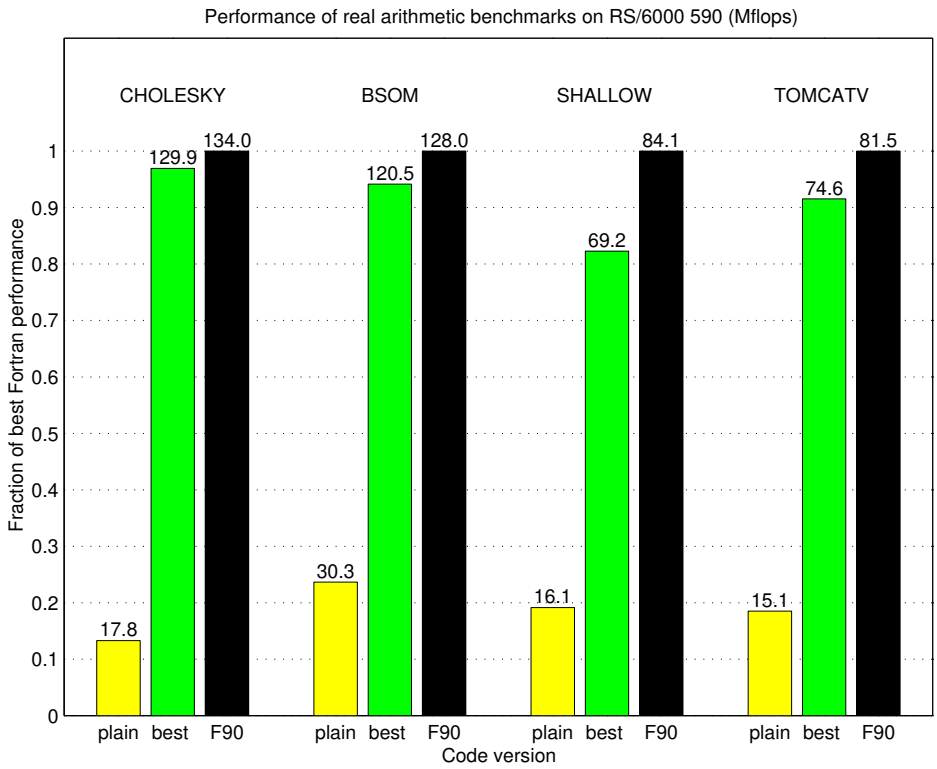


Figure 26: Summary of experimental results for real arithmetic benchmarks CHOLESKY, BSOM, SHALLOW, and TOMCATV. For each benchmark, three code versions: plain is straightforward Java, best is the best Java performance using Array package and compiler optimization, F90 is best Fortran performance. Numbers at the top of the bars indicate actual Mflops.

6 Related work

The importance of having high-performance libraries for numerical computing in Java has been recognized by many authors. In particular, [6, 8, 30] describe projects to develop such libraries entirely in Java. In addition, there are approaches in which access to existing numerical libraries (typically coded in C and Fortran) is provided to Java applications [5, 9, 14]. The Array package includes a BLAS class, so that efficient linear algebra libraries can be developed for it.

It is well know that programming with libraries has its limitations. Certain classes of applications can be effectively expressed in terms of linear algebra operations. These applications can be efficiently implemented on top of (very successful) linear algebra libraries such as LAPACK and ESSL. Other classes of applications are not so well structured. In those cases, the programmer has to develop the application from scratch, counting with minimal library support. For this reason, the Array package is more than just a library: It is a mechanism to represent array operations in a manner that can be highly optimized by compilers. Linear algebra code can be developed with the Array package using the BLAS primitives. Less structured code can use the Array operations directly. With the right compiler support, Java code built on top of the most basic `set` and `get` operations of the Array package can achieve Fortran-like performance.

In [11], Cierniak and Li describe a global analysis technique for determining that the shape of a Java array of arrays (*e.g.*, `double[][]`) is indeed rectangular and not modified. The array representation is then transformed to a dense storage and element access is performed through index arithmetic. This approach can lead to results that are as good as from using multidimensional Arrays in the Array package. We differ

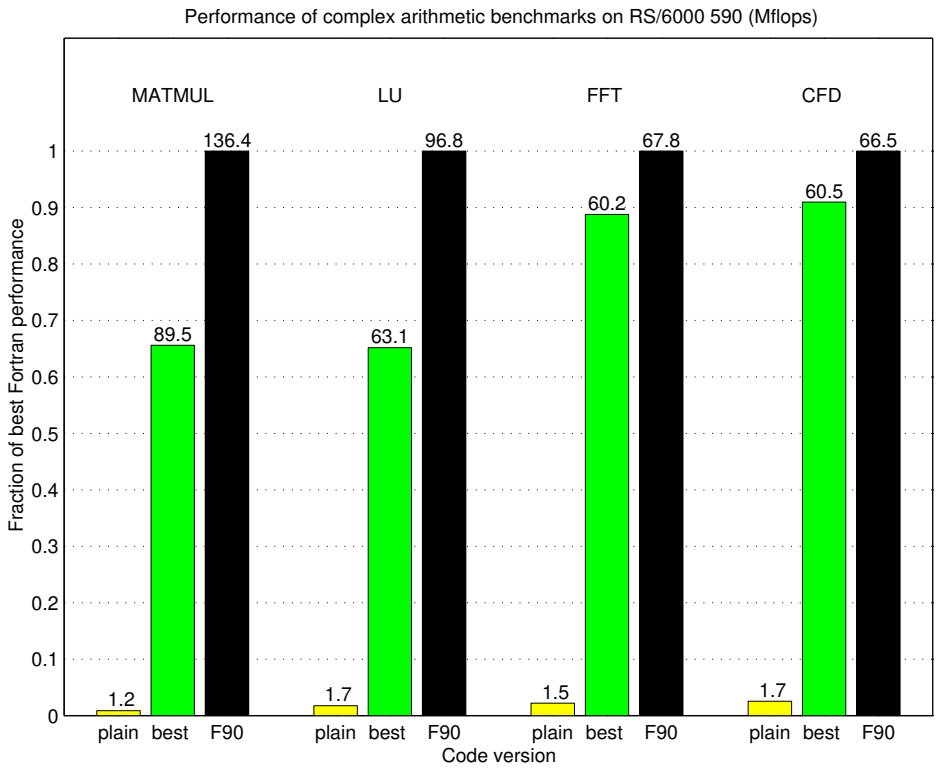


Figure 27: Summary of experimental results for complex arithmetic benchmarks MATMUL, LU, FFT, and CFD. For each benchmark, three code versions: plain is straightforward Java, best is the best Java performance using Array package and compiler optimization, F90 is best Fortran performance. Numbers at the top of the bars indicate actual Mflops.

in that we do not need global analysis: Arrays from the Array package are intrinsically rectangular and can always be stored in a dense form. We have also integrated support for complex numbers in our true multidimensional Arrays.

The technique of semantic expansion we use is similar to the approach taken by Fortran compilers in implementing some *intrinsic* procedures [1]. Intrinsic procedures can be treated by the Fortran compiler as language primitives, just as we treated the `Complex` class in Java. Semantic expansion differs from the handling of intrinsics in Fortran in that both new data types and operations on the new data types are treated as language primitives. Still regarding semantic expansion, the work by Peng and Padua [36] targets standard container classes for semantic expansion. In their paper, the container semantics are not used for local optimizations or transformations inside the container. Instead, the semantics are used to help data flow analysis and detect parallelizable loops. Their work illustrates how semantic information about standard classes exposes new optimization and parallelization opportunities.

Array operations are natural candidates for exploitation of parallelism. This has been done extensively before both at the language [10, 33, 7] and subsystem [18, 28, 26] levels, with different degrees of transparency to the application programmer. So far, we have successfully exploited parallelism inside the Array package operations while achieving total transparency to the programmer. A combination of the Array package with semantic expansion can lead to programs that are amenable to automatic parallelization using techniques developed, for example, for Fortran. This will result in transparent parallelization of the user code itself.

The efficacy of the Array package in supporting applications of a varied nature is demonstrated by the results of Sections 3, 4, and 5. The strong synergy between the Java Array package for multidimensional Arrays and the aggressive compiler optimizations is the main differentiator in our work.

7 Conclusion

We have demonstrated that high-performance numerical codes can be developed in Java. The benefits of Java from a productivity and demographics point of view have been known for a while. Many people have advocated the development of large, computationally intensive applications in Java based on those benefits. However, Java performance has consistently been considered an impediment to its applicability to numerical codes. This paper has presented a set of techniques that lead to Java performance that is competitive with the best Fortran implementations available today.

We have used a class library approach to introduce in Java features of Fortran 90 that are of great importance to numerical computing. These features include complex numbers, multidimensional arrays, and libraries for linear algebra. These features have a strong synergy with the compiler optimizations that we are developing, particularly bounds checking optimization and semantic expansion. We have demonstrated this synergy, and the resulting performance impact, through two detailed case studies. We have also performed a broader performance evaluation using eight other numerical benchmarks.

We demonstrated that we can deliver Java performance in the 60-95% range of the best Fortran performance for a variety of benchmarks. This is a very important step in making Java the preferred environment for developing large-scale computationally intensive applications. However, there is more than just performance to our approach. The Array package and associated libraries create a Java environment with many of the features that experienced Fortran programmers have grown accustomed to. This combination of delivering Fortran features *and* performance in Java is the core of our approach to making Java the environment of choice for new numerical computing.

Acknowledgments: The authors wish to thank Ven Seshadri of IBM Canada for helping and supporting our experiments with HPCJ.

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] R. Agrawal, T. Imielinski, and A. Swami. Mining associations between sets of items in massive databases. In *Proceedings of the ACM-SIGMOD 1993 International Conference on Management of Data*, pages 207–216, Washington, DC, May 1993.

[3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, 1995.

[4] Utpal Banerjee. *Dependence Analysis*. Loop Transformations for Restructuring Compilers. Kluwer Academic Publishers, Boston, MA, 1997.

[5] A. Bik and D. B. Gannon. A note on level 1 BLAS in Java. In *Proceedings of the Workshop on Java for Computational Science and Engineering - Simulation and Modeling II*, June 1997. available at `http://www.npac.syr.edu/users/gcf/03/javaforcse/acmspecissue/latestpapers.html`.

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

[7] F. Bodin, P. Beckmann, D. Gannon, S. Narayana, and S. X. Yang. Distributed pC++: Basic ideas for an object parallel language. *Scientific Programming*, 2(3):7–22, 1993.

[8] 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`.

[9] H. Casanova, J. Dongarra, and D. M. Doolin. Java access to numerical libraries. *Concurrency, Pract. Exp. (UK)*, 9(11):1279–91, November 1997. Java for Computational Science and Engineering - Simulation and Modeling II Las Vegas, NV, USA 21 June 1997.

[10] W.-M. Ching and D. Ju. Execution of automatically parallelized APL programs on RP3. *IBM Journal of Research and Development*, 35(5/6):767–777, 1991.

[11] M. Cierniak and W. Li. Just-in-time optimization for high-performance Java programs. *Concurrency, Pract. Exp. (UK)*, 9(11):1063–73, November 1997. Java for Computational Science and Engineering - Simulation and Modeling II, Las Vegas, NV, June 21, 1997.

[12] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. Society for Industrial and Applied Mathematics, 1991.

[13] C.-W. Tseng G. Rivera. Data transformations for eliminating conflict misses. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pages 38–49, Montreal, Canada, June 1998. ACM Press.

[14] V. Getov, S. Flynn-Hummel, and S. Mintchev. High-performance parallel programming in Java: Exploiting native libraries. In *ACM 1998 Workshop on Java for High-Performance Network Computing*. ACM SIGPLAN, 1998. Available at `http://www.cs.ucsb.edu/conferences/java98`.

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

[16] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, November 1997.

[17] International Business Machines Corporation. *IBM Engineering and Scientific Subroutine Library for AIX – Guide and Reference*, December 1997.

[18] International Business Machines Corporation. *IBM Parallel Engineering and Scientific Subroutine Library for AIX – Guide and Reference*, December 1997.

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

[20] 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.

[21] 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*, volume 1470 of *Lecture Notes in Computer Science*, pages 422–434. Springer Verlag, September 1998.

[22] S. P. Midkiff, J. E. Moreira, and M. Snir. Optimizing bounds checking in Java programs. *IBM Systems Journal*, 37(3):409–453, August 1998.

[23] J. E. Moreira and S. P. Midkiff. Fortran 90 in CSE: A case study. *IEEE Computational Science & Engineering*, 5(2):39–49, April-June 1998.

[24] 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.

[25] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufman, 1997.

[26] R. Parsons and D. Quinlan. Run time recognition of task parallelism within the P++ parallel array class library. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 77–86. IEEE Comput. Soc. Press, October 1993.

[27] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in FORTRAN: The Art of Scientific Computing*. Cambridge University Press, 1992.

[28] J. V. W. Reynders, J. C. Cummings, M. Tholburn, P. J. Hinker, S. R. Atlas, S. Banerjee, M. Srikant, W. F. Humphrey, S. R. Karmesin, and K. Keahey. POOMA: A framework for scientific simulation on parallel architectures. In *Proceedings of First International Workshop on High Level Programming Models and Supportive Environments, Honolulu, HI*, pages 41–49, April 16 1996. Technical report available at `http://www.acl.lanl.gov/PoomaFramework/papers/papers.html`.

[29] V. Sarkar. Automatic selection of high-order transformations in the IBM XL Fortran compilers. *IBM Journal of Research and Development*, 41(3):233–264, May 1997.

[30] 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`.

[31] V. Seshadri. IBM High Performance Compiler for Java. *AIXpert Magazine*, September 1997. Available at URL `http://www.developer.ibm.com/library/aixpert`.

[32] E. Simoudis. Reality check for data mining. *IEEE Expert: Intelligent systems and their applications*, 26(3), October 1996.

[33] R. G. Willhoft. Parallel expression in the APL2 language. *IBM Systems Journal*, 30(4):498–512, 1991.

[34] G. V. Wilson and P. Lu, editors. *Parallel Programming using C++*. MIT Press, 1996.

[35] P. Wu, S. P. Midkiff, J. E. Moreira, and M. Gupta. Efficient support for complex numbers in Java. In *To appear, Proceedings of the 1999 ACM Java Grande Conference*, 1999. available as IBM Research Division technical report no. 21393.

[36] Peng Wu and David Padua. Beyond arrays – a container-centric approach for parallelization of real-world symbolic applications. In *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing, LCPC'98*, 1998.