Java is not thought of as being competitive with Fortran for numerical programming. In this paper, we discuss technologies that can and will deliver Fortran-like performance in Java. These techniques include new and existing compiler technologies, the exploitation of parallelism, and a collection of Java libraries for numerical computing. We also present experimental data to show the effectiveness of our approaches. In particular, we achieve 1 Gflops with a linear algebra kernel on an RS/6000 SMP machine. Most of these techniques require no language changes; a few depend on extensions to Java currently under consideration.

1 Introduction

Although great progress is being made in the optimization of Java, it is still not considered to be a serious language for developing numerically intensive applications. Given the present state of the technology, these assumptions are justified. At the same time, the work described in this paper shows that the poor performance by Java on numerically intensive programs is not inherent to the language. We performed a simple comparison of Java and Fortran using two test programs. In both cases the Java programs were compiled using the IBM HPCJ static compiler [9, 18], which produces native code, and the Fortran programs were compiled using xlf90, a production Fortran compiler from IBM. The first program, a CFD application using complex numbers, was run on a 67 MHz POWER2 machine. Fortran performed 40 times better than a similarly structured Java code. After applying some of the techniques described in this paper [19], Java performance was 90% of that of Fortran. In [13], we examined a 500 × 500 matrix multiply on a 332 MHz PowerPC 604e processor. The Fortran code performed 50 times faster than a similarly structured Java code, and the hand-tuned ESSL (Engineering Scientific Subroutine Library) [8] function, also written in Fortran, performed almost 60 times faster. Again, after applying some of the transformations described in this paper, Java achieved 80% of the ESSL performance.

Why is the Fortran performance so much better than out-of-the-box Java? In the next section we identify five factors that inhibit Java performance, and discuss how these inhibitors can be overcome. We note that these issues are at the core of the Java Grande Forum work on improving Java performance for high-end computing [10].

2 Enhancing Java performance for numerical programs

The factors that we have identified in Java as major performance inhibitors are: (i) A precise exception model coupled with array bounds and null-pointer checking; (ii) the lack of true, dense, multidimensional arrays; (iii) no built-in complex data types; (iv) high order loop transformations are inhibited by (i)–(iii); and (v) no good libraries (e.g., ESSL) that allow easy access to well-tuned standard numerical operations. In the remainder of this paper we discuss lifting these impediments – while maintaining 100% pure Java – and enabling Fortran-like performance.

2.1 Array bounds checking optimizations

Java has as a primitive type only one-dimensional arrays. Two-dimensional arrays are simulated by having a one-dimensional array whose elements themselves are one-dimensional arrays (see Figure 1). Higher dimensional arrays are constructed by having a one-dimensional array whose elements are arrays of the next lower dimension.

Before accessing some element, e.g. X[2][3], Java mandates the JVM must ensure that (i) X is not a null-pointer; (ii) X[2] is an in-bounds access of X; (iii) X[2] is not a null-pointer; and (iv) X[2][3] is an in-bounds access of X[2]. If any of these conditions is false, a precise exception is thrown. Because the exception is precise, data accesses cannot be moved relative to possible exception points, which inhibits many compiler transformations. This problem can
be overcome by bounds checking optimizations [11, 13]. Figure 2 shows a versioning transformation for optimizing bounds checks. The original loop is replicated, and an if statement is used to determine if all references within the loop are valid or not. If all references are valid, a version of the loop can be executed that does not perform the null-pointer and bounds checks. More importantly, this loop body is provably exception free, so that operations can be reordered in any way that preserves data dependencies. Otherwise the version with checks and exceptions is executed.

2.2 A standard multi-dimensional array package

The Java array layout (shown in Figure 1) presents additional complications for bounds checking optimizations. In particular, the versioning optimization must be thread safe. With the Java layout, it is possible for another thread to replace some of the rows (possibly changing their size) of array \( X \) in Figure 2 after the if statement is executed. To prevent this, the array of pointers is copied into thread local storage (the for \( c \) loop of Figure 2), as allowed by the Java language semantics.

The Java array semantics also impede compile-time alias analysis. Different rows of two Java arrays can be aliased, as seen in Figure 1. To show that two references \( X(2\ast i,j) \) and \( Y(2\ast i-1,j) \) are independent in Fortran, it is sufficient to show that either \( X \) and \( Y \) are disjoint, or that \( 2\ast i-1 \) and \( 2\ast i \) never take on the same values in the iteration spaces of their surrounding loop nests. This is not sufficient in Java because \( 2\ast i-1 \) and \( 2\ast i \) can refer to the same storage. This occurs in Figure 1 for \( i = 2 \).

Another deficiency of Java has to do with support for the representation of array sections. Operations on array sections are common in Fortran codes, particularly when passing parameters to functions and subroutines. Java does not provide any facilities for the direct representation of array sections. User-defined classes can be used to implement array sections but, because such classes are nonstandard, their reuse across vendor supplied libraries will be very limited.

These problems can be solved by a standard Java Array package. A reference implementation of the package is done using standard Java arrays. This allows applications written using the package to be backward compatible on all Java JVM’s. The reference implementation we propose [10, 15] supports arrays with ranks from one to seven, and whose elements are the primitive numerical types, as well as complex numbers (see Section 2.3). Fortran 90 style array sections are also supported.

Access to array elements is via \( A.set(\sigma,u) \) and \( A.get(\sigma) \) methods, which write and read the element of \( A \) specified by \( \sigma \). The index \( \sigma \) is a comma separated list of subscript expressions, and \( u \) is the value to be assigned to the specified array element. Array operations (e.g., \(+, -, *, \) and \( \div \)) are supported. Thus, \( A.times(B) \) returns the element-wise multiplication of \( A \) and \( B \).

The array class semantics specify that the data storage for an array is shared by two arrays only if they reference the same base storage. This eliminates the problem with aliasing of parts of arrays discussed above. Figure 3 illustrates the abstract data structures implemented by arrays from the Array package. Note that the arrays are rectangular, and that there can be no aliasing between \( Z \) and \( T \).

![Figure 1. A Java two-dimensional array.](image1)

![Figure 2. Two-version bounds checking.](image2)
2.3 Complex numbers

Java lacks primitive complex data types, and therefore the mechanism for implementing complex numbers in Java is by defining a Complex class. (A standard Complex class for Java is being proposed by the Java Grande Forum [10].) Java arrays of Complex can be easily created with conventional language constructs like Complex[] C = new Complex[100]. This approach of implementing complex numbers is not, however, efficient in practice. Let a, b and c be Complex objects, and let plus and times be methods of the Complex class that perform complex addition and multiplication, returning a Complex object with the result. Consider the expression a = a.times(b.plus(c)). Two method invocations (times and plus) and two temporary creations (one each for b.plus(c) and a.times(...) are needed to evaluate this expression. The temporary creations will likely involve an allocation from the heap and a garbage collection. The overhead imposed by implementing complex numbers as objects, and not as a primitive data type, are almost completely eliminated by a technique called semantic inlining, discussed next.

2.4 Semantic Inlining

A naive compiler will yield poor performance on programs written using the standard Complex class and Array package. For example, each array access will require a method invocation, and each get operation on a Complex array will require creating a temporary Complex object. A more sophisticated compiler can use a technique called semantic inlining [19] to achieve dramatically better performance.

Semantic inlining 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. Thus, even though the reference implementation of the Array package is built using Java arrays, the compiler can implement a dense storage scheme as seen in Figure 4. This storage layout is compatible with both the defined semantics of the (proposed) standard class and the functionality of the reference implementation. Through semantic inlining, the get and set operations for an array of doubles (or other Java primitive and Complex types) can generate array indexing code that is identical to that generated by a Fortran compiler. In contrast, traditionally inlining the code in the reference implementation would require pointer chasing over the Java array-of-arrays. Finally, with semantic inlining the thread safety of bounds optimization is made trivial. Arrays from the Array package are immutable in shape. A compiler with that knowledge can avoid the privatization step shown in Figure 2.
the need for compiler analysis that detects non-value oriented method invocations on Complex objects, since such invocations cannot occur in a legal program if Complex is a lightweight object. On the other hand, it requires changes to the Java language, and introduces objects that have a different behavior than regular Java objects.

Another important proposal of the Java Grande Forum is to facilitate the use of classes such as Complex by operator overloading. With operator overloading, a programmer will be able to write \( a = b + c \cdot d \), rather than \( a.assign(b.plus(c.times(d))) \), for complex variables \( a, b, c \) and \( d \). The same applies to methods of the Array package. While this proposal has no impact on performance (and requires no change in the Java byte code or the Java Virtual Machine), it can obviously have a major impact on the usability of Java in Numeric Computing.

Other efforts to implement value objects via global analysis, rather than by the more local analysis needed by semantic inlining or by an explicit value object type, can be found in [5]. In [20], a more general application of what we call semantic inlining can be found.

2.5 High order transformations

The preceding techniques have produced exception free loop nests (bounds checking optimizations), reduced the possibilities of aliasing among arrays (the standard Array package) and put multidimensional array access and representation in a form that is amenable to traditional data flow and data dependence analysis. Therefore, the standard high order transformations (e.g., loop interchange, tiling, array privatization, unroll-and-jam, scalar replacement) [16] can be performed on the resulting Java programs. These transformations preserve the order in which floating point operations are executed.

A proposal by Sun to modify the floating point semantics of Java will also allow the use of the \textit{fused multiply-add} instruction (\texttt{fma}), which performs a multiply-add as a single operation. The use of this instruction is essential to gain full performance from the POWER architectures. Sun’s proposal (as well as a modified proposal from the Java Grande Forum [10]) will also allow the chaining of floating point operations using the 80-bit floating point registers of the Intel Pentium microprocessor. This is essential to gain full performance on the IA32 and IA64 architectures. (The current Java standard requires rounding to 64 bits after each floating point operation. This requires, on the IA32 architecture, an expensive conversion procedure for each computed value.) This relaxation of Java’s floating point model introduces more variability in the outcome of floating point computations, but does not reduce their accuracy.

It is still debated whether compilers should be allowed, in certain cases, to pursue more aggressive optimizations. Such optimizations, that take advantage of distributivity or associativity, are often used in current optimizing compilers. Their use may reduce accuracy, and their contribution to performance is usually secondary to the contribution of the other, already enabled, optimizations. Thus, the current Java floating point model should not preclude aggressive optimization of many loop nests.

2.6 Libraries

Compiler optimizations do not always give maximal achievable performance. Because resources are limited, programmers cannot tune their codes for every architecture. Both of these factors make libraries useful as a mechanism to utilize high quality implementations of standard mathematical operations (e.g., BLAS [6] and LAPACK [1]) in a portable fashion. As shown in our experimental results section, these libraries can be effectively implemented in 100% Java, bringing the benefits of Java to the implementation without significant performance losses. Because the standard Array package does not specify the layout of the array, a layout that provides the best performance for the libraries used can be chosen by the compiler. This layout can be row-order, column-order, or even a more advanced layout, such as recursive partitioning [7]. Different arrays can have different layouts.

Some other projects implementing high performance libraries in Java, or making high performance libraries available from Java, can be found in [2, 3, 4, 17].

3 Experimental results

In this section we present some experiments that illustrate the importance and effectiveness of the techniques discussed in Section 2. We use several benchmarks to quantify the performance of Java, with and without the previously mentioned optimizations. We always compare the performance of Java with an appropriate reference. In some cases this reference will be the highly optimized ESSL library. In other cases it will be fully optimized Fortran code. We perform most of our experiments on an RS/6000 model F50. This machine has four 332 MHz PowerPC 604e processors and 1 GB of main memory. Peak performance on each processor is 664 Mflops. We also run some benchmarks on an RS/6000 model 590. This machine has a single 67 MHz POWER2 processor, 512 MB of main memory, and a peak performance of 266 Mflops.

3.1 Matrix multiply

Figure 5 shows our results with a Java implementation of the matrix-multiply operation \( C = C + A \times B \), when \( A \),
$B$, and $C$ are all $500 \times 500$ matrices. The experiments were conducted on an RS/6000 F50. The bar plain represents the 5 Mflops performance we get with a straightforward dot-product implementation of this operation, using the commercially available IBM HPCJ static compiler for Java. Applying our versioning optimization to this code, to eliminate run-time checks, improves performance to 11 Mflops, as shown by the no check bar. Moreover, we can now apply to the safe region, created by versioning, the same high-order loop transformations that the Fortran compiler uses: tiling, unroll-and-jam, and scalar replacement. This brings the performance up to 173 Mflops, reported in the optimized bar. We finally enable the fma instruction of the PowerPC, pushing the performance to 234 Mflops (fma bar). We note that the Fortran compiler achieves 265 Mflops from the same straightforward code, through completely automatic application of the high-order loop transformations. ESSL (as shown in the ESSL bar) achieves 289 Mflops.

![Figure 5. Matrix-multiply in Java and ESSL.](image)

We also use a highly optimized version of matrix-multiply in our implementation of dgemm for the Java Array package BLAS library. (The dgemm function computes $C = \beta C + \alpha A \times B$, where $A$, $B$, and $C$ are matrices and $\alpha$ and $\beta$ are scalars.) In fact, the BLAS library can exploit the four processors in an RS/6000 F50 by partitioning the matrix-multiplication among several threads. This parallelization is done entirely within the library and is transparent to the application programmer. Figure 6 shows the performance of our Java dgemm on an F50, for different matrix sizes. The circles show the performance when a single thread (and therefore a single processor) is used. The crosses show the performance with four threads (and therefore four processors). We observe that for large matrices we achieve approximately 900 Mflops, with a good speedup over the single-threaded case. In at least one case ($150 \times 150$ matrices), we do better than 1 Gflops. As a reference, the performance of SMP ESSL is approximately 1200 Mflops with $500 \times 500$ matrices.

![Figure 6. Parallel DGEMM in Java.](image)

### 3.2 Complex arithmetic

Numerical codes with complex arithmetic represent another challenge for Java optimization. As described in Section 2.3, the cost of object creation and garbage collection for all intermediate results places a heavy toll on program execution. We analyze the performance of Java on computations with complex numbers using two benchmarks: MICROSTRIP and CFD. MICROSTRIP [12] computes the electric potential field of a microstrip structure in the presence of sinusoidal voltages. It uses an iterative Jacobi solver for the PDE defining the field. The microstrip structure is discretized by a $1000 \times 1000$ grid. CFD is a computational fluid dynamics kernel. It performs convolutions on three pairs of two-dimensional functions. Each function is represented as a $256 \times 256$ array of complex entries.

Figure 7 shows the results for our experiments with MICROSTRIP and CFD on an RS/6000 590. For each benchmark we report achieved Mflops for three implementations: (i) A Java implementation using Java arrays of Complex objects and the bounds checking optimization (no check bar); (ii) A Java implementation with true multidimensional arrays of Complex (from the Array package) and semantic inlining (semantic bar); and (iii) A Fortran implementation compiled with the highest level of optimization (Fortran bar).

Although a straightforward Java implementation results in dismal performance for MICROSTRIP and CFD (120
and 40 times slower than Fortran, respectively), the use of the `Complex` array class with semantic inlining improves things significantly. We were able to achieve 60% and 90% of the best Fortran performance with Java, for MICROSTRIP and CFD respectively. (The `fma` instruction is not an issue in this case, as these benchmarks do not take advantage of it.) These results show that semantic inlining is a powerful optimization technique. In particular, we can deliver Fortran-like performance for Java numerical codes with complex arithmetic. The concept of semantic inlining also applies, for example, to classes that support decimal arithmetic, interval arithmetic, and strings.

### 3.3 Data mining

Our last example illustrates our experience with developing a production-quality data mining application in Java [14]. The application is a recommendation code for suggesting new products to customers based on their previous spending behavior. The computational algorithm involves operations (including matrix-multiply) between two sparse matrices: an affinity matrix $A$ and a spending matrix $S$. Matrix $A$ has size $10330 \times 2103$ and 397, 559 non-zeros (1.8% of non-zeros). Matrix $S$ has size $4800 \times 2103$ and 231, 194 non-zeros (2.3% of non-zeros). Performance results for different implementations of this recommendation code are reported in Figure 8. The Mflops numbers reported correspond to actual computations with only the non-zeros elements.

The `plain Java` bar shows the 21 Mflops achieved with a straightforward Java implementation using Java arrays to represent the $A$ and $S$ matrices. If the run-time checks in this code are optimized away (a difficult task for the compiler, because of subscripted subscripts typical of sparse codes), the performance rises to 37 Mflops, as shown in the `no check Java` bar. By using the Java Array package and the Java BLAS library to represent and operate on these matrices, performance jumps to 109 Mflops (the `optimized Java` bar). An equivalent Fortran code does only slightly better, achieving 120 Mflops as shown in the `Fortran` bar. (The Java version did not use `fma`, whereas the Fortran version did.)

This data mining application is quite amenable to parallelization. We report some results from two parallel Java versions of this application in Figure 9. The implicitly parallel version is identical to the sequential version of our previous experiment, except that we enable parallelization inside the Array package and BLAS library. The explicitly parallel version is actually coded to make use of multiple threads at the application level.

As expected, the explicitly parallel version does better than the implicit version, achieving a speedup of 3.2 on 4 threads. Nevertheless, the implicit version delivers respectable performance, a speedup of 2.7 on 4 threads, with no effort on from the application programmers. This shown that the parallelism inside the Array package and BLAS library can indeed be beneficial at the application level.

### 4 Conclusion

This paper has shown that it is possible for Java to be competitive with Fortran for implementing numerically intensive applications. The necessary technologies include: (i) new compiler techniques, such as semantic inlining and bounds checking optimization; (ii) high-order loop trans-
deployments, already present in some Fortran compilers; and (iii) good quality numerical libraries. Some small language modifications (i.e., allowing the use of \texttt{fma}) can also be beneficial.

The activities of organizations like the Java Grande Forum in proposing standard classes and language enhancements is of great importance to the development of high-performance implementations of Java.

Because of its potential for high performance, its modern object oriented features, its portability, and the expanding base of Java programmers, we feel that Java will become an increasingly popular platform for the development and deployment of numerical applications.

References


