Suitability of Java for Solving Large Sparse Positive Definite Systems of Linear Equations using Direct Methods

by

Shea Armstrong

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2004

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The purpose of the thesis is to determine whether Java, a programming language that evolved out of a research project by Sun Microsystems in 1990, is suitable for solving large sparse linear systems using direct methods. That is, can performance comparable to the language traditionally used for sparse matrix computation, Fortran, be achieved by a Java implementation. Performance evaluation criteria include execution speed and memory requirements. A secondary criterion is ease of development.

Many attractive features, unique to the Java programming language, make it desirable for use in sparse matrix computation and provide the motivation for the thesis. The 'write once, run anywhere' proposition, coupled with nearly-ubiquitous Java support, alleviates the need to re-write programs in the event of hardware change. Features such as garbage collection (automatic recycling of memory) and array-index bounds checking make Java programs more robust than those written in Fortran.

Java has garnered a poor reputation as a high-performance computing platform, largely attributable to poor performance relative to Fortran in its early years. It is now a consensus among researchers that the Java language itself is not the problem, but rather its implementation. As such, improving compiler technology for numerical codes is critical to achieving high performance in numerical Java applications.

Preliminary work involved converting SPARSPAK, a collection of Fortran 90 subroutines for solving large sparse systems of linear equations and least squares problems developed by Dr. Alan George, into Java (J-SPARSPAK). It is well known that the majority of the solution process is spent in the numeric factorization phase. Initial benchmarks showed Java performing, on average, 3.6 times slower than Fortran for this critical phase. We detail how we improved Java performance to within a factor of two of Fortran.

Acknowledgements

I would like to thank my supervisor, Dr. Alan George, for his invaluable assistance, advice, and financial aid. I would also like to thank my parents, Bruce and Connie, for believing in me and always being there.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Since inception, the Java platform has been plagued with the perception of poor efficiency, especially when it comes to scientific or engineering applications, characterized by large-scale floating point computations. A recurring task in these applications is that of solving systems of linear equations. In this thesis we explore the suitability of the Java programming environment for this task. Ability to execute at a speed on par with Fortran (the language traditionally used for such computations) is our primary metric. Determining suitability is not as simple as running benchmarks and arriving at a 'yes' or 'no' answer, as the performance difference, and other relevant characteristics will each hold different weight depending on the particular application. We hope to give the reader information such that he/she will be able to make an educated decision as to whether the benefits of Java outweigh the potential pitfalls. We suspect that the reader will be surprised with the performance of Java relative to Fortran, as it has significantly improved since Java was first deployed and not much information has been published in recent years regarding this topic.

In this chapter, we will provide an introduction to both Java, and what is involved in solving sparse systems of linear equations using direct methods. Although it is customary for the thesis motivation and goals etc. to appear in the first chapter, we feel it necessary to introduce Java, the application

we intend to solve in Java, and the current state of Java and numerical computing first. As such, the motivation has been delayed until Chapter 3.

Chapter 2 will cover the current state of Java and numerical computing - including solutions proposed by various researchers. Much work has been done on determining suitability of Java for high performance computation, with performance being the key issue preventing Java from large scale adoption as a high performance computing platform. As such, considerable work has focused on ways to improve performance, which will also be covered in Chapter 2.

As mentioned, Chapter 3 presents the motivation, goals, objectives, and provides a detailed outline of the remainder of the thesis.

1.1 Introduction to Java

Java is a modern object-oriented programming language which evolved from a research project at Sun Microsystems™ in 1990. Java has garnered attention from both industry and academia alike due to its easy-to-learn nature, powerful object model, ease of portability, advanced memory management facilities, and multithreading capabilities. In this section we will describe the Java platform in greater detail.

The Java programming language is very user-friendly; its use in introductory programming courses is testament to this. User-friendliness is achieved, in part, through facilities which catch common programmer mistakes. For example, attempting to write an array element outside the bounds of an array causes an *ArrayIndexOutOfBoundsException* to be thrown. No memory write occurs, and the execution environment (typically a Java Virtual Machine - herein referred to as a JVM) notifies the programmer not only that an illegal write was requested, but indicates the source file and line number that attempted the illegal write. Figure 1.1 contains a simple Java program and resulting execution output. The program creates an array of 3 integers, and then (attempts) to print the first four elements of the array. The first three

```
public class test {
  public static void main(String args[]) {
    int arr[] = {1, 2, 3};
    int upperBound = arr.length + 1;
    for(int i = 0; i < upperBound; i++)
      System.out.println(arr[i]);
  }
}
```

Output:

```
1
2
3
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
        at test.main(test.java:6)
```

Figure 1.1: Example of a Java program accessing an out-of-bounds memory location and the resulting output when executed.

print as expected, but attempting to access the fourth element generates the exception as indicated by the program output. In traditional programming languages such as Fortran and C, the memory write would occur unimpaired, causing unpredictable behavior.

Compilers for traditional programming languages such as Fortran output native machine code which can then be immediately executed. Java takes a different approach. Java source code is first compiled into intermediate, machine-independent bytecode (*class* files). Upon execution, a JVM transforms each bytecode into a native instruction, and executes the native instruction. The additional instruction translation step incurs overhead which degrades performance relative to traditional languages.

In an attempt to lessen the impact of instruction interpretation, core to modern JVM's is a *Just-In-Time* (JIT) compiler which translates frequently executed methods into native code at runtime, allowing the compiled code to execute at raw machine speed, and eliminating the need for subsequent interpretation. Some refer to "Just-In-Time" as "Better-Late-Than-Never". Time required for program execution is now the sum of the time for compilation plus the time for execution.

It is possible to limit a JVM to execute a subset of the normal operations, a potential use of which includes improving safety for the host computer by preventing Java programs from writing to sockets or modifying disk files. This 'sandbox' environment is typically used when executing applets (embedded programs in web browsers).

Java is cross-platform compatible at both the source and bytecode level. The "write-once, run-anywhere" characterization is commonly used to describe such behavior. Cross-platform compatibility is especially important for high-performance applications where the hardware architecture generally has a significantly shorter lifespan than the software application [12]. Porting a Java program from one machine to another is as simple as copying bytecodes (or similarly, copying the source codes and recompiling). Reproducibility is achieved even when GUI's are employed by including GUI support in the Java Application-Programmer Interface (API) (i.e., Swing and AWT). Sun has essentially signed a contract guaranteeing *bit-by-bit* reproducibility over all platforms. Such a guarantee comes at a price, as we will see later.

When working with languages such as Fortran, a programmer is required to keep track of all dynamically allocated memory and is responsible for releasing the memory when it is no longer needed. Memory leaks occur when the programmer forgets to release a chunk of dynamically allocated memory. Java relegates the task of freeing dynamically allocated memory when it is no longer needed to a "garbage collector" (a separate thread running in the JVM). In doing so, application debug time is decreased. Furthermore, much cleaner code is produced as the code is not cluttered with memory

management statements [9]. Efficiency of modern garbage collectors is almost entirely attributable to a huge engineering effort by both industry and academia alike.

It should also be noted that since Java has been popular for use in introductory programming courses at post-secondary institutions, a large school of skilled Java programmers is available. Clearly, more programmers increases the attractivness of the language to developers deciding what language to adopt for a project.

All variables in Java, with the exception of intrinsics, (i.e., all objects) are references. An implication being it is impossible to inadvertently make a copy of a large object. In languages such as Fortran and C++, programmers have the option of passing deep copies of large objects and data structures into methods, a dangerous feature in the hands of an unskilled programmer. Java's approach effectively eliminates this potential source of performance degradation.

As mentioned, early Java implementations interpreted bytecode, and performance relative to Fortran often lagged by two orders of magnitude. Such dismal early figures resulted in Java acquiring a bad reputation for scientific and engineering applications demanding high performance. This will be explored further in Chapter 2.

1.2 Introduction to Sparse Matrix Computation

In this section, we will describe what solving large sparse symmetric positive definite linear systems using direct methods entails. The solution process is composed of a number of phases; each of which with varying computational demands. We will describe both what a sparse symmetric positive definite linear system is, and what it means to solve them using direct methods. Both of these concepts will be explained at a high level as they are not the main focus of the thesis.

1.2.1 Notation

Defintions from [11] will be adopted. Matrices will be represented by uppercase, bold, italic letters (such as $\boldsymbol{A}$). Vectors will be represented by lowercase, bold, italic letters (such as $\boldsymbol{x}$). $\boldsymbol{\Omega(A)}$ is the set denoting the nonzero subscript pairs in $\boldsymbol{A}$ and $\boldsymbol{\Omega(x)}$ is the set denoting the nonzero subscripts in $\boldsymbol{x}$. Let $\boldsymbol{\eta(v)}$ denote the number of nonzero elements in $\boldsymbol{v}$. The density of $\boldsymbol{v}$, denoted by $\boldsymbol{\delta(v)}$, is the ratio of the number of nonzero elements to the total number of elements and is thus

$$\boldsymbol{\delta(v)} = \frac{\boldsymbol{\eta(v)}}{n^2} \tag{1.1}$$

where n is the number of components of $\boldsymbol{v}$. Let $|S|$ denote the cardinality of the set S. Thus $|\boldsymbol{\Omega(A)}|$ represents the number of nonzeroes in the matrix $\boldsymbol{A}$.

1.2.2 Overview of the Problem

We are concerned with solving the $n \times n$ system of linear equations

$$\boldsymbol{Ax} = \boldsymbol{b} \tag{1.2}$$

(i.e., given a matrix $\boldsymbol{A}$ and a vector $\boldsymbol{b}$, determine a vector $\boldsymbol{x}$ which satisfies equation 1.2). Approaches used to determine $\boldsymbol{x}$ fall into two broad categories, iterative methods and direct methods. We will limit our study to direct methods. Although they share the same goal, the two approaches are largely orthogonal in nature from a computational perspective. For more information on iterative methods, see [8]. We are interested in the case where $\boldsymbol{A}$ is sparse, symmetric, and positive definite.

In solving a system of equations using direct methods, some variant of Gaussian elimintion is applied yielding a triangular factorization of the matrix (i.e., $\boldsymbol{A} = \boldsymbol{LU}$ where $\boldsymbol{L}$ and $\boldsymbol{U}$ are lower and upper triangular respectively). After factorization, the solutions of the two triangular systems $\boldsymbol{Ly} = \boldsymbol{b}$ and $\boldsymbol{Ux} = \boldsymbol{y}$ are computed to get $\boldsymbol{x}$. In computing the triangular

factorization $\boldsymbol{A} = \boldsymbol{LU}$, *fill* is introduced. By this we mean elements which are zero in $\boldsymbol{A}$ can be nonzero in $\boldsymbol{LU}$. This is not a problem when $n \times n$ storage has been allocated for the factor $\boldsymbol{LU}$ (a strategy often employed when $\boldsymbol{A}$ is dense). However, we are concerned with the case where $\boldsymbol{A}$ is sparse. A sparse matrix is one in which the majority of its elements are zero. When a matrix is sparse, a reduction in both memory usage and computational requirements can be achieved by storing, and operating on, only the nonzero elements.

If $\boldsymbol{A}$ is symmetric ($\boldsymbol{A}_{ij} = \boldsymbol{A}_{ji}$) and positive definite ($\boldsymbol{x}^t \boldsymbol{A} \boldsymbol{x} > 0$ for all vectors $\boldsymbol{x}$ where $\boldsymbol{x} \neq 0$), the factorization will have certain properties which can be exploited for additional performance gains. Cholesky showed that symmetric Gaussian elimination results in the factorization $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^{\boldsymbol{t}}$ (that is, $\boldsymbol{U} = \boldsymbol{L}^{\boldsymbol{t}}$). We refer to $\boldsymbol{L}$ as the *Cholesky factor*. We reduce our storage requirements by half by storing only the Cholesky factor $\boldsymbol{L}$. More important, factorization now involves computing one-half the number of entries as the general case.

In the general case, some form of *pivoting* is required to ensure numerical stability, often conflicting with the goal of minimizing fill. It is well known (see, for example, Forsyth and Moler [17]) that when $\boldsymbol{A}$ is symmetric positive definite, numerical stability is no longer an issue, allowing one to focus solely on minimizing fill.

To summarize, our task differs from naive Gaussian elimination of a general system in the following fashion. Our system is sparse, meaning that savings can be had if only the non-zero elements are stored. In storing only the non-zero elements, minimizing fill is now a concern. As such, an *ordering* which attempts to minimize fill incurred during the numeric factorization phase is first determined. Because we store only the non-zeroes during the solution process, and $\boldsymbol{\eta(LU)} \geq \boldsymbol{\eta(A)}$ (assuming no-cancellation), additional storage must be allocated prior to factoring $\boldsymbol{A}$ into $\boldsymbol{LU}$. The process of determining both how much, and the location of, additional required storage is known as *symbolic factorization*. Once the amount of storage required to

store $\boldsymbol{LU}$ is known, $\boldsymbol{L}$ and $\boldsymbol{U}$ are computed in the *numeric factorization* phase. Finally, the solution vector $\boldsymbol{x}$ obtained at by performing a *triangular solve* as described above.

Chapter 2

Java and Numerical Computing

2.1 Issues with Java and Numerical Computing

From James Gosling [3], a key player in the design of Java, "The original specification of floating point arithmetic in Java was done in a context where sophisticated high performance numerical computing was not an issue". As a result, Java contains many features that make the developers life easier at the expense of performance. In addition, it is missing some features which researchers have labelled as key for adoption as a numerical computing platform. In this section, we will discuss the current state of Java and numerical computing, including identification of both missing features and those present which are detrimental to high performance numerical computing.

First, we will provide some discussion on why these issues are relevant to our work. We suspect the performance gap between Java and Fortran for solving large sparse positive definite systems using direct methods will be similar to that for other high-performance computing applications such as those from the SciMark[1] benchmark suite. If so, performance-hindering characteristics of the Java programming environment identified by researchers work-

[1] http://math.nist.gov/scimark2/index.html

ing to improve the performance of Java for high-performance computing are likely to be relevant in our quest as well.

As mentioned in Section 1.1, early JVM implementations interpreted bytecodes in a relatively unsophisticated fashion. Figure 2.1 shows the evolution of virtual machine technology on a 333-MHz Sun Ultra 10. The SciMark 2.0 benchmark suite was used for the JVM comparison. The benchmark suite consists of a number of computational kernels, including a one-dimensional FFT, Jacobi Successive Over-relaxation, Monte Carlo integration, sparse matrix multiply, and dense LU matrix factorization. As of Java 1.3, vendors have been allowed to provide their own implementations of elementary functions (i.e., sin, cos etc.) under the restriction that results can differ by at most one unit in the last place of the correctly rounded result [14]. Figure 2.1 makes it clear that continued advances in JIT compilation coupled with further relaxation of Java floating point rules are crucial to increasing viability of Java as a numerical computing platform.

Figure 2.2, extracted from [14], further solidifies dependence of Java numerical code performance on the JVM. It shows up to a 20-fold performance difference of the SciMark 2.0 benchmark across varying hardware. JVM's available for PC's are typically much more evolved than those available for high-end workstations. The JVM PC userbase is much greater than the userbase for high-end workstations. As such, more software development effort has been devoted to the PC JVM's than those for high-end workstations[2].

Java does not contain intrinsic complex number support, a feature many say is required for numerical computations. Ideally, arithmetic with complex numbers would be supported as efficiently as that with numbers of type *float* and *double*. Although not intrinsically part of the language, complex number support can be added as follows.

A *Complex* class can be created, which contains two members of type

[2]The latest JVM version for the system we initially intended to use in our benchmarking was 1.1.4. We thus switched to the Win32 platform at the cost of having to find a decent Fortran compiler for that target.

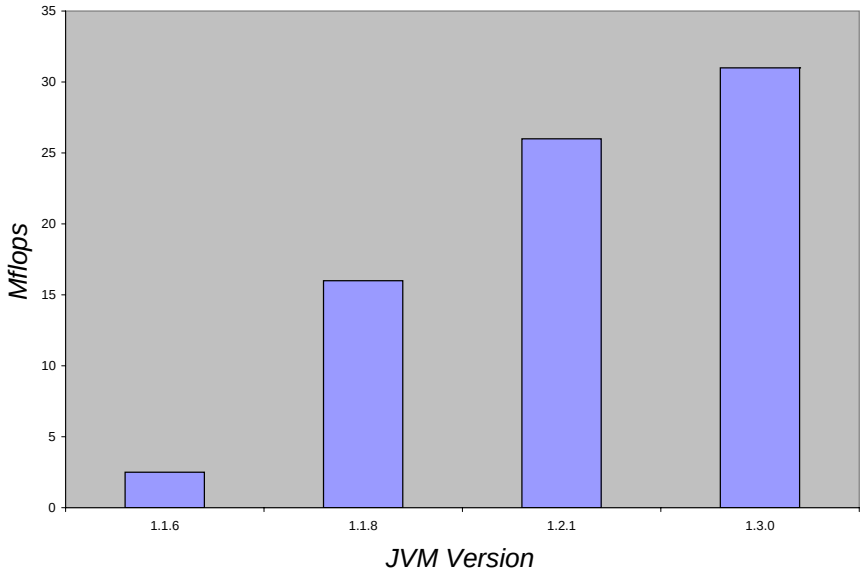

Figure 2.1: Evolution of Java performance for the SciMark benchmark on a single platform: a 333-MHz Sun Ultra 10 [14]

double representing the real and imaginary parts respectively. The class will contain methods which support operations on complex numbers such as addition, multiplication etc.. Figure 2.3 gives an example as to how this class could be used. The problems with this approach are as follows. Every object in Java, in addition to memory consumed by class members, contains a 16-byte object descriptor, bringing the amount of memory required for an instance of our class to 32 bytes (twice the amount consumed by a complex number in Fortran 90). Secondly, evaluating expressions such as the one in Figure 2.3 results in the creation of an unnecessary temporary (a new complex object is created by the operation `a.times(b)`). Moreira et al. in [15] showed

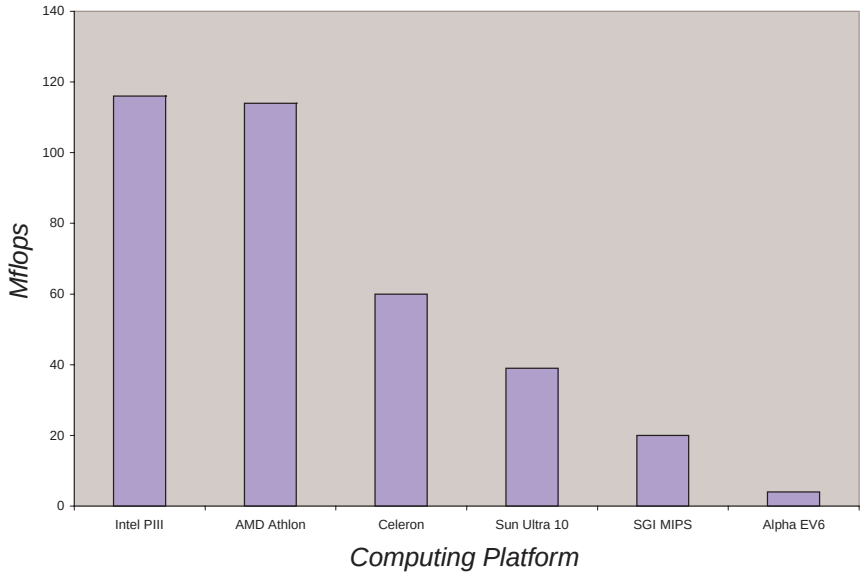

Figure 2.2: Java performance over a number of hardware platforms [14]

that computations involving complex numbers in this representation can be up to 100 times slower than Fortran 90. This is due to both the explicit cost of excessive object creation and the collateral effect of increased stress on the garbage collector.

```
Complex a = new Complex(1.0,2.0);
Complex b = new Complex(3.0,4.0);
Complex c = new Complex(4.0,5.0);
Complex d = a.times(b).plus(c)
```

Figure 2.3: Usage of user-defined complex numbers in Java

The syntax for manipulating complex numbers represented in this format is tedious and unintuitive. Ideally, we would like to be able to use the same familiar syntax used to manipulate intrinsic arithmetic data types such as *int* and *double*. In order to provide the same convenience to programmers as Fortran 90, the community claims that Java must include intrinsic complex number support, and operator overloading for easier manipulation of user-defined data types.

The Java Language Specification states that exceptions are *precise*:

> "Exceptions are precise: when the transfer of control takes place, all effects of the statements executed and expressions evaluated before the point from which the exception is thrown must appear to have taken place. No expressions, statements, or parts thereof that occur after the point from which the exception is thrown may appear to have been evaluated. If optimized code has speculatively executed some of the expressions or statements which follow the point at which the exception occurs, such code must be prepared to hide this speculative execution from the user-visible state of the program" [18]

Such a model effectively precludes most compiler optimizations which modify instruction order. Almost all optimizations traditionally applied to numerical programs modify instruction order [15]. Two pre-requisites for re-ordering instructions are as follows: (1) The section must be free of any exception (i.e., an *array-bounds* or *null-pointer* exception), and (2) instruction execution dependence must be preserved. By that we mean all writes to a datum must be kept in order, no write of a datum be moved prior to a read of the datum, and finally, no read of a datum be moved after a write to that datum [15]. Obviously, core to verifying the three aforementioned criteria is determining which datum's are 'the same'. In Fortran 90, variables are distinct if they have different names. In the case of arrays, two array elements are distinct if they are pointed to by different coordinates. In Java the

situation is exacerbated by what is called *array aliasing*. For example, in Figure 2.4, `A[3]` and `A[4]` reference the same array. As such, instruction execution dependencies are much more difficult to calculate in Java.

Another issue identified by researchers is Java's lack of true multidimensional arrays. The Java language specification does not directly support arrays with dimension greater than one. A 'two-dimensional' array in Java is in fact an array-of-arrays. Numerical programs do not need the flexibility provided by Java's array structure. In fact, it is a hinderance. Figure 2.4 depicts a legal Java array structure. The array object descriptor contains information such as the length of the array, and information contained in all Java objects such as a lock bit, bits used by the garbage collector and type information etc. Note that there is no guarantee that a two-dimensional array in Java has a rectangular structure. Finally, Java's array structure does not guarantee rows are stored contiguously, thus sacrificing locality of reference. An unfortunate implication of this being traditional blocking algorithms which enjoy great popularity on Fortran platforms will generally not provide any benefit in Java; thus, direct conversion of numerical libraries from Fortran to Java should be done with great care.

As mentioned in Section 1.1, every array access is checked prior to being executed with an *ArrayIndexOutOfBoundsException* thrown in the event the access is illegal. The problems with this are two-fold. There is a cost associated with making sure every array index is in bounds (two integer comparisons). Secondly, as we have seen, any potential exception-generator is at odds with compiler optimizations which are necessary for high performance in numerical codes. One important research area to improve Java performance has dealt with identifying sections of code which are guaranteed not to throw an array-out-of-bounds exception. If one can prove a section of a Java program will not raise an array-out-of-bounds exception, it can be executed without checks (saving both the explicit cost, and bringing us one step closer to legalizing otherwise illegal compiler optimizations). In the general case, removal of array bounds checks is a very interesting (or annoying)

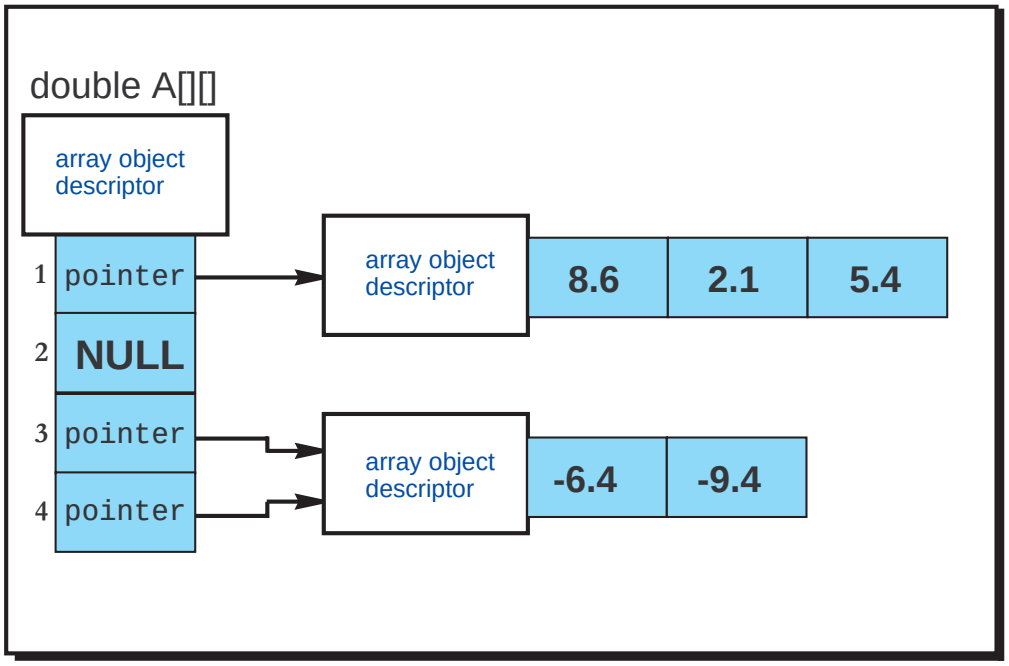

Figure 2.4: Visualization of a legal Java array construction

problem. We will present a scenario demonstrating this.

Imagine a section of code that indexes an array with an arbitrary element from another array (i.e., `A[b[i]]`). Removing the bounds check for `A` involves proving at runtime that the minimum value in `b` is greater than or equal to zero, and the maximum value is less than or equal to the length of `A`. This is referred to as *indirect* array addressing. Further complicating the issue is the possibility that a concurrently executing thread has a reference to `b`, at which point we must concede modification of `b` could occur *at any point during the execution of the program*. For more information on this problem, and on potential solutions, see [20].

The benefit obtained by removal of array-bounds checks is dependent on both the application and the compiler. A program with many array accesses (e.g., a scientific computing application) has much to gain from

array bounds check removal. Compiler dependence results from the level of optimization which can subsequently be performed once exception-free regions are identified. Reported performance improvements have ranged from 15% (Blount et. al. in [13]) to 15 times that of code with array bounds checks (Moreria et. al. in [16]). One must be skeptical when evaluating practical merit as most reported improvements indicated benefit achieved by simple (illegal) removal of array bounds checks. In an environment conforming to Java language semantics, the true benefit is determined by discounting the cost associated with legal removal of the checks in conjunction with cost saved.

Java's implementation of multi-dimensional arrays causes further anxiety when attempting to eliminate array bounds checks. Proving access of `A[i][j]` is legal requires determining the minimum and maximum values that `i` and `j` can take followed by checking to see if those values are within the bounds of the array. If `A` were rectangular, this check would be relatively trivial. Since we cannot guarantee that `A` is rectangular, the check involves iterating through each row of the array, requiring $O(n)$ operations. If `A` were rectangular, the check could be performed in $O(1)$.

In addition to performing array bounds exception checks, the Language Specification mandates that every referenced object be non-null. As such, indexing element `A[i][j]` requires first determining that `A` is non-null, retrieving `A[i]`, verifying `A[i]` is non-null, and finally retrieving `A[i][j]`.

In summary, we have identified many issues which stand in the way of Java's adoption as a numerical computing platform. Some of these issues are mere inconveniences (i.e., operator overloading), whereas others are much more serious (those which prevent ever-important compiler optimizations). In the next section, we will identify and discuss proposed solutions to some of the problems identified in this section.

2.2 Proposed Solutions to Java and Numerical Computing

Although this section might appear to be "a related work" section, recall that our goal is to determine suitability of Java for a particular type of computation and not to improve performance of Java, which is the goal of the majority of the works presented in this section. Summary of research more directly related to ours (i.e., determining suitability of Java for high performance numerical computing) can be found in Section 3.1.

Moreira et. al in [19] performed a comparison of Java versus C and Fortran for numerical computing. The study is slightly outdated (1998), and the Java source code seems to have been compiled into native machine code in the same fashion as Fortran and C instead of being interpreted. However, as we will see, the conclusions they draw are pertinent to our work. They compared results from a simple matrix-matrix multiplication routine; 100% pure Java ran at 2.2 Mflops, C ran at 137.6 Mflops, and FORTRAN achieved 205.4 Mflops. The remainder of the paper explained why 100% pure Java ran two orders of magnitude slower than Fortran. Removing the run-time checks from their Java compiler (i.e., removing null reference and array bounds checks) improved performance to 33.3 Mflops, much better than before but still not close to C or FORTRAN. The performance boost was partially due to removal of the explicit cost associated with the checks, and partially due to compiler optimizations which became possible once the possibility of an exception being thrown was eliminated.

Removing the runtime checks is not strictly legal, but was done to show the associated cost. However, the authors recognized that if they could identify guaranteed exception-free regions of code, they could achieve similiar performance figures while maintaining conformance with the Java language specification. To accomplish this, a technique known as *versioning* is employed. Refer to the example in Figure 2.5. If the *if* statement on line one evaluates to true, the loop is guaranteed exception free and traditional

```
if(A != null && p <= A.length) {
  // This version is safe and can be executed without runtime checks
  for(int i = 0; i < p; i++)
    System.out.println(A[i]);
} else {
  // This version is unsafe
  for(int i = 0; i < p; i++)
    System.out.println(A[i]);
}
```

Figure 2.5: Example of versioning

compiler optimizations can be applied. If it evaluates to false, the exception-raising version of the loop is executed. The paper does not clarify whether the compiler identifies exception-free sections of code, or whether it is up to the programmer. If it is left up to the programmer, some mechanism instructing the compiler as to which sections of code were exception free would be necessary. Also, that would open the possibility for the programmer to incorrectly specify a region of code exception-free, causing the type of unpredictable results typical of less-robust languages such as Fortran and C. Clearly, the merit of versioning depends on automatic identification of these regions. Checking for non-null references is trivial, but guaranteeing all array accesses are in bounds is not as was demonstrated in Section 2.1.

Continuing with discussion of the Moreira paper, in an attempt to further understand the cause of poor Java performance, they systematically disabled features of the C compiler to the point where C performance equaled Java performance (33 Mflops). They disabled the use of the FMA (fused multiply-add) instruction, loop unrolling, instruction reordering, and used Java-like arrays (i.e., `double* A[m]` instead of `double A[m][n]`). Re-enabling these features one by one, they were able to determine the performance benefit associated with each. Switching back to true rectangular arrays increased per-

formance to 44.2 Mflops. Re-enabling the FMA instruction boosted throughput to 64.2 Mflops, and a final 2-fold performance increase to 137.6 Mflops resulted from allowing the compiler to unroll the loops and perform instruction re-ordering. Clearly, legalizing these features for Java is critical in boosting Java performance to the level of C and Fortran.

NINJA

The NINJA group is part of IBM research and is working to make Java competitive with Fortran and C++ in the domain of technical computing. One project to emerge from their research was the *IBM Array Package*. The goal of the array package was to add true multi-dimensional array support to Java, and improve complex number performance. To guarantee a multi-dimensional array occupies a contiguous section of memory, they encapsulated a one-dimensional array in a class, and added methods to simulate multi-dimensional array support. A downside of this approach being the total number of elements in any rank array is $2^{31} - 1$ (i.e., the maximum of elements allowed in a one-dimensional array). An n-dimensional true Java array can hold up to $n \times 2^{31} - 1$ elements. Once instantiated, the rank and dimensions of a multi-dimensional array can not change, improving ease of compiler optimizations.

Encapsulating a multi-dimensional array as a one-dimensional array comes at the price of a method call for each *get* or *set* operation. Modern JVM's which implement method inlining effectively nullify the cost of the method call at the expense of actually inlining the methods. Taking the concept of general inlining a step further, the NINJA group created a non-conforming compiler and JVM implementing a technique they refer to as *semantic inlining*. Semantic inlining treats calls to known methods on known datatypes as language primitives. Semantic inlining removes the cost of known method calls at compile time as opposed to run-time (as is the case with conventional modern JVM's). In addition, it allocates known objects on the stack as opposed to the heap whereever possible, effectively eliminating the problem of

temporary object creation illustrated in Figure 2.3.

To test array package performance, the NINJA group benchmarked it versus a 100% pure Java implementation (without the array package), and Fortran compiled with maximum optimizations. Over a wide range of numerical benchmarks, 100% pure Java ran, on average, at 1.5% the speed of Fortran. The array package coupled with the compiler supporting semantic inlining ran at 73% of Fortran. The figure achieved with the array package is especially impressive since it conforms entirely to the Java language specification. The downside to this approach is that performance improvements require explicit semantic knowledge of certain classes and methods.

Conclusion

We presented two works in the section. The first diagnosed why Java runs slower than Fortran and C for numerical computing applications. Additionally, it presented a few solutions. The second presented a solution to the lack-of-true-multidimensional-arrays problem. Compiler optimizations can be legalized by creating two versions of critical code sections, one of which is guaranteed to be exception free; a technique known as versioning. To aide identification of exception-free regions of code, a collection of immutable, rectangular multi-dimensional array classes was created. Semantic inlining was used to eliminate the cost of the extra method calls associated with use of the multi-dimensional array classes. In addition, semantic inlining improved complex number performance by allocating objects on the stack whenever possible. Use of the aforementioned techniques improved Java performance to within 73% of Fortran.

Some of the techniques presented in this section require a compiler with instrinsic knowledge of certain classes, adding to both compiler size and complexity. Given that the techniques are primarily aimed at improving performance of numerical codes, it might not be feasible to include them in a general-purpose Java compiler, such as the one available from Sun. Ideally, performance-improving techniques should be general enough that they can be

applied to a wide-range of Java programs, rather than providing individual solutions for different 'types' of computation. In the following section, we will discuss the latest compiler technology employed in todays fastest Java Virtual Machine.

2.3 Java Compiler Technology

In this section, we will present an overview of Java compiler technology. As mentioned in the introduction, the transition from Java source code to actual program execution is a two-step process. First, the source code is compiled into a JVM-targetted, platform-independent bytecode. A JVM then executes the bytecode.

As demonstrated, compiler technology plays a substantial role in Java performance. We benchmarked the most popular JVM's early on in the project with various numerical benchmarks for Java and discovered the Sun JVM with the -server option to be the best performer. Obviously, the JVM yielding the best performance should be the one used in our tests, since performance is our primary criterion. Ideally we would like to determine why this JVM outperforms the others. Accomplishing this would require the ability to selectively toggle compiler optimizations on and off. The Sun JVM does not provide that functionality, nor is the source code available. Sun releases documentation stating what optimizations their VM performs, but not how or to what degree. Namely, for certain optimizations, it is not known how Sun overcomes the hurdles identified in section 2.1. We realize it is very unsatisfying to present a problem, saying that a solution exists, but not explain it. Given that we must use the top-performing JVM for numerical computations in our study, we see no work-around. In this section, we will present high-level details of the Sun JVM.

Much of the information to follow in this chapter has been extracted from the Java HotSpot Virtual Machine v.1.4.1 Whitepaper [24]. A few challenges facing creators of optimizing Java compilers were highlighted in

section 2.1 (e.g., the strict exception model). The whitepaper identified more complications which will also be covered in this section.

Additional Hurdles

A popular compiler optimization whereby a method call is replaced by the actual body of the method called is known as *method inlining*. Programs to which method inlining has been applied benefit on two fronts. First, the cost associated with the actual method call is saved. More importantly however, following inlining the optimizing compiler has much larger blocks of code to work with, increasing effectiveness of other optimizations.

The majority of methods in Java programs are *virtual* (i.e., the actual method invoked is unknown at compile-time). This has the effect of both increasing the cost of method calls in Java (virtual method calls are more expensive than non-virtual), and inlining methods in Java programs substantially more difficult.

The Java Hotspot Virtual Machine

Core to modern JVM's is a JIT compiler, which translates frequently executed methods into native code at runtime (as opposed to compilers for traditional languages which generate native code prior to runtime). All methods are interpreted the first time they are called. Recognizing that most programs spend the majority of their execution time in a minority of their code, the Java Hotspot VM attempts to limit optimizations to these "hotspots". By interpreting infrequently executed code, the JIT compiler minimizes time spent compiling while still optimizing critical locations. The hotspot analyzer runs for the life time of the program, adapting to changing program behavior as necessary.

The Sun JVM package actually includes two virtual machines, the *client* virtual machine, and the *server* virtual machine. Both contain an optimizing JIT compiler. The *server* VM is intended to be used for long-running applications, such as those typically found in server environments. The server

VM is a fully optimizing compiler, applying almost all optimizations characteristic of compilers for traditional languages. Optimizations such as dead code elimation, loop invariant hoisting, common subexpression elimination, and constant propagation are applied. Futhermore, optimizations specific to the Java programming language such as array-index and null-check elimination are performed. Register allocation is performed by using a global graph coloring algorithm. The compiler performs method inlining, coupled with dynamic deoptimization, which we will explain following a description of the client compiler.

The *client* compiler performs very few global optimizations (which are typically the most expensive), but rather does peep-hole optimizations on an intermediate program representation generated from the bytecodes. Methods which are neither synchronized nor throw any exceptions are candidates for inlining.

To deal with the issue of being unable to inline virtual methods, the Sun JVM employs a technique referred to as *dynamic deoptimization*. An example best demonstrates how dynamic deoptimization inlines virtual method calls. Suppose an abtract class `Shape` existed, with an abstract method `getArea()`. There exists classes `square` and `circle` derived from `Shape`, both of which implement the `getArea()` method appropriately. If, during the execution of a program utilizing objects of class `Shape`, the hotspot compiler notices that every call to the `getArea()` method of a `Shape` object executes the `circle` version, the compiler will actually inline that version at the call site! In the event the `square getArea()` method should be called instead, the compiler *dynamically deoptimizes* the `circle getArea()` method, and calls the `square` one instead. In this fashion, the hotspot compiler is able to inline virtual method calls. It must still be realized that, even after a virtual method call is inlined, a check must be performed prior to executing any of the inlined code, to ensure the inlining is still valid.

Both the client and server VM's interpret each method first prior to any compilation. In doing so, a long-running interpreted method could cause seri-

ous performance degradation. In the event this occurs, the hotspot compiler saves method state, halts exeuction, compiles and optimizes the method, and continues execution at the halted point with the native as opposed to interpreted code. This technique is referred to as *On-Stack Replacement* (OSR).

In conclusion, we see that the hotspot server compiler performs many classical optimizations performed on statically-compiled languages, somehow overcoming issues created by the Java Language Specification. Given that the server compiler is a high-end fully optimizing compiler, it seems that the asymptotic performance of Java programs should approach that of traditional programs. Although certainly not the case in studies presented so far (most of which were 2+ years told), it will be interesting to see how the server compiler performs under our application. It will remain to be seen whether the toll of JIT compilation can ever be offset by potential performance gains over statically compiled languages.

On a final note, it is speculated by some that performance of languages employing JIT compilation with run-time analysis and the ability to dynamic optimize generated code will eventually surpass performance of traditional lanuages (e.g., under no circumstance can a static C++ compiler inline a virtual method call).

2.4 Thesis Outline

In Chapter 3, we present related work, dicuss thesis motivation, and discuss what we mean by suitability of Java for solving large symmetric sparse positive definite systems of linear equations using direct methods. Furthermore, we state how we are going to determine suitability. We also discuss what we hope to achieve by benchmarking.

Chapter 4 discusses issues overcome in the conversion of SPARSPAK into Java due to both language differences and access to numerical libraries. In addition, we discuss the problems we chose to benchmark SPARSPAK.

Chapter 5 presents initial results. With the aid of profiling, we identify

performance-critical areas of the solution process with the eventual goal of lessening the performance gap between Java and Fortran. In Chapter 6, we discuss our attempts to improve the performance-critical areas of the solution process identified in the previous chapter. And finally, in Chapter 7, we present our conclusions and identify areas for future work.

Chapter 3

Related Work, Motivation, and Methodology

3.1 Related Work

To the best of our knowledge, no prior study has been done evaluating Java for solving large sparse systems of equations using direct methods. In this section, we will report results of other researchers who have studied Java versus traditional languages such as Fortran and C for numerical or scientific applications. In addition, we will report any other research viewed as being applicable to ours. Our work differs in that the applications most researchers used to evaluate suitability of Java for numerical computing consist of small computational kernels – the computational requirements of SPARSPAK are much broader with stresses placed on system components often unstressed by computational kernels (i.e., out-of-cache memory accesses, disk I/O).

Saad et al. developed a benchmark suite intended to measure the performance of systems for sparse matrix computation. One decision they had to make was whether computational kernels can be representative of whole application performance. On the one hand, it can be argued that kernels do not adequately stress components commonly used by whole applications such as the hard disk and perhaps even main memory in cases where the kernel

dataset fits entirely in cache. However, they also note that to a performance analysis specialist, total application run time is useless. Relating to our work, in addition to stating wall-clock timing differences, we should attempt to determine computational kernels whose performance is representative of whole application performance. Successfully doing so will more precisely identify what (if anything) needs work to improve Java performance for solving large symmetric sparse positive definite systems using direct methods.

In [12], Bull et. al. performed a comparsion of Java versus C for scientific applications. They converted a subset of the Java Grande Benchmark into C and compared performance versus Java. The subset they converted consisted of:

- computing the first n Fourier coefficients of the function $f(x) = (x+1)^x$ on the interval [0,2]
- solving an $n \times n$ linear system using LU factorization followed by a triangular solve
- sorting an array of n integers using heapsort
- performing 100 iterations of successive over-relaxation (SOR) on an $n \times n$ grid
- performing a one-dimensional forward transform of n complex numbers
- performing sparse matrix-vector multiplication using a sparse matrix stored in compressed-row format with a prescribed sparsity structure

Running on PC's (Linux, Windows NT), Java executed a mean 1.23 times slower than C which they deem very respectable (we agree). They use a wide variety of JVM's, with no one JVM emerging as the clear winner. The following results compare the best Java result (over all JVM's) with the best C result. Relatively, they found the FFT to be slowest (2.26 times slower than C), and the Fourier series calcuation to be the fastest (finishing in 0.87

times the speed of C). The sparse matrix-vector multiplication was "middle of the pack", running at 1.30 times slower than C. They conclude that the performance gap between Java and C for scientific applications is no longer a major reason to choose C over Java.

Blount et. al. **??** converted a subset of the LAPACK (a high-performance Fortran 77 numerical library) into Java and labelled it JLAPACK. Their design was object oriented in nature containing classes such as `Vector` and `Matrix`. Their implementation attempted to keep data orthogonal to shape information, allowing one to choose the most natural view of the data (e.g., given a matrix, one could specify a view in the form of a new matrix such that every other element of the original matrix is an element of the new matrix – the same copy of the data is used, with different shape information). This feature is present in Fortran 77 and widely used throughout LAPACK. They cited absence of parametric polymorphism and operator overloading as being detrimental to their implementation effort; substantial code bloat and extra programmer effort was necessary. Their tests indicated that JLAPACK performed within a factor of three of LAPACK on most architectures. Disabling the JIT compiler made JLAPACK unusably slow.

Java contains a mechanism by which methods written in other programming languages can be executed with the results available to the calling Java environment. These methods are referred to as *native* methods [4]. When taken to the extreme, the entirety of a sparse matrix solver (such as SPARSPAK [5]) could be compiled and called from a Java program yielding "Fortran-like" performance under Java. In order to test both the performance of the Java Native Interface (JNI) and the dependence of JALPACK performance on the BLAS libraries, Blount et. al. replaced all calls to the Java BLAS routines with calls to the native BLAS routines. They reported a performance increase to within 15% of LAPACK for sufficiently large problem sizes. It would seem as though identifying hotspots in Java programs and replacing those hotspots with native code is a promising method of increasing Java performance – but one must keep in mind that a significant

portion of the benefits offered by Java are lost as soon as the JVM relinquishes control to any code outside its jurisdiction. Portability is much more difficult to achieve when the JNI is utilized - the shared library being called must be available on all platforms of interest. In addition, the shared library is likely to have slight differences from platform to platform, resulting in different outcomes [14]. Robustness is lost upon calling a native method as the calling JVM has no way to prevent out-of-bounds array indexing, object mis-casting, accessing of deleted objects, or any other form of memory corruption. These drawbacks directly conflict with our primary motivation for determining Java's worthiness as a sparse matrix computation tool and as such we will not consider the JNI in this thesis.

3.2 Motivation

So far, we have described the essence of Java, and the nature the problem we are trying to solve using Java, and the current state of Java and numerical computing. Given that background, we can now discuss the motivation of the thesis. When performing numerical computations, execution speed is a high priority and, given the choice, generally would not be sacrificed for programmer convenience. This begs the question, why are we attempting to coax Java into a role it was not designed for? In this section, we will address that question.

Firstly, how does our problem differ from work that others have done? The majority of papers which have evaluated Java's suitability for numerical computation have used a collection of popular kernels (sparse matrix-matrix multiplication, FFT etc.) and either reported the results in a standard measure such as MFlops, or compared the results to those obtained from executing the same kernels in a traditional language such as Fortran or C. The authors deem the results either satisfactory or not, and proceed to label Java as either suitable or unsuitable for numerical or scientific computation. Granted, it is possible (perhaps even likely), that performance of these

kernels is indicative of real-world performance of the majority of scientific applications. However, this clearly can not be verified due to the exceptionally large number of applications which fall into the category of either numeric or scientific. We are curious as to whether relative performance of Java to Fortran will hold when solving symmetric positive definite systems of equations using direct methods. In addition, there is a lack of papers addressing high-performance Java computing in recent years, the majority were published between the years 1998-2001. Given the dependence of Java performance on compiler technology, a recent study is overdue.

A brief introduction to the Java programming language was given in Section 1.1. A number of general (i.e., not specific to numerical computing) characteristics were presented; in this section we will identify and discuss those relevant to our application. Catching common programmer mistakes such as indexing an array out of bounds increases programmer productivity by decreasing development (specifically debug) time. Arrays are the primary data structure used in most numerical applications. In fact, all data types used in SPARSPAK are either instrinsics or arrays of instrinsics, making array index bounds checking a boon to development, but a hinderance to a stable, bug-free system. Use of array bounds checking in Java has encouraged researchers to find ways to legally eliminate the checks – as the technology matures, we anticipate the performance impact to be minimal.

Forcing all objects to be passed by reference into methods thereby preventing inadvertent copies to be made is a boon to numerical programmers. As an example, a large sparse matrix can easily occupy hundreds of megabytes of memory; inadvertently copying such a structure would result in both needless performance degradation and increased memory consumption.

As mentioned in Section 1.1, Java is cross-platform compatible at both the source and bytecode levels. That is, a Java program will exhibit similar behavior over all inputs on all hardware for which a conforming JVM is available. Numerical programs typically outlive the hardware they were originally designed to run on, making cross-platform compatibility especially

useful. Saad et al. in [6] make the remark regarding large scientific programs:

> Because the cost of writing large application programs is considerable, most of these codes are used over long periods of time. As a result many application codes running on today's latest architectures are not necessarily developed for these architectures. In addition, due to the diversity of existing architectures, application codes cannot be optimized uniformly across these architectures. These difficulties will probably not be resolved in the near future...

When they say 'application codes cannot be optimized uniformly across these architectures', they mean that code-level optimizations applied to a program for one architecture may not be useful (in fact, might even be a hinderance) for a different architecture. The virtual machine concept used by Java essentially introduces another layer of abstraction; the Java program specifies what to do, but decisions on how to do it are largely left up to the underlying virtual machine. Virtual machines can (and should) be tailored to the target architecture. We realize that cross-platform compatibility is not magically present; time and effort has to be spent developing a conforming JVM for the target platform. Furthermore, as evidenced in Section 2.1, not all virtual machines are created equal. Therefore, usefulness of cross-platform compatibility in our case depends on the availability of not only a conforming JVM, but a JVM whose performance under our application is acceptable.

Garbage collection neither hinders, nor is especially useful, for our application. On one hand it simplifies implementation by relegating the task of freeing dynamically allocated memory to the JVM, effectively preventing memory leaks. However it is usually the case that only a few large, persistent data structures are used in numerical applications. SPARSPAK is testament to this as fewer than one hundredy dynamically allocated objects (arrays) are created during a solution process, and it is usually very clear cut when they are no longer needed. As such, it would not be difficult to manually

manage memory in this case. Furthermore, relegating memory management to the JVM degrades application performance. But again, so few objects are created, we doubt garbage collection has substantial performance impact.

Most numerical applications tend to spend the majority of their time in a small minority of the code (these sections of code are commonly referred to as kernels or hot spots). Such behavior is ideally suited to Just-In-Time (JIT) compilation where code is selectively optimized based on execution frequency.

Perhaps Java's most frequent criticism is its poor performance relative to Fortran and C. As with virtually all other researchers working on high-performance computing in Java, one of the goals is to convince computational scientists that Java's reputation of being unacceptably slow for high performance computing is no longer suitable.

While it is true that benchmarks designed to test Java's numerical computation abilities typically contain some sort of simplistic sparse matrix kernel (typically sparse matrix-vector multiplication), the results can hardly be extended to our application. As such, another motivation for conducting our evaluation is lack of anything similar.

The leading edge Java compilers are free to users, whereas advanced Fortran 90 compilers can be very expensive. Showing that a complex numerical Java application can run at an acceptable speed relative to Fortran will alleviate users from having to purchase expensive sofware to achieve high performance in their numerical applications. Academic researchers will especially benefit.

In Section 2.1, we presented and discussed language issues identified by researchers as being detrimental to acceptance of Java as a platform for numerical computing. We will demonstrate that many are not relevant to our application. Early recognition of this fact provided substantial thesis motivation.

Lack of instrinsic complex number support has been cited as detrimental to adoption of Java as a platform for numerical computation. Our primary

tool for evaluating Java, SPARSPAK, does not contain complex number support. SPARSPAK is accepted as a standard for solving large sparse positive definite systems of linear equations using direct methods and has enjoyed success in both industry and academia. Its wide acceptance suggests that a substantial number of real-world problems do not involve complex numbers and allows us to conclude that limiting our study to real arithmetic does not inhibit its usefulness.

Java does not contain true rectangular multi-dimensional array support; introducing a slew of problems when multi-dimensional arrays are required. Every array in SPARSPAK is of rank one (i.e., there are no multi-dimensional arrays). Thus, another feature identified as necessary for numerical computing is not used by SPARSPAK, implying it is not required for solving large sparse positive definite systems of linear equations using direct methods.

Operator overloading is another feature cited as being useful for numerical applications. Operator overloading increases readability of numerical programs which in turn simplifies maintenance. However, operator overloading is only useful when abundant use of user-defined data types is made. SPARSPAK does not do much manipulation of user-defined data types and as such does not use operator overloading (even though it is a feature of Fortran 90). Thus, while useful in the general case, lack of operator overloading is not detrimental to creation of programs for solving large sparse positive definite systems of linear equations using direct methods.

Lastly, we would like to relay a success "story" detailing a massive conversion effort undertaken at Boeing Inc. to migrate its flagship legacy 3D Modelling software (AGPS) written in Fortran/C into Java[23]. Throughout the 25 year lifetime of AGPS, it was ported many times (a case of software outliving the hardware it was designed for). In most of these instances (especially in the early years, when critical code sections were written in assembly), the migration required a huge engineering effort. Requests were made for a PC port of AGPS. These requests were not taken seriously as AGPS required mature Fortran/C compilers, an X-Windows environment,

and contained numerous POSIX-standard system calls. As the number of systems supported by AGPS increased, the development team found itself spending most of its time working on porting issues, as opposed to attending to the ever-growing list of requested features. The team started development of a "proof-of-concept" Java program. They were attracted by Java's simple object model, networking capabilities, and promise of effortless cross-platform compatibility. As Java was new at this point, they were concerned with its performance, availability, and evolution. Development productivity increased by a factor of 2, in part due to the compiler catching many errors at compile time as opposed to runtime. They found that the Java prototype (containing some very intensive numerical computations) ran within a factor of 2 of the Fortran version, which they had lablled as their minimum acceptable performance differential. Java development was being performed on an SGI workstation – to test PC portability the class files (compiled on an SGI) were copied over to a PC, executed, and to the amazement of the developers, the application ran flawlessly! They had assumed some sort of tweaking would be necessary to achieve cross-platform compatbility. The PC compatibility that had been a running joke in the company for many years, was achieved for free with Java migration. New releases were deployed every 2-3 months as opposed to every 12-24 months as before. Users felt more comfortable requesting new features as they would be realized in less than 4 months as opposed to 12-24 as before. The performance difference between Java and Fortran continued to decrease as the Sun JVM matured. Equivalent functionality of over 300,000 lines of mixed Fortran and C code was achieved in 150,000 lines of pure Java code. From the paper[23]:

> The resulting gains in development, maintainability, robustness, plus the achieved portability made the migration to Java an extremely good investment for this legacy program and our engineering customers.

We have presented an industrial application involving high-performance numerical computation which has substantially benefitted from migration

into Java, further emphasizing the need to eliminate perceived inefficiencies of the langauge.

We have highlighted the primary motivations for the thesis. Java presents a friendlier programming environment than traditional languages, which makes it attractive for any project, ours included. Array bounds checking is are particularly conducive to numeric program development. Cross-platform compatibility is useful for numerical programs as the software typically outlives the hardware. The virtual machine concept coupled with just-in-time compilation is ideal for numeric computation as the majority of time is spent in kernels or hot-spots. Further motivation originates from a desire to promote Java as a viable alternative to Fortran and C for numeric computation. Finally, many characteristics identified by the community as being detrimental to numeric computing in general (operator overloading, lack of true multidimensional arrays) do not hinder our application.

3.3 Methodology

In this section we will describe how we intend to determine the suitability of the Java programming environment for solving large sparse symmetric positive definite systems using direct methods. This involves explicitly defining both the *Java programming environment* and what we mean by suitability for solving large symmetric sparse positive definite systems of linear equations using direct methods.

By the Java programming environment, we mean the combination of the language, a compiler, and a virtual machine. Obviously, many compilers and many virtual machines exist for the Java programming language. These components are interchangeable as long as conformance with the Java language specification is maintained. We are allowed to pick the compiler and virtual machine which best suit the needs of our application.

We define suitability of the Java programming environment for a particular application to be the environment's ability to satisfy the requirements

of the application (which clearly are application dependent). As previously mentioned, Fortran is the language traditionally associated with large scale scientific computations. As such, comparable performance of Java relative to Fortran is the primary issue. The second interest is ease of writing programs which achieve comparable performance. Clearly, the more difficult it is to coax high performance out of a program designed for our application, the less suitable Java is for it. We will now provide a more in-depth discussion of the first requirement.

To evaluate the first requirement, we converted a subset of SPARSPAK, a collection of Fortran subroutines for solving sparse systems of linear equations and least squares problems into Java (which we refer to as J-SPARSPAK). Specifically, we converted the portion of SPARSPAK designed to solve positive symmetric definite systems. We ask the following question: if J-SPARSPAK runs at a comparable speed to SPARSPAK, can we conclude that Java's performance is comparable to Fortran's for solving large sparse positive definite systems using direct methods (thus satisfying the first requirement)? SPARSPAK is the culmination of 25 years of intensive sparse matrix research and is accepted as a de facto standard for solving large symmetric sparse positive definite systems of linear equations using direct methods. Thus, any solver achieving performance comparable to SPARSPAK (J-SPARSPAK included), can be deemed suitable for solving large sparse positive definite systems using direct methods.

However, if J-SPARSPAK does not perform comparably to SPARSPAK, we need not conclude that Java is unsuitable for solving large symmetric sparse positive definite systems of linear equations using direct methods. As we will see, performance of Java applications is incredibly sensitive to data structure choice and programming style. During the conversion of SPARSPAK to Java, we attempted to minimize implementation differences by using the same data structures, algorithms, and programming style – we preserved the nature of SPARSPAK entirely. However, we have to accept the possibility that techniques delivering high performance in Fortran 90 might

not in Java. Although unlikely, we concede that entirely different data structures and programming style could radically improve Java performance and as such, can not conclude that poor performance of J-SPARSPAK relative to SPARSPAK implies that Java is not suitable for solving large symmetric sparse positive definite systems of linear equations using direct methods. We could merely conclude that direct adaptation of design choices and programming style in SPARSPAK is not conducive to the Java programming environment when performance is a concern.

The second requirement, ease of writing programs which achieve performance comparable to Fortran, is much more qualitative in nature. If the straight conversion yields respectable performance, we argue that good performance is not difficult to achieve in Java programs thus satisfying the second requirement. However, if unintuitive modifications to J-SPARSPAK are required for high performance, suitability is negatively affected.

We must provide a discussion regarding what relative performance difference is acceptable to scientific programmers. In other words, what constitues *comparable* performance. The issue is not clear-cut and is largely dependent on the particular situation. We again turn to the Boeing project [23] which migrated a legacy mixed Fortran/C application into Java. Analysis of this the conversion effort indicated that performance within a factor of 2 of Fortran is acceptable.

In conclusion, we have defined what we mean when we by suitability of Java for solving large sparse positive definite systems of equations using direct methods, along with stating how we are going to determine suitability.

3.4 Benchmarking

As previously mentioned, there are four phases to solving large symmetric sparse systems of linear equations using direct methods. Each of these phases has their own characteristics and requirements. We see no reason to believe that decent relative performance in one phase would imply decent relative

performance in another. Thus, even though our goal is determining the performance of all phases relative to Fortran in conjunction, we recognize each phase to be a separate computational unit requiring its own analysis.

Again, it is likely that modifications to J-SPARSPAK will improve performance. The initial benchmarks will determine which phase(s) we should focus our efforts on; subsequent effort should be directed towards profiling those phases to determine the bottlenecks, and attempting to eliminate the bottlenecks by any means possible. In the worst case scenario, no one bottleneck is identified, but rather J-SPARSPAK as a whole is unnacceptably slower than SPARSPAK. Ideally, a computational kernel is identified, the performance of which is pivotal to whole application performance, allowing us to focus our efforts.

Note that many benchmarks available today for Java programs 'warmup' the methods. By 'warmup' we mean the method to be benchmarked is exeucted a couple times prior to any actual timing which, in theory, causes the JIT compiler to translate the method into native code and perform any optimizations. We chose not to use this approach for a couple of reasons. First, the results are not indicative of a real-world environment (Java programs in an industrial setting do not 'warmup' prior to execution). Second, the server JVM uses on-stack replacement which compiles on the fly and switches to the compiled code for any long running method which was being interpreted.

Chapter 4

Implementation

4.1 Conversion of Sparspak to Java

Sparspak 90 is a collection of FORTRAN 90 subroutines for solving large sparse systems of linear equations and least squares problems [5]. One of the two schools of thought regarding benchmarking a system for performance deals with measuring the time required to complete a whole applcation code. We converted Sparspak 90 into Java in order to compare whole application code performance between Fortran 90 and Java.

The entire conversion was performed manually. Due to the modular, object-oriented structure of SPARSPAK, conversion to Java was relatively straightforward. However, Fortran 90 does contain language features not present in Java. These language features are simply 'syntactic sugar' – Fortran 90 is no more expressive than Java. We will highlight some of the challenges faced.

Passing Functions as Arguments

Fortran 90 allows the programmer to supply functions as parameters to functions. SPARSPAK utilizes this feature by allowing users to supply their own ordering function to the solver without recompilation of the entire package.

To simulate the user-defined ordering feature, we created an interface in Java, *userOrdering*, with one abstract method, *findOrder*. The user can then create a class which implements this interface, and then pass an instance of this class to the ordering routine.

Optional Function Arguments

Functions in Fortran 90 can contain optional parameters; this feature is utilized throughout SPARSPAK. We employ Java's method-name overloading to simulate optional function parameters. For example, if a Fortran 90 function had the following signature: *foo(Integer,Double)*, where the Double was optional, we would create two methods in Java, *foo(Integer)* and *foo(Integer,Double)*. Our approach fails when two optional parameters of the same data type exist consecutively in the signature of a function (this is referred to an ambiguous parameter pair). The aforementioned problem is rectified by creating uniquely-named methods. In the general case, if n is the number of ambiguous parameter pairs within the signature of a function, then 2^n uniquely-named methods are required. Practically speaking, this problem only presented a slight inconvenience as at most one ambiguous parameter pair was encountered in any function signature.

BLAS

SPARSPAK makes several calls to level 3 BLAS routines. Writing all necessary BLAS routines in Java would have been a major undertaking. Fortunately, J. Dongarra et. al wrote a Fortran 77-to-Java converter which they used to convert the BLAS routines into Java [10]. The first Java version of SPARSPAK utilized this package.

Pass by Reference

Arguments to Fortran functions can be passed either by value or reference. Primitivies passed into Java methods are passed by value, and objects are

passed by reference. It was sometimes the case that primitives passed into Fortran 90 functions were passed by reference and modified. To simulate this behavior in Java, we encapsulated the primitive in an object, allowing the proper modifications to occur. Slight use of this was required making performance degradation a non-issue.

Input/Output Formatting

Benchmarking Fortran vs. Java required test problems. We used a selection of matrices taken from the University of Florida Sparse Matrix Collection[1]. Matrices from this collection are stored using the Harwell-Boeing matrix exchange format[2]. An indication of Fortran's dominance in scientific computing, parsing matrices stored in the HB format requires reading data against Fortran formats. To handle this in the general case is a very complicated task. Jocelyn Paine[3] wrote a Java package for reading and writing data against Fortran formats. Using this package, we created a Harwell-Boeing file reader for Java[4] which we use in the Java version of SPARSPAK.

Arrays

Arrays in Fortran 90 are polymorphic in that the number of dimensions is dependent upon the array declaration at the top of the function. For example, an array with four elements could be viewed as a single-dimensional array (i.e., a vector) in one context and a two-dimensional array (i.e., a matrix) (of size (4,1), (2,2), or (1,4)) in another. It was often the case that a vector in one context was treated as a matrix in another. For example, a BLAS routine performing matrix-matrix multiplication, $DGEMM$, treats arrays passed into it as two-dimensional, whereas they are one-dimensional in the calling program of SPARSPAK. Fortran arrays are stored in column major order -

[1] http://www.cise.ufl.edu/research/sparse/matrices/
[2] http://www.netlib.org/utk/papers/matrixmarket/node8.html
[3] http://www.j-paine.org/
[4] http://www.math.uwaterloo.ca/~saarmstr/software.html

consecutive column elements are stored contiguously in memory. As such, each call of the form $A[i][j]$ in $DGEMM$ was translated into $A[i+(j-1)*n]$ with n being the number of columns in A. The obvious performance impact will be explored later.

Fortran 90 supports the ability to pass sections of arrays into a function. For instance, if the parameter $A(10)$ was passed into a function, accessing $A(5)$ from within the function is equivalent to accessing $A(15)$ in the calling program. For every use of this feature within SPARSPAK, we added an extra integer parameter to the method in the Java version serving as an offset which was added to every index to the array within the function. Lack of pointers in Java clearly hinders performance in this case as an extra addition is required for every index into an array of this sort. The performance hit can be lessened by clever programming techniques as will be explored later.

By default, arrays in Fortran 90 are indexed from 1 to n, where n is the length of the array. Arrays in Java are always indexed from 0 to $n-1$. We chose to make the length of each Java array one greater than necessary (allowing us to use the same index into each Java array as was made into the corresponding Fortran array). If arrays of the same length were used, 1 would have to be subtracted from each array index in Java making translation mistakes more likely and the resulting code less efficient. The downside to the approach, of course, is a slightly higher memory requirement. In SPARSPAK, this is negligible since fewer than 100 arrays are used throughout any solution process.

Labelled Loops

Fortran allows loops to be labelled and provides the ability to transfer control flow from anywhere inside the labelled loop to outside the labelled loop. This feature is typically used to branch from a loop nested inside the labelled loop to outside the labelled loop. An example best demonstrates how we simulated this feature in Java. Figure 4.1 contains a Fortran 90 program and the corresponding semantically equivalent Java program. The program

consists of an outer loop, labelled *outer*, inside of which lies another loop, *inner*, and inside another loop, unlabelled. Inside the innermost loop we have both a branch to outside *outer* and a branch to outside *inner*.

For each loop designated *label* in Fortran 90, we create a boolean variable *breakLabel* in Java (in this case, *breakOuter* and *breakInner*). If a statement of the form *exit label* is encountered in Fortran 90, we set *breakLabel* to true, and immediately break out of the currently executing loop. Suppose that we made it to line 5 of the Fortran 90 program, control flow immediately jumps to line 12. Similarly, in Java, *breakOuter* is set to true, and we break out of the inner-most loop. Control then jumps to line 8, followed by a jump to line 11, and finally out of the *outer* loop, as required [5].

4.2 Test Problems

To compare the performance of Java vs. Fortran running SPARSPAK we solve equations of the form $\boldsymbol{Ax}=\boldsymbol{b}$ where $\boldsymbol{A}$ is an n by n, large sparse symmetric positive definite matrix either generated by SPARSPAK's built-in grid-problem creator, or one of a selection of matrices from the University of Florida Sparse Matrix Collection. The vector $\boldsymbol{b}$ is generated so that the solution $\boldsymbol{x}$ is $1, \ldots, n$ (allowing us to both easily verify the solution, and that numerical stability was maintained).

In Table 4.1, we present properties of the matrices used to test SPARSPAK. The integer n represents the number of rows and columns in the matrix (all matrices are rectangular). NNZ per row of $\boldsymbol{A}$ indicates the average number of non-zeroes in a row of $\boldsymbol{A}$ and is defined as

$$\frac{\boldsymbol{\eta(A)}}{n} \tag{4.1}$$

[5] Clearly, use of 'goto' statements would be a more natural mechanism of accomplishing the same behavior. Java supports a 'goto' statement at the bytecode, but not the source level.

Problem	n	NNZ per row of $\boldsymbol{A}$	NNZ per row of $\boldsymbol{LL^t}$
HB Problems			
cfd2	123 440	24	1054
ct20stif	52 329	51	422
finan512	74 752	7	172
nd3k	9 000	363	3398
nd6k	18 000	382	5736
nemeth26	9 506	158	167
pwtk	217 918	52	516
Grid Problems			
600 by 600	360 000	5	67
700 by 700	490 000	5	69
800 by 800	640 000	5	73
900 by 900	810 000	5	76
1000 by 1000	1 000 000	5	78
1100 by 1100	1 210 000	5	79
1200 by 1200	1 440 000	5	79

Table 4.1: Test Problem Properties

Similarly, NNZ per row of $\boldsymbol{LL^t}$ indicates the average number of non-zeroes per row of $\boldsymbol{LL^t}$.

Grid problems are characterized by a very regular and predictable matrix structure – they are symmetric with a cluster of elements along the diagonal and bands of elements parallel to the diagonal. The predictable matrix structure offers a controlled test bed under which to examine the effects of incrementing problem size with everything else remaining proportional. We found that a *1200 by 1200* grid, (fifty-eight million non-zeroes in the Cholesky factor) was the largest problem we could solve (limited by the amount of main memory in our test machine). Anything smaller than *600 by 600* is too small to be of any interest. Thus, we examined square 5-point grid problems of size

600 to *1200*, in increments of *100*. Each node of a 5-point grid is connected to four other grid points and itself (except around the border of the grid). Since the points around the border constitute the minority, the average number of non-zeroes per row of a 5-point grid problem is five, as evidenced by Table 4.1.

SPARSPAK uses the Multiple-Minimum-Degree (MMD) ordering algorithm. When the MMD algorithm is applied to a $n = k \times k$ grid problem, the number of non-zeroes in the Cholesky factor is predicted roughly by equation:

$$\boldsymbol{\eta(L)} = ck^2 \log_2 k \tag{4.2}$$

where c is a constant. Dividing Equation 4.2 by n gives us the average number of non-zeroes per row (or column) of a factored grid problem: $c \log_2 k$. We see that the number of non-zeroes per row of a grid problem increases proportional to the log of the grid size, k. The growth of the NNZ per row of $\boldsymbol{LL^t}$ of Table 4.1 over the grid problems is consistent with this finding.

As of May 2004, there were 943 matrices in the University of Florida Sparse Matrix Collection. We examined the properties of each, and selected all of those which fit our criteria; we required our matrices to be sufficiently large (i.e., $n > 10000$) so as to decently stress the solver causing solution times to be at the bare minimum many seconds, diminishing timing and operating system effects which plague very short computations. However, we want the maximum amount of memory required to be less than that available on our system (1 GB) to avoid use of virtual memory, the performance of which is dependent on current file system fragmentation and beyond our control. In addition, we are concerned with solving problems which are symmetric positive definite.

The sizes of the problems extracted from the University of Florida Sparse Matrix Collection vary from *9000 by 9000* to *217918 by 217918*. The MMD ordering algorithm performs the best on the matrix *nemeth26*, and the worst on *cfd2*. The *ndXX* problems are by far the most dense, both in terms

of $\boldsymbol{A}$ and $\boldsymbol{LL^t}$. In the subsequent chapter, we will present the results of solving these matrices using both SPARSPAK and the initial version of J-SPARSPAK.

Fortran 90

```
1  outer: do while( test1 )
2    inner: do while( test2 )
3      do while( test3 )
4        if( test4 ) then
5          exit outer
6        else
7          exit inner
8        end if
9      end do
10   end do inner
11 end do outer
12
```

Java

```
1  boolean breakOuter = false;
2  while( test1 ) {
3    boolean breakInner = false;
3    while( test2 ) {
4      while( test3 ) {
5        if( test4 ) { breakOuter = true; break; }
6        else        { breakInner = true; break; }
7      }
8      if( breakOuter ) { break; }
9      if( breakInner ) { break; }
10   } // end while( test2 )
11   if( breakOuter ) { break; }
12 }
```

Figure 4.1: Fortran 90 Code showing loop labelling

Chapter 5

Results and Experimentation

5.1 Hardware

Tests were conducted on a machine with an AMD Athlon XP Processor clocked at 1.4GHz with 1GB of RAM. As mentioned in Section 2, virtual machines for PCs deliver much better performance than those available for other architectures. As such, we chose to run J-SPARSPAK on Windows XP Professional Edition with service pack 1 using Sun JVM v.1.5.0. J-SPARSPAK was executed with the following options: `Xmx1024m`, which sets the maximum heap size to 1024 megabytes (the default 64 megabyte heap size is not adequate to solve all but the smallest of problems); `Xms128m`, which sets the minimum heap size to 128 megabytes thus decreasing amount of time spent acquiring memory from the operating system. As mentioned, the server VM typically out-performs the client VM for longer running computations. However, to provide evidence of this we will present results from both the client and server VMs, after which we will focus solely on the server VM.

We used Compaq Visual Fortran v.6.6b[1] to compile SPARSPAK into native win32 code. SPARSPAK was compiled with maximum optimizations, and the generated code was optimized for the K7 (AMD Athlon) processor.

[1] http://h18009.www1.hp.com/fortran/

We noticed that early tests exhibited unacceptable variability in solution times (i.e., we were unable to achieve repeatability). This was fixed by executing SPARSPAK and J-SPARSPAK at highest priority (decreasing the probability of being pre-empted by another process). All times presented in all tables are measured in seconds unless otherwise specified, and are the average[2] at least three runs at high priority.

5.2 Initial Results

In this section, we will present initial results and analysis comparing the initial version of J-SPARSPAK to SPARSPAK.

Tables 5.1 (server VM) and 5.2 (client VM) present results comparing the first version of J-SPARSPAK to SPARSPAK. As mentioned, we use both Harwell-Boeing (HB) problems, and problems generated by SPARSPAK's internal grid problem generator. The two tables have the same format. The first column indicates the name of the problem for the Harwell-Boeing problems, and the size of the grid for the grid problems. The row labelled **HB ratio** is a measure of how much slower Java is relative to Fortran for that particular phase of computation over all HB problems. Similarly, **Grid ratio** indicates the same statistic over all grid problems. The row labelled **Average ratio** is the average ratio over both HB and grid problems.

Note that the Fortran numbers presented are the same for Tables 5.1 and 5.2 – we are comparing different Java VM implementations against Fortran compiled with maximum optimizations. We will first discuss performance of the server VM versus the client VM. The client VM appears to be a better overall performer than the server VM, yielding better times relative to Fortran (signified by lower ratios) in the ordering, symbolic factorization, and triangular solve phases. However, the numeric factorization phase performs much worse under the client VM, on average 5.25 times slower than Fortran

[2]We chose the average as opposed to minimum of the times to give a real-world indication of what can be expected

	Ordering		Symbolic Factorization		Numeric Factorization		Triangular Solve	
Problem	Java	Fortran	Java	Fortran	Java	Fortran	Java	Fortran
cfd2	3.84	1.69	4.16	2.52	956.96	214.61	3.22	1.85
ct20stif	2.45	1.50	1.51	0.57	67.89	17.51	0.65	0.32
finan512	12.85	12.10	1.26	0.39	58.38	16.76	0.45	0.21
nd3k	4.69	2.56	1.25	0.66	348.83	74.92	0.79	0.43
nd6k	9.62	6.28	2.98	2.23	2059.3	451.50	2.47	1.55
nemeth26	2.90	1.26	0.74	0.11	4.30	0.90	0.10	0.06
pwtk	5.13	2.56	4.36	2.82	328.58	89.72	2.80	1.62
HB ratio	**1.80**		**2.72**		**4.21**		**1.81**	
600	3.66	1.25	2.90	1.41	24.86	8.69	0.83	0.49
700	4.32	1.71	3.17	1.69	37.50	13.48	1.11	0.69
800	4.80	2.25	4.08	2.26	60.18	20.98	1.51	0.92
900	5.82	2.88	4.32	2.92	97.76	31.83	1.87	1.22
1000	6.55	3.62	5.39	3.66	125.68	40.35	2.34	1.52
1100	7.19	4.60	5.86	4.43	167.19	52.07	2.87	1.85
1200	8.68	5.57	6.68	5.28	202.34	65.19	3.39	2.23
Grid ratio	**2.08**		**1.61**		**3.00**		**1.58**	
Average Ratio	**1.94**		**2.17**		**3.61**		**1.70**	

Table 5.1: Initial Fortran and Java Comparison (Server VM)

whereas under the server VM the numeric factorization runs only 3.61 times slower. The performance difference of the two VMs can be explained in terms of phase runtimes. Notice that the ordering, symbolic factorization, and triangular solve phases, as a general rule, all complete well within 10 seconds. Recall that the server VM spends much more effort optimizing the JIT compiler output – the benefit of which is only realized for significantly

	Ordering		Symbolic Factorization		Numeric Factorization		Triangular Solve	
Problem	Java	Fortran	Java	Fortran	Java	Fortran	Java	Fortran
cfd2	2.27	1.69	3.20	2.52	1375.58	214.61	2.88	1.85
ct20stif	1.55	1.50	0.74	0.57	100.85	17.51	0.51	0.32
finan512	11.26	12.10	0.60	0.39	81.52	16.76	0.33	0.21
nd3k	3.67	2.56	0.84	0.66	457.97	74.92	0.68	0.43
nd6k	8.54	6.28	2.48	2.23	2825.13	451.50	2.28	1.55
nemeth26	2.16	1.26	0.25	0.11	1.61	0.90	0.06	0.06
pwtk	3.72	2.56	3.33	2.82	538.81	89.72	2.57	1.62
HB ratio	**1.32**		**1.41**		**5.31**		**1.47**	
600	1.83	1.25	1.58	1.41	42.00	8.69	0.67	0.49
700	2.27	1.71	2.26	1.69	66.52	13.48	0.93	0.69
800	3.10	2.25	2.96	2.26	106.63	20.98	1.26	0.92
900	3.75	2.88	3.77	2.92	168.70	31.83	1.68	1.22
1000	4.71	3.62	4.79	3.66	215.78	40.35	2.07	1.52
1100	6.35	4.60	5.60	4.43	280.86	52.07	2.55	1.85
1200	8.02	5.57	6.07	5.28	354.44	65.19	3.05	2.23
Grid ratio	**1.37**		**1.25**		**5.19**		**1.37**	
Average Ratio	**1.35**		**1.33**		**5.25**		**1.42**	

Table 5.2: Initial Fortran and Java Comparison (Client VM)

longer running computations (such as the numeric factorization). Smaller computations such as the three aforementioned phases over our problem set are in fact ill-suited towards the server VM. Even though the client VM outperforms the server VM in three of the four phases, the server VM is still a better choice for solving large symmetric sparse positive definite systems of linear equations using direct methods as the benefit realized by the server VM over the numeric factorization more than compenstates for the slower

	Ordering		**Symbolic Factorization**		**Numeric Factorization**		**Triangular Solve**	
Problem	Java	Fortran	Java	Fortran	Java	Fortran	Java	Fortran
cfd2	0.4%	0.8%	0.4%	1.1%	98.8%	97.3%	0.3%	0.2%
ct20stif	3.4%	7.5%	2.1%	2.9%	93.6%	88.0%	0.9%	0.4%
finan512	17.6%	41.1%	1.7%	1.3%	80.0%	56.9%	0.6%	0.3%
nd3k	1.3%	3.3%	0.4%	0.8%	98.1%	95.4%	0.2%	0.1%
nd6k	0.5%	1.4%	0.1%	0.5%	99.3%	97.8%	0.1%	0.1%
nemeth26	36.1%	54.1%	9.2%	4.7%	53.5%	38.6%	1.2%	0.7%
pwtk	1.5%	2.7%	1.3%	2.9%	96.4%	92.8%	0.8%	0.5%
600	11.4%	10.6%	9.0%	11.9%	77.1%	73.4%	2.6%	4.1%
700	9.4%	9.7%	6.9%	9.6%	81.3%	76.7%	2.4%	3.9%
800	6.8%	8.5%	5.8%	8.6%	85.3%	79.4%	2.1%	3.5%
900	5.3%	7.4%	3.9%	7.5%	89.1%	81.9%	1.7%	3.1%
1000	4.7%	7.4%	3.8%	7.4%	89.8%	82.1%	1.7%	3.1%
1100	3.9%	7.3%	3.2%	7.0%	91.3%	82.7%	1.6%	2.9%
1200	3.9%	7.1%	3.0%	6.7%	91.5%	83.3%	1.5%	2.9%
Average	**7.6%**	**12.1%**	**3.6%**	**5.2%**	**87.5%**	**80.4%**	**1.3%**	**1.9%**

Table 5.3: Initial Results Breakdown by Percentage

times over the three less significant phases. Unless otherwise specified, all further attention will be directed towards the server VM.

Table 5.3 indicates the percentage of the total solution time consumed by each phase for both Fortran and Java. We present the statistic for each problem along with the average percentage consumed over all problems. We will now discuss the initial results and their implications. We will examine each of the phases seperately, and then the solution process as a whole.

Ordering

In the ordering phase for Harwell Boeing problems, Java runs, on average, 1.8 times slower than Fortran. Notice that the longer the ordering takes, the better Java performs relative to Fortran. For example, *nk6k* takes only 1.52 times as long, and *finan512* takes only 1.06 times as long. Recall that under JIT compilation, run time is the sum of the compilation time plus the execution time. Thus, JIT compilation really shines when subjecting large and/or stressful datasets to small sections of code, as is the case when ordering the aforementioned matrices using the MMD degree algorithm.

One might wonder why the ordering for *finan512* takes longer than any other matrix, even though it is neither the largest nor the most dense. From the documentation associated with the matrix[3]:

> This matrix is quite strange, and appears to be highly sensitive to tie-breaking issues, or some other phenomenon...

The majority of matrices, when subjected to some variation of the minimum-degree ordering algorithm, produce comparable results but often with large discrepancies in ordering times. The matrix *finan512* produces neither comparable results, nor comparable ordering times. For more information on this phenomenon, see [22].

On average, the ordering process for the grid problems takes longer in Java relative to Fortran than the Harwell-Boeing problems (2.08 as opposed to 1.80). This is due to none of the grid orderings taking a particularly long time – even *1200 by 1200* only requires 8.68 seconds in Java. We suspect that ordering larger grid problems would reduce the gap between Java and Fortran. Overall, Java runs at approximately half the speed of Fortran for the ordering process. We deem this acceptable, especially given that the ordering process constitutes a relatively small minority of the entire solution time.

[3] http://www.cise.ufl.edu/research/sparse/HBformat/Mulvey/

From Table 5.3, we see that the ordering process consumes, on average, 7.6% of the time required for the solution process for Java, and 12.1% for Fortran. Note that this does not mean the ordering runs faster in Java (it does not in most cases), it simply means that there are other phases which, relative to the ordering phase, perform worse in Java. In both Fortran and Java, the ordering process is the second-most demanding phase, albeit a distant second.

Symbolic Factorization

The amount of work performed by the symbolic factorization process is roughly proportional to the number of non-zeroes contained in, and the structure of, the Cholesky factor. Similar to the ordering process, we see that the longer running computations (such as symbolically factoring the *1200 by 1200 grid*) perform the best under Java relative to Fortran. Again, this is due to the decreasing cost/benefit ratio characteristic of JIT compilation under longer running computations. Overall, Java runs approximately half as fast as Fortran during the symbolic factorization phase. But like the ordering phase, the amount of time spent is small relative to the total solution time.

Numeric Factorization

By far the most interesting and important phase of the solution process is the numeric factorization. Recall that this phase is responsible for numerically computing the Cholesky factor. The density of the Cholesky factor plays a larger role than simply the number of non-zeroes (as is the case in the symbolic factorization), as evidenced by cross-referencing Table 4.1 with Table 5.1. The matrices containing the most non-zeroes per row in their Cholesky factor are those which consume the most time during the numeric factorization.

The numeric factorization performs far worse in Java than it does in Fortran – 4.21 times as bad for Harwell-Boeing problems, and 3.00 times as bad for Grid problems, average of 3.61 times slower. It is unfortunate (and

further proof of Murphy's Law) that the most critical phase of the solution process is plagued with such poor efficiency while the less important phases exhibit respectable performance. As the problem size increases, we anticipate the numeric factorization to further dominate the overall computation to the point where performance of the other phases is nearly irrelevant. This trend is confirmed by examining the percentage of time spent in the numeric factorization of the grid problems. The percentage of time spent in the numeric factorization increases monotonically with respect to the grid problem size (as evidenced in Table 5.4. Similarly, the Harwell-Boeing problems with the most non-zeroes in the Cholesky factor (*cfd2*, *nd3k*, *nd6k*) spend most of their time in the numeric factorization. Clearly, the suitability of Java for solving large symmetric sparse positive definite systems of linear equations using direct methods depends on the ability to perform a fast and efficient numeric factorization. This will be re-visited later in the thesis.

Triangular Solve

The triangular solve phase is trivial relative to the other three phases of the solution process. It consumes, on average, only 1.2% of the solution time in Java, and 1.8% of the solution time in Fortran. Java achieves decent performance relative to Fortran in this phase, running at 1.7 times the speed of Fortran over both Harwell-Boeing and Grid problems. The triangular solve is accomplished in only 30 lines of Fortran 90 code. We suspect the simplicity of the routine allowed the JIT compiler to translate it to native code very quickly; thus, even though it is very short-lived, we do not pay a significant penalty for the JIT compilation. Like the ordering and symbolic factorization phases, the triangular solve phase is relatively insignificant to the solution process as a whole, and as such no further attention will be paid to the triangular solve phase.

Overall Analysis

Table 5.4 presents the total solution times for all of the problems. The column labelled **Ratio**, similiar to previous tables, is the Java solution time divided by the Fortran solution time. The next column, **Numeric Factorization Percentage**, indicates the percentage of time spent in the numeric factorization. Java performs, on average, 3.79 times worse for Harwell-Boeing problems, and 2.79 times worse for the grid problems. The relatively worse performance of Harwell-Boeing problems with respect to the grid problems is attributable to generally higher density of the Cholesky factors, causing more time to be spent on the numeric factorization. The performance of Java relative to Fortran over grid problems as the grid size increases remains fairly constant. The *600 by 600* grid runs approximately 2.72 times slower in Java, and the *1200 by 1200* grid runs 2.76 times slower. We attribute the consistency to the (expected) logarithmic growth of the number of non-zeroes per row in the Cholesky factor with respect to the grid size – 67 per row in the *600 by 600* grid, and only 79 per row in the *1200 by 1200* grid.

The problems range in solution times (Java) from 8.04s (*nemeth26*) to 2074s, over 30 minutes (*nd6k*). Over all problems, Java runs 3.29 times slower than Fortran. Again, we anticipate that as the problem size increases, this ratio will more closely resemble the average numeric factorization ratio (3.62). We repeated the numeric factorization percentages from Table 5.3 to show the correlation between solution time ratio and the amount of time spent in the numeric factorization. With the exception of *finan512* (confuses the MMD algorithm) and *nemeth26* (a relatively small problem), we can conclude that the more time spent in the numeric factorization, the worse the resulting solution time ratio. We point this out to further illustrate reliance of solver performance on a fast and efficient numeric factorization.

Conclusion

After benchmarking the first version of J-SPARSPAK against SPARSPAK, we have discovered that Java runs, on average, 3.21 times slower than For-

	Total Solution Time		**Ratio**	**Numeric Factorization Percentage (Java)**
Problem	Java	Fortran		
cfd2	968.17	220.67	4.39	98.84%
ct20stif	72.50	19.90	3.64	93.65%
finan512	72.94	29.46	2.48	80.04%
nd3k	355.55	78.57	4.53	98.11%
nd6k	2074.42	461.56	4.49	99.27%
nemeth26	8.04	2.33	3.45	53.47%
pwtk	340.88	96.72	3.52	96.39%
600	32.26	11.85	2.72	77.07%
700	46.11	17.58	2.62	81.34%
800	70.57	26.42	2.67	85.27%
900	109.77	38.86	2.83	89.06%
1000	139.96	49.14	2.85	89.80%
1100	183.12	62.94	2.91	91.30%
1200	221.09	78.27	2.82	91.52%

Table 5.4: Total Solution Times

tran. The majority of the performance difference is attributable to poor numeric factorization performance relative to the other phases. Improving J-SPARSPAK performance will depend on our ability to diagnose exactly why this phase is performing slower relative to the other phases, and hopefully determine ways to improve performance. Recall from Section 3.4, in the event Java performs radically slower than Fortran, the key to improving performance is identifying one or more kernels which dominate the computation. Successfully identifying such kernels allows a focused effort on improving performance over those code portions, something well within our abilities. In the

event that Java is simply slower across the board than Fortran and no specific code section can be identified as dominating the numeric factorization process, we suspect lack of compiler maturity and general run-time overhead associated with Java to be the culprit – the only solution to which requires access to, and intricate knowledge of, a Java compiler. In the following section, we will present the results of profiling the numeric factorization, with the goal of identifying hot-spots, which we can then attempt to optimize and thus improve performance of Java relative to Fortran.

5.3 Profiling Results

We are confident that profiling the solution process will reveal one or more code sections which dominate the numeric factorization process. The other 3 phases each run approximately twice as slow in Java as opposed to Fortran – one would think that the numeric factorization phase could be coaxed to exhibit similar behaviour. We anticipate Java performance over a large portion of the numeric factorization to be on par with other phases, and performance in a small minority of code to be significantly worse. In this section, we will discuss how we profiled both J-SPARSPAK and SPARSPAK and discuss the results of profiling the numeric factorization phase.

Annoyingly, when running in server mode, the JVM does not provide accurate profiling results. We speculate that the more agressive optimizations of the server virtual machine are at odds with providing accurate profiling information. As such, we used the client virtual machine when profiling J-SPARSPAK. We acknowledge that results obtained with the client virtual machine differ from those obtained with the server virtual machine (short phases such as the ordering, symbolic factorization, and triangular solve run faster, whereas the numeric factorization runs much slower). However, we suspect the results are useful since we are simply trying to identify where the numeric factorization phase is spending most of its time.

The Sun JVM provides profiling support through the Java Virtual Ma-

chine Profiling Interface (JVMPI). The JVMPI allows heap, cpu, or monitor profiling. We would like to utilize the cpu profiling ability to determine where the numeric factorization phase is spending the majority of its time. The VM option `-Xrunhprof:cpu=samples,depth=8` instructs the virtual machine to perform cpu sampling at the default interval of 20ms at a stack depth of 8 (the profiler will probe as far as 8 method calls deep – more than enough to accurately profile J-SPARSPAK). The result of the profiling is a file containing traces in the following format:

```
TRACE 300090:
 org.netlib.blas.Dgemm.dgemm(Dgemm.java:391)
 SparseSpdMethod.SpkLDLtFactor.LDLtFactor(SpkLDLtFactor.java:182)
 SparseSpdMethod.SparseSpdBase.Factor(SparseSpdBase.java:281)
 SparseSpdMethod.SparseSpdSolver.Factor(SparseSpdSolver.java:109)
 SparseSpdMethod.SparseSpdSolver.Solve(SparseSpdSolver.java:46)
 driver.main(driver.java:60)
```

indicating that the sample observed the program to be on line 391 of the method `DGEMM` in class `org.netlib.blas.Dgemm`, which was called from the `LDLtfactor` routine etc.. The number of traces in the file is dependent on the number of unique samples identified by the profiler. The profiler records the stack trace for each unique sample, and the number of times each sample was hit. An open-source utility *PerfAnal*[4] is available to parse and display the profiler output in a useful fashion. Running *PerfAnal* on the trace file of *nd3k* produces the output shown in Figure 5.1. We have expanded the call to the numeric factorization phase, and see that, of the 33.79% of the time spent in the numeric factorization phase, 32.14% of the time is spent in the BLAS routine `DGEMM`. Thus, a total of 95% of the numeric factorization is spent in that one routine!

Notice that the profiler produces results inconsistent with figures reported earlier (which indicated that 98.11% of the solution process was spent in the

[4]http://java.sun.com/developer/technicalArticles/Programming/perfanal/

numeric factorization of *nd3k*). This is clearly not due to differences between the client and server virtual machines, but instead the profiler reports numeric factorization time as a percentage of total program run time, which includes time required to read the problem. Recall that we are using the Fortran format parser created by Jocelyn Paine, which, although functional, is very slow – thus the total program runtime when solving Harwell-Boeing problems is actually dominated by the time required to read in the problem. Fortunately, the suitability of Java for solving large sparse positive definite systems of linear equations using direct methods does not depend on the ability to read and parse Harwell-Boeing files in a timely manner.

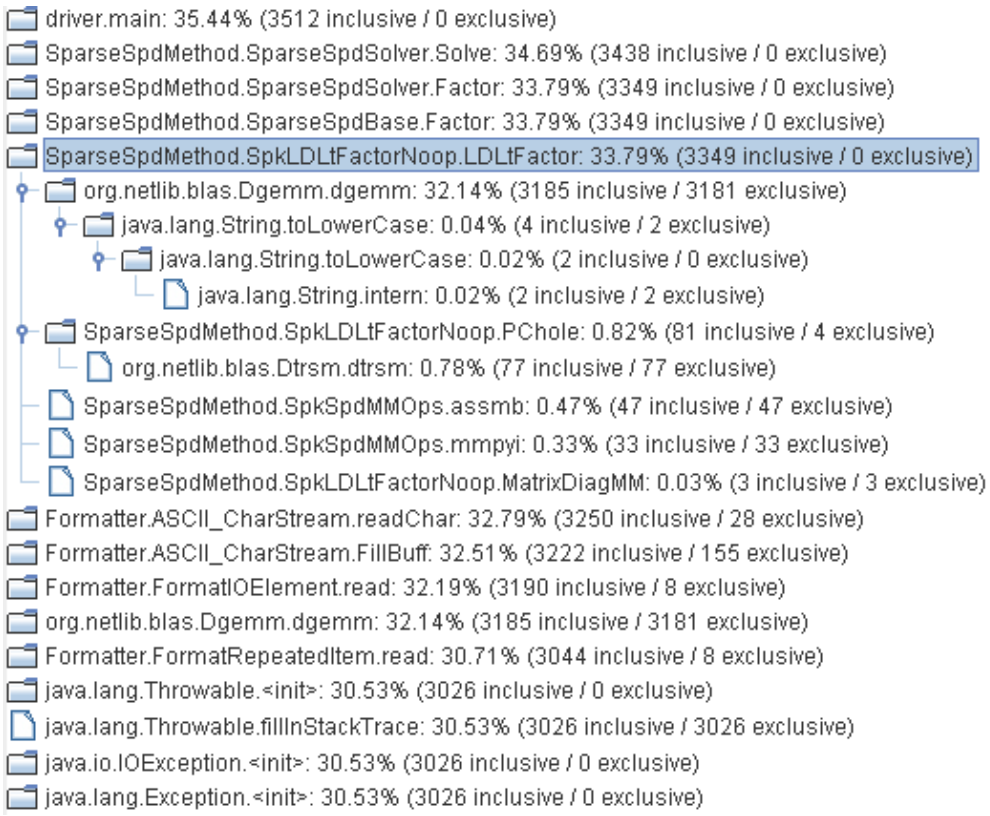

Figure 5.1: Output of PerfAnal

Compaq Visual Fortran (CVF) contains built-in profiling through the IDE. We employed the function timing ability of CVF to determine areas where SPARSPAK is spending the majority of its time in the numeric factorization routine. Similar to profiling Java, we set the stack probe depth to 8. The output of the profiler is less cryptic than that output from the Java profiler. In fact, the profiler sorts the output in terms of most active function. For example, a portion of the the profiler output for *nd3k* is as follows:

```
                                    | Func          Func+Child     Hit
 Functions and Callers              | Time     %    Time      %    Count

_DGEMM@60 (dxml_ts_gemm.obj)        |74102.5  57.9 74213.2  58.0 41492
  _SPKLDLTFACTOR_mp_LDLTFACTOR@36 *|72357.1  97.6 72404.5  97.6 13181
    _SPKSPARSESPDBASE_mp_SPARSESPD*|72357.1  97.6 72404.5  97.6 13181
  _DTRSM@60 (dxml_ts_trsm.obj)      |1745.4   2.4 1808.7   2.4 28311
    _SPKLDLTFACTOR_mp_LDLTFACTOR@3*|1745.4   2.4 1808.7   2.4 28311
```

The output indicates that 58.0% of the time was spent in the BLAS routine `DGEMM`. However, of that 58.0%, 97.6% was the result of calls from `SPKLDLTFACTOR`, and 2.4% the result of calls from another BLAS routine `DTRSM`. Interestingly enough, the `DTRSM` from the version of LAPACK used in J-SPARSPAK does not call `DGEMM` (as indicated by Figure 5.1: `org.netlib.blas.Dtrsm.dtrsm...` is a leaf node). Thus, all of the time spent in `DGEMM` of J-SPARSPAK is the result of calls directly from `SPKLDLTFACTOR`. When comparing to time spent in `DGEMM` of SPARSPAK we must therefore be careful to exclude all time spent in `DGEMM` calls not made directly from `SPKLDLTFACTOR` (which constitute the minority for *nd3k* – 2.4%). Figure 5.2 shows the numeric factorization call graphs for both Java and Fortran.

Another interesting aspect of the SPARSPAK profiling result is that `PCHOLE` is mysteriously absent from the list of functions identified by the profiler – this function comprises, on average, 8.7% of the numeric factorization

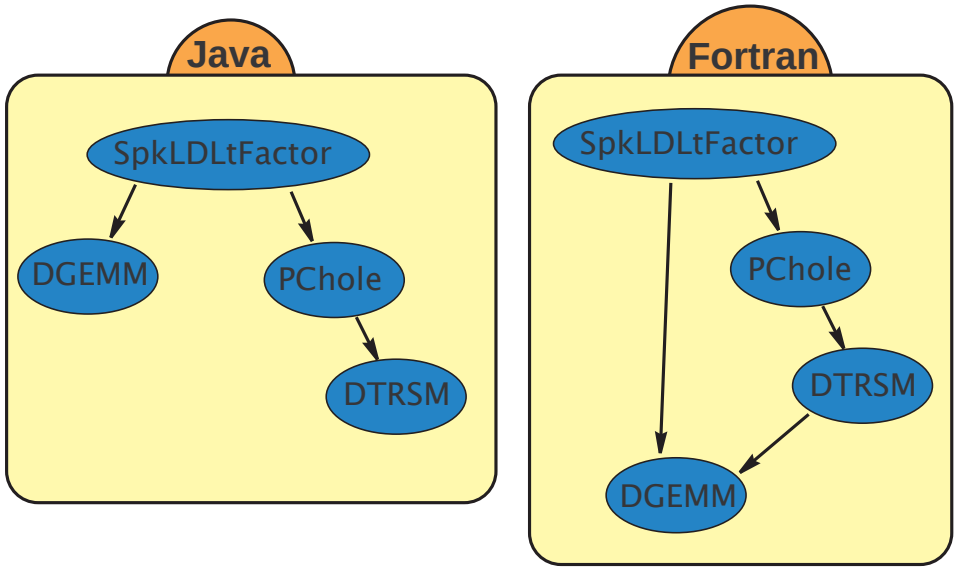

Figure 5.2: Function call graphs for J-SPARSPAK AND SPARSPAK

time of J-SPARSPAK. We eventually realized that the CVF compiler must be inlining this function directly into its calling routine, `SPKLDLTFACTOR`. The runtime system obviously has no knowledge of this, and thus gives no indication that a `PCHOLE` function even exists. The clue that the function was inlined came from a profiler trace indicating a call to `DTRSM` directly from `SPKLDLTFACTOR`, when in fact the only call to `DTRSM` is from `PCHOLE`. Upon inspection, `PCHOLE` is a reasonable choice for inlining as it consists of only 30 lines, and is called only once from `SPKLDLTFACTOR` (recall that inlining a function results in increased code size). Due to the inlining of `PCHOLE` we are unable to get an accurate comparison of the time J-SPARSPAK spends in `PCHOLE` relative to SPARSPAK. However, looking at the output of the Java profiler, we see that the vast majority of the time spent in `PCHOLE` is spent in `DTRSM`; thus we can get a reasonable approximation of the time spent in `PCHOLE` from the time spent in `DTRSM`.

The numeric factorization process is initiated by a call to the routine `LDLTFACTOR` in both J-SPARSPAK and SPARSPAK. When the call finishes,

the numeric factorization is complete (note that the routine makes multiple calls to subroutines). As such, we would like to determine where this routine is spending the majority of its time – whether in a subroutine or in the actual `LDLTFACTOR` routine itself.

In Table 5.5, we present a breakdown of percentage of time spent in the three most active functions during numeric factorization, and the percentage of time spent in the `LDLTFACTOR` routine and all other subroutines (The `PCHOLE` routine of SPARSPAK is actually the time spent in `DTRSM`, which we suspect is very close to the amount of time spent in `PCHOLE`). In Figure 5.3, we present a graph of the data contained in the rows of the table corresponding to the Harwell-Boeing problems. Similarly, Figure 5.4 displays the grid problem data. For both graphs, the first bar of a problem indicates the Java breakdown, and the second bar indicates the Fortran breakdown. With the graphs, one is better able to visualize the amount of time spent in each of the four identified areas of the numeric factorization routine (keep in mind the segregation is applied to help us better understand where the numeric factorization is spending most of its time – the breakdown is not meant to imply there are four logical phases of the numeric factorization the same way there are four phases to the overall solution process).

The first thing one notices in Table 5.5 is the amount of time spent in the routine `DGEMM` in both Java and Fortran. Java spends, on average, 87.1% of its numeric factorization time in the routine, whereas Fortran spends 67.7% of its time in `DGEMM`. For longer running computations, such as *nd6k*, 95.81% of the Java numeric factorization time is spent in the routine `DGEMM`, giving rise to the possibility that greater than 90% of the entire solution process is spent in this one routine! Let us now try to make some sense as to exactly what is implied by the results in Table 5.5.

Recall that Java with the client VM spends, on average, 5.25 more time in the numeric factorization routine than Fortran. Suppose that the Java `DGEMM` percentage was the same as the Fortran `DGEMM` percentage – that would imply `DGEMM` in Java ran, on average, 5.25 times slower than `DGEMM` in Fortran.

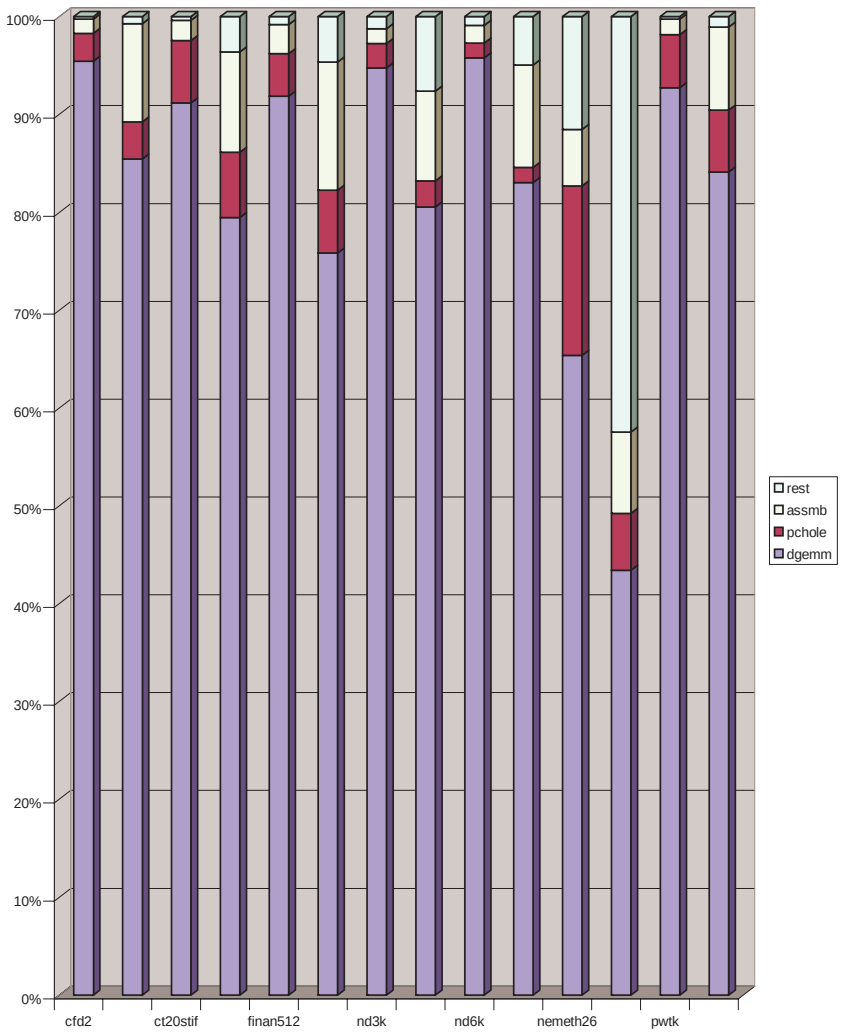

Figure 5.3: Visualization of the HB Factorization percentage breakdown

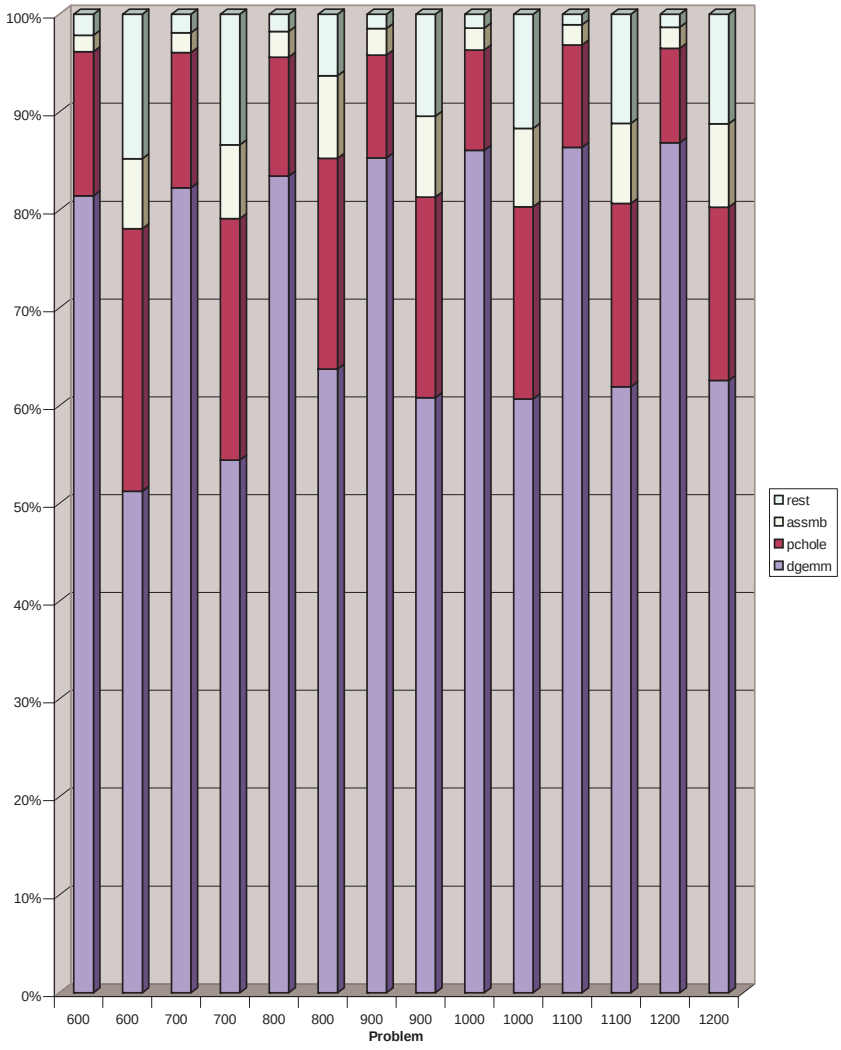

Figure 5.4: Visualization of the Grid Factorization percentage breakdown

	DGEMM		PCHOLE		ASSMB		Rest	
Problem	Java	Fortran	Java	Fortran	Java	Fortran	Java	Fortran
cfd2	95.45%	85.47%	2.84%	3.78%	1.46%	10.04%	0.25%	0.72%
ct20stif	91.20%	79.46%	6.36%	6.67%	2.08%	10.28%	0.37%	3.59%
finan512	91.86%	75.84%	4.36%	6.43%	3.00%	13.10%	0.78%	4.63%
nd3k	94.77%	80.53%	2.50%	2.69%	1.49%	9.19%	1.23%	7.59%
nd6k	95.81%	83.03%	1.50%	1.55%	1.80%	10.48%	0.90%	4.94%
nemeth26	65.38%	43.40%	17.31%	5.81%	5.77%	8.34%	11.54%	42.45%
pwtk	92.71%	84.12%	5.44%	6.32%	1.61%	8.50%	0.24%	1.06%
600	81.43%	51.26%	14.76%	26.83%	1.68%	7.16%	2.13%	14.75%
700	82.28%	54.46%	13.81%	24.65%	2.04%	7.54%	1.87%	13.36%
800	83.46%	63.74%	12.16%	21.54%	2.65%	8.44%	1.73%	6.27%
900	85.31%	60.81%	10.51%	20.49%	2.71%	8.28%	1.47%	10.42%
1000	86.08%	60.70%	10.26%	19.64%	2.24%	8.00%	1.42%	11.66%
1100	86.40%	61.92%	10.49%	18.74%	2.03%	8.19%	1.08%	11.15%
1200	86.89%	62.59%	9.62%	17.70%	2.17%	8.52%	1.32%	11.19%
Average	**87.1%**	**67.7%**	**8.7%**	**13.1%**	**2.3%**	**9.0%**	**1.9%**	**10.3%**

Table 5.5: Breakdown of time spent in numeric factorization

However, Java spends a greater percentage of its time in `DGEMM` than Fortran; therefore, Java must run more than 5.25 times slower than Fortran in the `DGEMM` routine. Given that we know how much slower the numeric factorization routine runs in Java as opposed to Fortran (5.25 times), and the respective breakdown of time spent in the numeric factorization (Table 5.5), we can derive an approximation of how much slower each area of the numeric factorization is relative to Fortran (Table 5.6).

Table 5.6 presents the relative performance data for the numeric factorization routine. We will present an example showing how the entries of the table were calculated. Java runs, on average, 5.21 times slower than Fortran during the numeric factorization phase using the client VM. From Table

5.5, Java spends an average of 87.1% of its numeric factorization time in `DGEMM` and Fortran spends 67.7% of its time in `DGEMM`. Therefore, the time spent in `DGEMM` of J-SPARSPAK is $\frac{.871 \times 5.21}{.677} = 6.70$ times greater than that of SPARSPAK.

	DGEMM	**PCHOLE**	**ASSMB**	**REST**
Ratio	6.70	3.46	1.33	0.96

Table 5.6: Java to Fortran Ratio for Numeric Factorization Functions

We have discoverd that J-SPARSPAK running in the client VM spends the majority of its numeric factorization time in the BLAS routine `DGEMM`. This routine runs 6.7 times slower in Java than Fortran – significantly worse than any other portion of J-SPARSPAK relative to SPARSPAK. It is highly likely that J-SPARSPAK running in the server VM exhibits similiar behaviour (with perhaps better `DGEMM` performance as the numeric factorization runs faster). Improving Java performance relative to Fortran for our application requires a fast and efficient numeric factorization, which in turn depends on `DGEMM`. In the subsequent Chapter, we will analyze this routine, its purpose, and attempt to determine why Java exhibits such poor performance in this routine relative to Fortran.

Chapter 6

DGEMM

6.1 What is DGEMM?

`DGEMM` calculates, in double-precision arithmetic, a matrix-matrix product[1] and addition for real general matrices or their transposes[2]:

$$\boldsymbol{C} = \alpha \times \mathrm{op}(\boldsymbol{A}) \times \mathrm{op}(\boldsymbol{B}) + \beta \times \boldsymbol{C},$$

where $\mathrm{op}(\boldsymbol{X})$ is either $\boldsymbol{X}$ or $\boldsymbol{X^T}$, α and β are scalars, and $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ are real matrices. The designation $\mathrm{op}(\boldsymbol{A})$ is an $m \times k$ matrix, $\mathrm{op}(\boldsymbol{B})$ a $k \times n$ matrix, and $\boldsymbol{C}$ an $m \times n$ matrix. The Fortran function signature is as follows:

```
SUBROUTINE DGEMM ( TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB,
                   BETA, C, LDC )
```

`TRANSA` and `TRANSB` specify whether $\boldsymbol{A}$ and $\boldsymbol{B}$ should be transposed respectively, `m`, `n`, `k`, `alpha`, and `beta` are as defined above. The parameters

[1] Ironically, the primary performance requirement for solving large sparse positive definite systems of linear equations using direct methods is the ability to perform a fast and efficient dense matrix multiplication. This is no fluke – 25 years of intensive sparse matrix research have made this possible. By reducing the problem to manipulation of dense matrices, algorithms achieve much better data locality of reference and as a result exhibit fewer cache misses.

[2] http://www.netlib.org/blas/dgemm.f

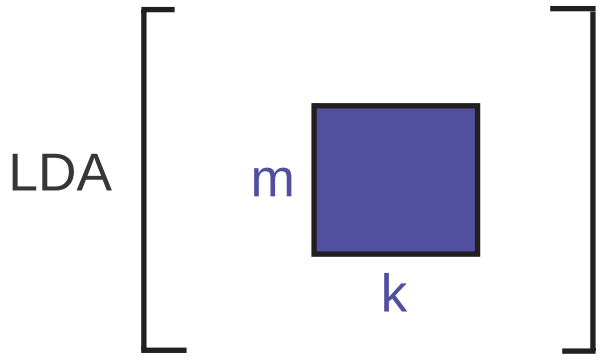

Figure 6.1: Visualization of a Fortran array passed into DGEMM

`LDA`, `LDB`, and `LDC` represent the leading dimension of the data pointed to by `A`, `B`, and `C` respectively. Recall that in Fortran, one can pass portions of arrays into functions. In Figure 6.1, we present a visualization of a Fortran array (using the dimensions of $\boldsymbol{A}$) passed into `DGEMM`. The matrix $\boldsymbol{A}$ actually used in the multiplication is an $m \times k$ rectangular portion of the memory block passed into `DGEMM`. `LDA` is necessary to determine the appropriate index into the memory block when accessing elements of $\boldsymbol{A}$.

J-SPARSPAK uses a version of `DGEMM` created by an automatic Fortran 77 to Java converter[3]. The routine treats each of the four possible matrix orientations ($\boldsymbol{A}$ or $\boldsymbol{A^T}$ coupled with either $\boldsymbol{B}$ or $\boldsymbol{B^T}$) separately. The `LDLTFACTOR` routine has three different calls to `DGEMM`:

1. `dgemm ('n', 't', jlen, nj, nk, -one, lnz(klpnt), ksuplen, temp2, nj, one, lnz(jlpnt), jlen )`

2. `dgemm ('n', 't', klen, nups, nk, -one, lnz(klpnt), ksuplen, temp2, width, one, lnz(ilpnt), jlen )`

3. `dgemm ('n', 't', klen, nups, nk, -one, lnz(klpnt), ksuplen, temp2, width, 0.0d0, temp, klen )`

[3]See Appendix A for the Fortran 77 code given to the converter

All three calls to `DGEMM` use the matrix orientations $\boldsymbol{A}$ and $\boldsymbol{B^T}$ (indicated by `'n'` and `'t'` as the first two parameters), thus allowing us to focus our efforts on the portion of the routine dedicated to that case.

Similar to our manual conversion effort, the automatic converter had to deal with issues such as passing portions of arrays into functions. Like us, they provided an integer offset for every array, emulating Fortran behaviour. In Figures 6.2, we present a subset of the Fortran 77 code given to the automatic converter, namely that which is responsible for the multiplication when the matrix orientations $\boldsymbol{A}$, $\boldsymbol{B^T}$ are supplied[4].

We will briefly describe the way the Fortran code performs the matrix-matrix multiplication. The outer loop (`J`) runs from `1` to `N` – at the end of one iteration of the outer loop, an entire column of `C` will have been computed. The column of `C` is first multiplied by `beta` (if `beta` does not equal one). Note that in the event `beta` equals zero, the column of $\boldsymbol{C}$ is explicitly set to zero, as opposed to being multiplied by zero. A difference in the way arrays are allocated between Fortran and Java could render this step redundant in Java. Namely, all elements of a numeric Java array are initialized to zero upon allocation, whereas the values in a newly allocated Fortran array are dependent on the bit pattern existing prior to allocation. Thus, if `C` is newly allocated, and `beta` equals zero, then explicitly setting `C` equal to zero prior to performing the multiplication is redundant. Although probably a minor issue in our case (potentially adding $m \times n$ operations to each multiplication), it is not obvious and spotting it requires fairly extensive knowledge of Java run-time semantics.

The algorithm computes `C[*,J]` by first scaling it by `BETA`, and then adding the matrix-vector product (the result of which is a vector) `A x B[*,J]` (`A` multiplied by column `J` of `B`).

In Figure 6.3, we present the Java code produced by the automatic converter (call this DGEMM version 1). In addition to being less readable than

[4]Note that the Fortran code given to F2J is not the same code used in SPARSPAK – we will elaborate later in the chapter

```
*
*           Form  C := alpha*A*B' + beta*C
*
            DO 170, J = 1, N
               IF( BETA.EQ.ZERO )THEN
                  DO 130, I = 1, M
                     C( I, J ) = ZERO
  130             CONTINUE
               ELSE IF( BETA.NE.ONE )THEN
                  DO 140, I = 1, M
                     C( I, J ) = BETA*C( I, J )
  140             CONTINUE
               END IF
               DO 160, L = 1, K
                  IF( B( J, L ).NE.ZERO )THEN
                     TEMP = ALPHA*B( J, L )
                     DO 150, I = 1, M
                        C( I, J ) = C( I, J ) + TEMP*A( I, L )
  150                CONTINUE
                  END IF
  160          CONTINUE
  170       CONTINUE
```

Figure 6.2: The Fortran 77 code given to F2J

```
1  for (j = 1; j <= n; j++) {
2   if (beta == zero)  {
3     for (i = 1; i <= m; i++) {
4        c[(i)- 1+(j- 1)*Ldc+ _c_offset] = zero;
5      }
6    }
7    else if (beta != one)  {
8      for (i = 1; i <= m; i++) {
9        c[(i)- 1+(j- 1)*Ldc+ _c_offset] =
10           beta*c[(i)- 1+(j- 1)*Ldc+ _c_offset];
11     }
12   }
13   for (l = 1; l <= k; l++) {
14     if (b[(j)- 1+(l- 1)*ldb+ _b_offset] != zero)  {
15       temp = alpha*b[(j)- 1+(l- 1)*ldb+ _b_offset];
16       for (i = 1; i <= m; i++) {
17         c[(i)- 1+(j- 1)*Ldc+ _c_offset] =
18               c[(i)- 1+(j- 1)*Ldc+ _c_offset]+
19               temp*a[(i)- 1+(l- 1)*lda+ _a_offset];
20       }
21     }
22   }
23 }
```

Figure 6.3: Java code resulting from the conversion (Java DGEMM version 1)

```
int lowerLimit = -1 + (j-1)*Ldc + _c_offset;
int upperLimit = lowerLimit + m;
for(int i = lowerLimit; i <= upperLimit; i++)
  c[i] = zero;
```

Figure 6.4: More efficient scaling of a column of C

the corresponding Fortran code in Figure 6.2, we see that computing each array index requires four additions and a multiplication. Our first attempt to optimize the routine involved decreasing the amount of work required to compute each array index. For example, examine the section of code where `c[*,j]` is set to zero (lines 3 and 4 of Figure 6.3). We are successively accessing elements of column `j` of `c`, and these elements are stored consecutively in memory. Examining the index into `c`, we see that the only variable changing over the course of the loop is `i`. We suggest the code in Figure 6.4 as an alternative, which is semantically equivalent, but more efficient.

In Figure 6.4, we present code which eliminates redundant computations (at the expense of creating two temporary integers), while maintaining semantics. Notice that we have explicitly created a temporary variable representing the lower limit of the loop. This is unnecessary as substituting the expression used to calculate `lowerlimit` into the definition of the loop would be equally efficient – the expression would only be evaluated once. The temporary `upperLimit` should improve efficiency unless the compiler could have otherwise proved that `upperLimit` is invariant and move it out of the loop.

We re-wrote version 1 of the routine (Figure 6.3) with the goal of removing as many redundant array indexing computations as possible, and present the result (DGEMM version 2) in Figure 6.5. Compare the amount of work performed indexing the arrays in DGEMM version 1 relative to version 2 – the indexes in version 2 require at most an addition, with most requiring no computation. When multiple arrays are accessed inside a loop, such as the inner most loop (lines 22-23), we must choose one index to optimize. In

Problem	Version 1	Version 2
cfd2	956.96	775.83
ct20stif	67.89	50.72
finan512	58.38	56.58
nd3k	348.83	277.01
nd6k	2059.35	2054.83
nemeth26	4.30	2.81
pwtk	328.58	236.25
ratio to fortran	**4.21**	**3.41**
600	24.86	13.20
700	37.50	20.89
800	60.18	35.73
900	97.76	63.58
1000	125.68	80.30
1100	167.19	110.39
1200	202.34	142.59
ratio to fortran	**3.00**	**1.87**
combined ratio	**3.60**	**2.64**

Table 6.1: Numeric factorization time comparision of DGEMM versions 1 and 2

lines 22-23, we arbitrarily chose to optimize the index into `c`. In tailoring the code to efficiently index `c`, we had to take special care to index `a` properly by subtracting that which was added to loop iteration values (line 16). The code of version 2 is even less readable than that of version 1.

In Table 6.1 we present benchmark results for the numeric factorization of J-SPARSPAK using DGEMM version 2. We see that with version 2, the ratio of Java performance relative to Fortran improved from a factor of 4.21, to 3.41 (19.0%) over all HB problems. Similarly, over all grid problems, the ratio improved from 3.00 to 1.87 (37.6%). Overall, the solution time ratio improved from 3.60 to 2.64 (26.7%).

```
1  int tempc = -ldc;
2  int tempb = -ldb + b_offset;
3  for( int j = 1; j <= n; j++ ) {

4    tempc += ldc;
5    tempb += ldb;

6    if( beta == zero ) {
7      int upLimit = tempc + c_offset + m;
8      for( int i = tempc + c_offset + 1; i <= upLimit; i++ )
9        c[i] = zero;
10     }
11   else if( beta != one ) {
12     int upLimit = tempc + c_offset + m;
13     for( int i = tempc + c_offset + 1; i <= upLimit; i++ )
14       c[i] *= beta;
15   }

16   int tempa = -lda + a_offset - tempc - c_offset;

17   for( int l = 1; l <= k; l++ ) {
18     tempa += lda;
19     if( b[ l + tempb ] != zero ) {
20       double temp = alpha * b[ l + tempb ];
21       int upLimit = tempc + c_offset + m;
22       for( int i = tempc + c_offset + 1; i <= upLimit; i++ )
23         c[ i ] *= temp * a[ i + tempa ];
24       }
25   }
26 } // end for( j=1... )
```

Figure 6.5: Java DGEMM version 2 - redundant array index computations eliminated

The increase in performance yielded by version 2 of DGEMM exemplifies the lack of Java compiler maturity. Version's 1 and 2 of DGEMM are semantically equivalent - a mature optimizing compiler should be able to recognize and optimize version 1 of DGEMM to (at least) the point of version 2 using loop invariant hoisting and induction variable analysis. Indeed, when the two routines were coded in Fortran 90 and compiled with maximum optimizations, they performed equivalently . When compiled with no optimizations, version 2 out-performed version 1 by 29%. Also noteworthy, when the DGEMM routine presented in Appendix A was compiled in CVF with maximum optimizations, it ran at the same speed as DGEMM versions 1 and 2, all of which verifies earlier allegations that poor Java performance is in large part due to immature compiler technology.

As we have seen however - a number of compiler optimizations can be applied by hand. With version 2 of DGEMM, we improved numeric factorization performance by 26.7%; however, Java is still running 2.67 times slower than Fortran – better, but still unacceptably slower than the other phases. Nonetheless, a viable option for Java programmers developing high performance numerical codes is to hand-tune critical code sections for performance. As we have seen, hand-tuning code comes at the expense of code-readability. In addition, performance improvements yielded by hand-tuned code may not yield high performance on different platforms. Since one of Java's main strength is cross-platform compatiblity, this is a significant issue.

Our next step was to examine the DGEMM routine being used by SPARSPAK. Compaq Visual Fortran ships with a library called the 'Compaq Extended Math Library'[5] (CXML). CXML includes a complete LAPACK implementation, as well as implementations of BLAS levels 1, 2, and 3. Recall that DGEMM resides in BLAS level 3. CXML employs a number of techniques to achieve high performance in numerical codes. The hierarchical memory system is efficiently managed by enhancing the data locality of reference. For example, the algorithms are structured to operate on sub-blocks of arrays

[5]http://h18000.www1.hp.com/math/documentation/cxml/dxml.3dxml.html

that are sized to remain in cache until all operations involving the data in the sub-block are complete[6]. Performance of the CXML DGEMM routine trumped that of the Fortran routine presented in Appendex A compiled with maximum optimizations by close to a factor of three.

Unfortunately, Compaq has not released the source code for the CXML library. We attempted to contact Compaq and request the source code, but did not get a response. It is possible that the code in the library was created, or at least tweaked, at the assembly level, rendering the source code useless as far as a conversion into Java is concerned. Regardless, not knowing the specific techniques used to coax performance out of the CXML DGEMM routine merited no further investigation.

6.2 IBM Array Package Revisited

Recall from Section 2.2, the NINJA group at IBM developed a library called the IBM Array Package. In addition to classes simulating true multi-dimensional arrays, the package also includes an implementation of BLAS routines utilizing the multi-dimensional array classes. Our next step was to compare the version of DGEMM included in the IBM Array Package with our version of DGEMM. Unfortunately, integrating the IBM DGEMM into J-SPARSPAK was not as easy as simply replacing the 3 calls to DGEMM from `LDLTFACTOR` with the IBM version. The IBM DGEMM has the following function signature

```
dgemm(int transA, int transB, double alpha, doubleArray2D A,
      doubleArray2D B, double beta, doubleArray2D C)
```

Notice that the arrays `A`, `B`, and `C` are all of type `doubleArray2D`, the IBM Array Package class encapsulating an immutable 2-dimensional rectangular array of doubles. We cannot pass objects of type `double[]` (as present in J-SPARSPAK) into the IBM DGEMM routine, preventing integration.

[6]http://h18000.www1.hp.com/math/documentation/cxml/cxmlref.pdf

One option is conversion of the arrays passed into DGEMM into arrays of type `doubleArray2D` immediately prior to each call site – unfortunately that would require copying all of the data in the array negating any advantages yielded by a more efficient DGEMM implementation. A second option would be to modify J-SPARSPAK such that all arrays to be passed into the IBM DGEMM routine were of type `doubleArray2D` for the lifetime of the program. This approach would require a significant amount of effort and used only as a last resort.

At first glance, it appeared as though the IBM DGEMM routine did not support offsets into the arrays, further complicating integration into J-SPARSPAK. However, the array classes in the IBM Array Package support efficient *array sectioning* in a fashion similar to Fortran, negating the need for array offsets. Support of array sectioning is efficient in the sense that no data-copying occurs, but still less efficient than Fortran as a new object must be created (and eventually deleted) to store sectioning information.

In order to compare the IBM DGEMM routine to versions 1 and 2 of our DGEMM routine, we decided to perform one large (i.e., 1000×1000) matrix multiplication in each of the routines. To test whether one large multiplication is representative of many thousand small multiplications (as occuring in SPARSPAK, see Appendix B), we ran a 1000×1000 problem on DGEMM versions 1 and 2. DGEMM version 1 performed the multiplication in 27.46 seconds, and version 2 required 20.21 seconds. Recall that in J-SPARSPAK, version 2 performed on average 27% faster than version 1. With one large multiplication, version 2 runs 26.5% faster than version 1, validating usage of one large multiplication for subsequent DGEMM tests.

Running the 1000×1000 problem on the IBM DGEMM routine using matrix orientations $\boldsymbol{A}$ and $\boldsymbol{B^T}$ resulted in a time of 7.46 seconds! As a comparison, Fortran with maximum optimizations executing the routine in Appendix A on the same problem runs in slightly over 11 seconds. The CXML DGEMM routine used in SPARSPAK completes in 4.21 seconds. The IBM DGEMM routine must be using some of the same techniques used in the

CXML routine (which are not performed by the optimizing compiler shipping with CVF). Integrating the IBM DGEMM routine into J-SPARSPAK became a much higher priority. Like the CXML, IBM did not release the source code for the array package. We emailed the developers of the IBM Array Package requesting the source code but did not receive a response. Fortunately, we had the class files available to us, and a number of free Java decompilers (.class to .java) are readily available.

6.2.1 Decompilation

Using the DJ Java Decompiler[7], we decompiled both the IBM DGEMM routine, and the implementation of the class `doubleArray2D`. The result of decompiling the IBM DGEMM routine was 214 lines long! Analysis of the decompiled code revealed that a combination of blocking and loop unrolling, both of which aim to improve data locality, can be attributed to the increased performance. We will now describe in detail the internal representation of the `doubleArray2D` class and how the IBM DGEMM routine uses the objects to perform the multiplication. Understanding the workings of these two designs is key to J-SPARSPAK integration.

The doubleArray2D Class

A `doubleArray2D` object represents a 2-dimensional, immutable, rectangular array using 6 values:

1. a data storage pointer `(double [] data)`
2. a 2-element shape vector `(int n0, n1)`(the number of rows and columns respectively)
3. a 3-element weights vector `(int w0, w1, w2)` (describes placement of elements in the data vector)

[7]http://members.fortunecity.com/neshkov/dj.html

Element `i,j` of a `doubleArray2D` object is accessed in the following fashion: `data[i*w0 + j*w1 + w2]`. By default, `w0` is initialized to `n1` (the number of columns), `w1` is initialized to one, and `w2` is initialized to zero. By changing the values of `w0`, `w1`, and `w2`, it is clear how the `doubleArray2D` class can support sectioning. The implementation of the `get` call reveals another technicality which must be accounted for if an attempt to integrate into J-SPARSPAK is made: by default the `doubleArray2D` class uses row-wise (C-style) storage as opposed to column-wise (used in SPARSPAK AND J-SPARSPAK). Therefore, the matrix orientations $\boldsymbol{A^T}$, $\boldsymbol{B}$ must be used to achieve conformance with J-SPARSPAK.

The IBM DGEMM Routine

We will now describe the implementation of the IBM DGEMM routine. At a high level, the IBM DGEMM routine computes 4×4 blocks of $\boldsymbol{C}$ at a time, using 4 rows of $\boldsymbol{A}$ and 4 columns of $\boldsymbol{B}$. Like all other matrix-matrix multiplication routines encounterd so far, the routine consists of 3 nested loops, one for each of the unique matrix dimensions.

The inner-most loop runs from 1 to the number of (columns in $\boldsymbol{A}$/rows in $\boldsymbol{B}$) in strides of one. The data access pattern after the inner-most loop completes is shown in Figure 6.6 (a) – 4 rows of $\boldsymbol{A}$ and 4 columns of $\boldsymbol{B}$ are used to compute 4×4 blocks of $\boldsymbol{C}$. The 16 elements of $\boldsymbol{C}$ are not accessed for the remainder of the multiplication.

The middle loop runs from 1 to the number of columns in $\boldsymbol{B}$ in strides of 4. The same 4 rows of $\boldsymbol{A}$ are repeatedly multiplied by 4 different columns of $\boldsymbol{B}$, to compute a unique 4×4 block of $\boldsymbol{C}$. In the event the number of columns of $\boldsymbol{C}$ is not a multiple of 4, an epilogue exists to the middle loop to compute the remaining rows of $\boldsymbol{C}$ in 4×1 sections. After the middle loop and its epilogue finish, 4 entire rows of $\boldsymbol{C}$ have been computed (see Figure 6.6 for the data access pattern).

The outer loop runs from 1 to the number of rows in $\boldsymbol{C}$, in strides of 4, with an epilogue outside the loop to compute any remaining rows of $\boldsymbol{C}$ in the

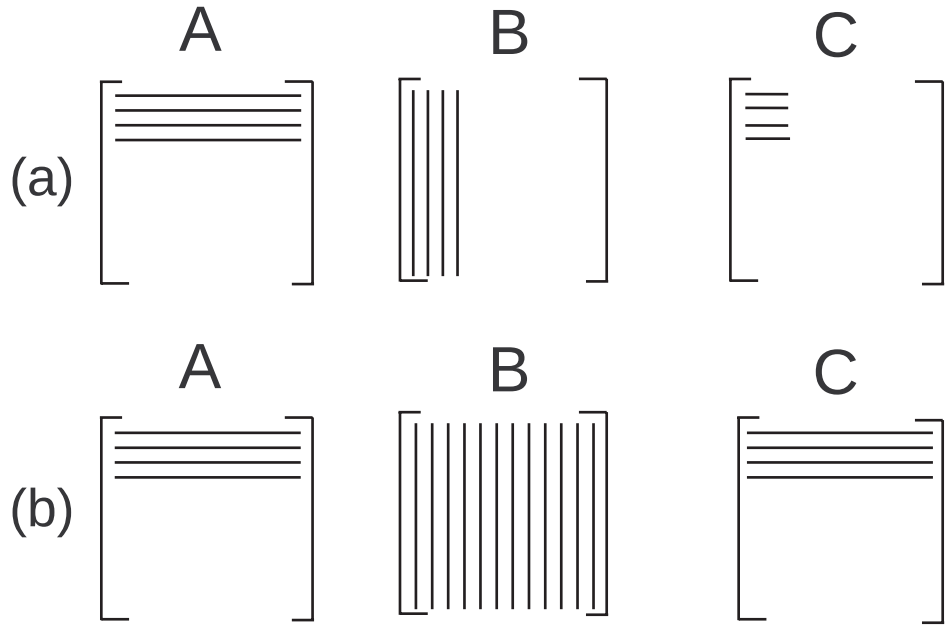

Figure 6.6: Data access patterns after one iteration of the middle (a) and outer (b) loops respectively

event the number of rows in $\boldsymbol{C}$ is not a multiple of 4 – these are computed one by one.

The IBM DGEMM routine uses the same access pattern depicted in Figure 6.6 regardless of the matrix orientations. To observe the effect of matrix orientation on performance, we ran the 1000×1000 problem using the four possible matrix orientations and the results were surprising. Referring to Table 6.2, we see that the multiplication times range from 7.46s, to 14.46s! We will now describe what occurs inside the `doubleArray2D` class when a transpose operation is applied, in the hopes of understanding the cause of the performance difference. Changing the matrix orientation of a `doubleArray2D` object amounts to changing the values of the member variables `w0` and `w1`. The call `A.get(1,5)` where `A` is a `doubleArray2D` object returns `data[1*w0 + 5*w1 + w2]` (where, by default, `w0` is initialized to the number of columns

in the matrix, `w1` is initialized to one, and `w2` to zero). If one requests that `A` be transposed, the member variables `w0` and `w1` get swapped, as do the member variables `n0` and `n1`. A subsequent call `A.get(x,y)` returns the equivalent of the call `A.get(y,x)` before the transpose operation, as required.

In the context of the IBM DGEMM routine, supplying different matrix orientations to the routine results in different data storage orientations (i.e., instructing the routine to use $\boldsymbol{A^T}$ instead of $\boldsymbol{A}$ means $\boldsymbol{A}$ is stored in column-major instead of row-major order). Thus, the performance difference is entirely attributable to different memory access patterns.

Matrix Orientation	**Multiplication Time**
$\boldsymbol{A}$, $\boldsymbol{B}$	13.84
$\boldsymbol{A^T}$, $\boldsymbol{B}$	14.46
$\boldsymbol{A^T}$, $\boldsymbol{B^T}$	9.14
$\boldsymbol{A}$, $\boldsymbol{B^T}$	7.46

Table 6.2: IBM DGEMM Matrix-Matrix multiplication times for the four different matrix orientations

Note that none of the allowable matrix orientations in the IBM Array Package coincide directly with what J-SPARSPAK requires, namely $\boldsymbol{A}$ and $\boldsymbol{C}$ stored column-wise and $\boldsymbol{B}$ stored row-wise. Fortunately, we doubt the storage of $\boldsymbol{C}$ has much impact on performance – every element of $\boldsymbol{C}$ is accessed only twice during the course of a multiplication, once to retrieve the value existing prior to the multiplication, and once to set the new value. Furthermore, the routine never iterates across rows or down columns of $\boldsymbol{C}$, it retrieves (and sets) 4×4 blocks at a time.

Let us now examine the different memory access patterns to determine which result in best performance, hopefully allowing us to modify our DGEMM routine to use techniques similiar to that used in the IBM DGEMM routine, while maintaining conformance with the calling conventions and expected data-layout of J-SPARSPAK.

First, we examine the memory layout of the fastest multiplication, $\boldsymbol{A}$ with $\boldsymbol{B^T}$. With this orientation, $\boldsymbol{A}$ is stored row-wise, and $\boldsymbol{B}$ is stored column-wise. Referring again to Figure 6.6, we see that to compute a 4×4 block of $\boldsymbol{C}$, we iterate across 4 rows of $\boldsymbol{A}$ and down 4 columns of $\boldsymbol{B}$. It is well-known that accessing elements in the *natural order* (e.g., column-major for 2-dimensional Fortran array, row-major for 2-dimensional C arrays) results in best performance. Performance of the $\boldsymbol{A}$, $\boldsymbol{B^T}$ multiplication is testament to this – both $\boldsymbol{A}$ and $\boldsymbol{B^T}$ are accessed in the natural order.

Turning our attention to the memory layout of the slowest multiplication ($\boldsymbol{A^T}$, $\boldsymbol{B}$), we see that both matrices are accessed orthogonal to the natural order – further evidence that traversing arrays along the natural order is key to high performance. Looking at the cases where one array is traversed in the natural order and one is not, we see that the case $\boldsymbol{A}$, $\boldsymbol{B}$ performs worse than $\boldsymbol{A^T}$, $\boldsymbol{B^T}$. We propose the following explanation, consistent with our natural order explanation presented earlier. Recall that 4 rows of $\boldsymbol{C}$ are computed at a time, in 4×4 blocks. In computing each 4×4 block, *the same* 4 rows of $\boldsymbol{A}$ are repeatedly multiplied by 4 *different* columns of $\boldsymbol{B}$. In computing 4 rows of $\boldsymbol{C}$, only 4 rows of $\boldsymbol{A}$ are traversed (after which they are in cache), whereas the entirety of $\boldsymbol{B}$ is traversed. Thus, the penalty for traversing $\boldsymbol{A}$ out of order is significantly less than the penalty for traversing $\boldsymbol{B}$ out of order, and is reflected in the multiplication times.

Another note of interest is the amount of computation being performed to index each element of a `doubleArray2D` object, namely 3 additions and 2 multiplications. We are surprised that such decent performance figures were achieved, given the performance improvement we saw in moving from DGEMM version 1 to DGEMM version 2. It is possible that performance of the IBM DGEMM routine can be improved using techniques similar to those used in transforming DGEMM version 1 to DGEMM version 2.

6.3 DGEMM Version 3

Armed with the keys to improving matrix-matrix multiplication performance from the previous section, we set out to write a third J-SPARSPAK compatible DGEMM routine. We define the following requirements for a J-SPARSPAK compatible DGEMM routine:

- accepts arrays $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ as single dimensional `double` arrays
- $\boldsymbol{A}$ and $\boldsymbol{C}$ are stored column-wise, and $\boldsymbol{B}$ is stored row-wise
- all matrices support offsets

Although not a requirement, we would also like the matrices to be accessed in their natural order whenever possible.

In Figure 6.7, we present the sizes and data layout of the matrices $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$. We want the algorithm to iterate down columns of $\boldsymbol{A}$, across the rows of $\boldsymbol{B}$, and down the columns of $\boldsymbol{C}$. We thus propose a multiplication routine in which the inner-most loop runs to m (iterating down the columns of $\boldsymbol{A}$ and $\boldsymbol{C}$), the middle loop runs to n (across the rows of $\boldsymbol{B}$), and the outer-loop to k. Such a scheme results in iteration of all matrices in their natural order – although we suspect not much gain is achieved by traversing $\boldsymbol{B}$ in order as, between accessing 4 elements of $\boldsymbol{B}$, 4 entire columns of $\boldsymbol{A}$, and 1 entire column of $\boldsymbol{C}$ is accessed (potentially pushing elements of $\boldsymbol{B}$ out of cache).

In Figure 6.8, we present the data access patterns of DGEMM version 3. The data access pattern after one iteration of the inner-most loop is shown in Figure 6.8 (a). The routine computes the product of an $m \times 4$ section of $\boldsymbol{A}$ and an 4×1 section of $\boldsymbol{B}$, the result of which is a $m \times 1$ section of $\boldsymbol{C}$. The 4 columns of $\boldsymbol{A}$ are repeatedly multiplied by a different 4×1 section of $\boldsymbol{B}$. After one iteration of the outer loop, every element of $\boldsymbol{C}$ has been updated. Subsequent iterations of the outer loop use 4 different columns of $\boldsymbol{A}$ with 4 different rows of $\boldsymbol{B}$ to again update every element of $\boldsymbol{C}$.

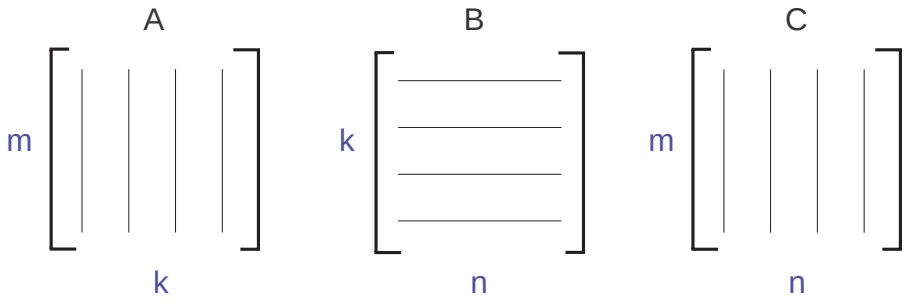

Figure 6.7: Sizes and data layout of matrices $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$

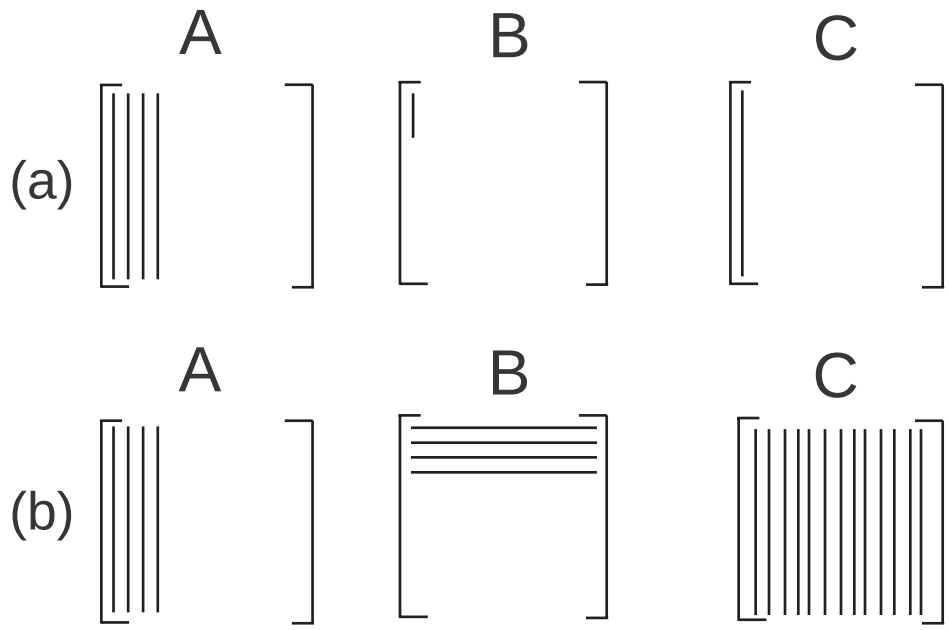

Figure 6.8: Data access patterns after one iteration of the middle (a) and outer (b) loops respectively

DGEMM version 3 performs a 1000×1000 multiplication in 8.84 seconds, 11.2 seconds faster than DGEMM version 2, and only 1.39 seconds slower than the IBM DGEMM routine with the optimal data layout. The IBM DGEMM routine achieves better data locality due to the potentially poor

Problem	Version 1	Version 2	Version 3
cfd2	956.96	775.83	418.86
ct20stif	67.89	50.72	32.59
finan512	58.38	56.58	28.37
nd3k	348.83	277.01	145.23
nd6k	2059.35	2054.83	877.67
nemeth26	4.30	2.81	3.42
pwtk	328.58	236.25	161.95
ratio to fortran	**4.21**	**3.41**	**2.14**
600	24.86	13.20	15.11
700	37.50	20.89	21.69
800	60.18	35.73	33.65
900	97.76	63.58	51.91
1000	125.68	80.30	65.61
1100	167.19	110.39	96.45
1200	202.34	142.59	105.84
ratio to fortran	**3.00**	**1.87**	**1.67**
combined ratio	**3.60**	**2.64**	**1.90**

Table 6.3: Numeric factorization time comparision of DGEMM versions 1, 2, and 3

cache performance when accessing $\boldsymbol{B}$ in DGEMM version 3.

In Table 6.3, we present the timings of the numeric factorization routine for DGEMM versions 1, 2, and 3. Version 3 improves the HB ratio to 2.14 from 3.41, and the grid ratio from 1.87 to 1.67. The total ratio over all problems improved from 2.64 down to 1.90 – the numeric factorization routine is now on par with the ordering, symbolic factorization, and triangular solve routines relative to Fortran.

Notice that in most cases, DGEMM version 3 yields a substantial improvement over DGEMM version 2. However, with smaller problems, such as *nemeth26*, and the 600×600 and 700×700 grid problems, version 3 is

actually slightly slower than version 2. Examining the table of DGEMM call statistics in Appendix B, we see that the 3 problems which ran slower with DGEMM version 3 are those which, on average, perform the fewest number of floating-point operations per multiplication. DGEMM version 3 advances DGEMM version 2 by performing more operations per memory fetch. However, that technique relies on those matrix dimensions iterated over in strides of four being of at least size 4 (e.g., the outer loop runs from 1 to k in strides of 4). In the event k is less than 4, version 3 essentially reduces to version 2 (i.e., 1 row/column at a time) with the upper limit of the outer and middle loops swapped (i.e., version 3 goes to k, n, and m in that order, and and version 2 goes to n, k, and m). We are unsure why this would yield a performance improvement, but it is not an issue since it is only slight. In conclusion, increased performance of DGEMM version 3 yields the most improvement when relatively large matrices are supplied to DGEMM, such as with the problem *nd6k*.

Interestingly, as the density of the problem increases, the ratio of the numeric factorization time for DGEMM versions 2 and 3 approaches the ratio of the 1000 multiplication for DGEMM version 2 and 3 (20s vs 8s), even though, in the case of *nd6k*, 39772 calls are made, with an average of 2.5 million operations per multiplication (a total of 100 billion). Contrast that with one call consisting of one billion operations in the case of the 1000×1000 multiplication. We suspect that once the matrices reach a certain size, the time required for the multiplication increases linearly with the matrix size.

In conclusion, we have improved DGEMM performance by almost a factor of 2 since version 1, bringing Java performance within a factor of 2 of Fortran and on par with the other 3 routines. Eliminating inefficient array indexing resulting from simulation of 2-dimensional arrays by 1-dimensional arrays yielded a performance improvement of 27%. The remaining improvement was achieved by making better usage of the memory cache, increasing the number of operations performed per memory fetch.

Chapter 7

Future Work and Conclusion

7.1 Future Work

With a thesis of this scope, a number of branches for future work have been exposed. We essentially made a 'beeline' for the best possible performance in J-SPARSPAK, without much investigation as to the specific reasons for Java's performance deficiencies. Broadbrush generalizations such as 'immature compiler technology' were cited as culprits – the community would benefit from more specific information (e.g., the Java compiler was unable to eliminate array bounds checks from this section of code because the loop upper limit variable was not declared as final decreasing performance by X%). We would ideally like to determine the exact cause of performance difference beteween identical Fortran and Java code. Doing so would precisely identify the hurdles present and the direct effect of current compiler technology on solving symmetric large sparse positive definite systems of equations using direct methods.

Although we focused on performance of 100% pure Java code, it would be interesting to call the CXML `DGEMM` routine from J-SPARSPAK through the JNI and measure resulting performance. We suspect that the asymptotic performance of J-SPARSPAK would approach that of SPARSPAK, as, for sufficiently large problems, solution time is almost entirely dependent on

DGEMM performance. Although, in the general case, robustness is sacrificed when the JNI is employed, the CXML library is very mature and most likely bug-free. In an industrial setting, the advantage of the performance increase yielded by the CXML library would most likely outweigh the possibility of a slightly-less robust system which is no longer cross-platform compatible.

The optimal DGEMM performance on a 1000×1000 multiplication occurred when $\boldsymbol{A}$ was stored row-major, $\boldsymbol{B}$ column major, and $\boldsymbol{C}$ row-major. It would be possible to modify J-SPARSPAK to use those storage layouts for the three arrays, further improving DGEMM performance. Changing the storage layout of the matrices is unlikely to negatively affect performance of other areas of J-SPARSPAK.

We have evaluated Java vs. Fortran for one scientific computing application and have discovered Java code converted directly from Fortran code to run within a factor of 2. Although consistent throughout our application, it would be interesting to see whether other solvers achieve a similar ratio.

Given that the ability to optimize Java code depends on the ability to prove that same code free of exceptions, it would be useful to know what prevents the compiler from proving a certain section of code exception-free. Most mature Java programs do not contain any errors and therefore would not throw any exceptions. It would be a shame if a mature program could not be proved exception-free and thus could not be optimized. Therefore, determining how to write Java code conducive to optimization would be a useful contribution to the community.

7.2 Tips for achieving high-performance in numerical Java applications

On a more practical note, in this section, we will present tips and techniques we have discovered which improve performance of scientific applications in Java. When performance is a concern, it is important to simulate two-dimensional arrays with one-dimensional arrays, for a couple of reasons.

First, only one array-bounds/null-pointer check is required for each access (instead of two). More importantly however, a one-dimensional array is guaranteed to be stored contiguously in memory, improving data locality of reference when iterating over the elements. Whenever possible, arrays should be traversed in their natural order, improving cache performance.

JVM's for PC's are typically more advanced than those for less-prevalent platforms. As such, given similar hardware running different operating systems, the best performance is to be had on Linux/Windows machines.

7.3 Summary

We will present a brief summary of the project. We converted a sparse matrix solver written in Fortran 90 into Java. Issues encountered stemmed primarily from features present in Fortran 90 but not in Java. Unlike most Fortran environments, no 'instrinsic' version of BLAS is available for Java. As a result, we used a version of BLAS output by a Fortran to Java converter. Following that, we tested the Java version using a variety of JVM's and found the Sun JVM with the -server option to surpass the others in performance. Initial benchmarks comparing J-SPARSPAK to SPARSPAK showed the numeric factorization phase to dominate the solution process in both Fortran and Java and unfortunately, with the Java version running 3.61 times slower than the Fortran version. Given that the other phases ran within a factor of 2 of Fortran, we figured improvement of the numeric factorization routine was possible. Profiling the execution revealed the majority of the numeric factorization routine was spent in a matrix-matrix multiplication BLAS function, `DGEMM`. J-SPARSPAK was using a version of `DGEMM` created by the Fortran-to-Java converter, whereas SPARSPAK was using a highly-tuned `DGEMM` routine from the CXML library. We did not have access to either the source code for the CXML `DGEMM` routine, nor have the creators revealed the techniques used to achieve such high performance. Examining the code produced by the automatic Fortran-to-Java converter, we saw

that alot of redundant array indexing computations were being performed. Eliminating these improved performance by 27% (`DGEMM` version 2). Still, J-SPARSPAK was running 2.64 times slower than SPARSPAK in the numeric factorization routine – we suspected further performance improvements were possible. Time required to perform a 1000×1000 multiplication with the `DGEMM` routine from the IBM Array Package was 7.46 seconds, `DGEMM` version 2 performed the same multiplication in slightly over 20 seconds. Decompiling the IBM `DGEMM` routine revealed that loop unrolling and blocking had a part to play in the performance improvement. Benchmarking the routine with different matrix orientations also showed that accessing arrays in their natural order is critical for high performance. Based on that information, we created a third `DGEMM` routine. The new routine, `DGEMM` version 3, performed within a factor of 2 of the CXML `DGEMM` function, on par with the other solution phases relative to Fortran.

7.4 Conclusion

Our work has exposed deficiencies in the numerical libraries currently available for Java. Numerical programming courses encourage use of bug-free, mature numerical libraries whenever possible. Although good advice when programming in tradtional languages such as Fortran and C, numerical programming in Java currently requires a different approach. One can not assume that numerical libraries available for Java will yield optimum performance. As we have shown, significant tweaking of our `DGEMM` routine was required to achieve comparable performance to Fortran.

Re-usable numerical libraries, crucial to adoption of a language as a high-performance computing medium, are not yet mature in Java. Numerical libraries currently existing use proprietary data formats (i.e., `doubleArray2D`) which, unless adopted at the beginning of the project, are difficult to exploit (e.g., integration of the IBM `DGEMM` routine into J-SPARSPAK would have required changing all arrays to type `doubleArrayXD` where `X` is the rank of

the array).

The asymptotic performance of Java does not approach that of Fortran as one might predict given the depth of compiler optimizations that are applied by the server VM. We suspect this is either because the aggressivness of the optimizations still do not approach those applied by Fortran compilers, or issues created by the Java Language Specification (such as the need to prove sections of code to be free of exceptions prior to optimization) take a signifcant toll on performance.

As a general rule, Java code converted directly[1] from Fortran code will run within a factor of 2. Decreasing the performance gap depends on Java compiler technology maturing. We suspect that performance of Java code will eventually reach or even surpass the level of Fortran due to additional information available to the compiler at run-time. If performance greater than a factor of 2 is required, high-level compiler optimizations such as loop unrolling, loop invariant hoisting, and common subexpression elimination can be applied by hand to critical sections of code (as we did creating `DGEMM` version 2). Keep in mind that code-readability is adversely affected by manually applying compiler optimizations, detracting from code maintainability.

We would like to discuss the project from a practical point of view – namely what are the implications of our work towards conversion of high-performance numerical into Java, possibly in an industry setting. The path we took in the initial conversion effort could easily be followed by anyone converting software from Fortran into Java. Initial results would have shown Java to run almost 4 times slower than Fortan for sufficiently large problems - most likely unacceptable for all but trivial tasks. At this point, the conversion effort is likely to have been dropped and Java again labelled as unsatisfactory for applications demanding high performance. Even had the team behind the conversion recognized the importance of `DGEMM`, and used the IBM version (with all arrays of type `doubleArray2D`), it is possible that they would have used the 'fast' orientation in their tests, and the 'slow' orien-

[1] as close as language differences permit

tation in their application, not realizing there is a factor of two performance difference between the two. The problem, in our case, was one of immature numerical libraries. Had a tuned `DGEMM` function been used in J-SPARSPAK the first time around, it would have ran within a factor of two of Fortran immediately. Thus, our final conclusion is that Java, in its current state, has the ability to perform within an acceptable level of Fortran for solving large sparse symmetric positive definite systems of equations with access to mature numerical libraries.

The two primary contributions we have made to the community are as follows. First, numerical programs consisting of instrinsic language constructs (i.e., no complex numbers) converted from Fortran, with today's fastest Java JIT compiler, will run within a factor of two of Fortran. This includes the program J-SPARSPAK, a program designed to solve large symmetric sparse positive definite systems of equations using direct methods. Secondly, Java numerical libraries in their current state can not be thoughtlessly used by numerical programs. They must be tested and possibly modified before being employed if performance is a concern. In light of these two issues, in terms of performance, we feel that Java is indeed suitable for solving symmetric large sparse positive definite systems of equations using direct methods. However, in terms of ease of program development, mature and efficient numerical libraries running within a factor of two of equivalent Fortran libraries must be developed and readily available to the Java community. Regardless, significant progress has been made since Java's inception, and we feel scientific Java applications will eventually become ubiquitous.

Appendix A

```
      SUBROUTINE DGEMM ( TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B, LDB,
     $                   BETA, C, LDC )
*     .. Scalar Arguments ..
      CHARACTER*1        TRANSA, TRANSB
      INTEGER            M, N, K, LDA, LDB, LDC
      DOUBLE PRECISION   ALPHA, BETA
*     .. Array Arguments ..
      DOUBLE PRECISION   A( LDA, * ), B( LDB, * ), C( LDC, * )
*     ..
*
*  Purpose
*  =======
*
*  DGEMM  performs one of the matrix-matrix operations
*
*     C := alpha*op( A )*op( B ) + beta*C,
*
*  where  op( X ) is one of
*
*     op( X ) = X   or   op( X ) = X',
*
*  alpha and beta are scalars, and A, B and C are matrices, with op( A )
```

```
*  an m by k matrix,  op( B )  a  k by n matrix and  C an m by n matrix.
*
*  Parameters
*  ==========
*
*  TRANSA - CHARACTER*1.
*           On entry, TRANSA specifies the form of op( A ) to be used in
*           the matrix multiplication as follows:
*
*              TRANSA = 'N' or 'n',  op( A ) = A.
*
*              TRANSA = 'T' or 't',  op( A ) = A'.
*
*              TRANSA = 'C' or 'c',  op( A ) = A'.
*
*           Unchanged on exit.
*
*  TRANSB - CHARACTER*1.
*           On entry, TRANSB specifies the form of op( B ) to be used in
*           the matrix multiplication as follows:
*
*              TRANSB = 'N' or 'n',  op( B ) = B.
*
*              TRANSB = 'T' or 't',  op( B ) = B'.
*
*              TRANSB = 'C' or 'c',  op( B ) = B'.
*
*           Unchanged on exit.
*
*  M      - INTEGER.
*           On entry,  M  specifies  the number  of rows  of the  matrix
```

```
*           op( A )  and of the  matrix  C.  M  must  be at least  zero.
*           Unchanged on exit.
*
*  N      - INTEGER.
*           On entry,  N  specifies the number  of columns of the matrix
*           op( B ) and the number of columns of the matrix C. N must be
*           at least zero.
*           Unchanged on exit.
*
*  K      - INTEGER.
*           On entry,  K  specifies  the number of columns of the matrix
*           op( A ) and the number of rows of the matrix op( B ). K must
*           be at least  zero.
*           Unchanged on exit.
*
*  ALPHA  - DOUBLE PRECISION.
*           On entry, ALPHA specifies the scalar alpha.
*           Unchanged on exit.
*
*  A      - DOUBLE PRECISION array of DIMENSION ( LDA, ka ), where ka is
*           k  when  TRANSA = 'N' or 'n',  and is  m  otherwise.
*           Before entry with  TRANSA = 'N' or 'n',  the leading  m by k
*           part of the array  A  must contain the matrix  A,  otherwise
*           the leading  k by m  part of the array  A  must contain  the
*           matrix A.
*           Unchanged on exit.
*
*  LDA    - INTEGER.
*           On entry, LDA specifies the first dimension of A as declared
*           in the calling (sub) program. When  TRANSA = 'N' or 'n' then
*           LDA must be at least  max( 1, m ), otherwise  LDA must be at
```

```
*           least  max( 1, k ).
*           Unchanged on exit.
*
*  B      - DOUBLE PRECISION array of DIMENSION ( LDB, kb ), where kb is
*           n  when  TRANSB = 'N' or 'n',  and is  k  otherwise.
*           Before entry with  TRANSB = 'N' or 'n',  the leading  k by n
*           part of the array  B  must contain the matrix  B,  otherwise
*           the leading  n by k  part of the array  B  must contain  the
*           matrix B.
*           Unchanged on exit.
*
*  LDB    - INTEGER.
*           On entry, LDB specifies the first dimension of B as declared
*           in the calling (sub) program. When  TRANSB = 'N' or 'n' then
*           LDB must be at least  max( 1, k ), otherwise  LDB must be at
*           least  max( 1, n ).
*           Unchanged on exit.
*
*  BETA   - DOUBLE PRECISION.
*           On entry,  BETA  specifies the scalar  beta.  When  BETA  is
*           supplied as zero then C need not be set on input.
*           Unchanged on exit.
*
*  C      - DOUBLE PRECISION array of DIMENSION ( LDC, n ).
*           Before entry, the leading  m by n  part of the array  C must
*           contain the matrix  C,  except when  beta  is zero, in which
*           case C need not be set on entry.
*           On exit, the array  C  is overwritten by the  m by n  matrix
*           ( alpha*op( A )*op( B ) + beta*C ).
*
*  LDC    - INTEGER.
```

```
*           On entry, LDC specifies the first dimension of C as declared
*           in the calling (sub) program.  LDC must be at least
*           max( 1, m ).
*           Unchanged on exit.
*
*
*  Level 3 Blas routine.
*
*  -- Written on 8-February-1989.
*     Jack Dongarra, Argonne National Laboratory.
*     Iain Duff, AERE Harwell.
*     Jeremy Du Croz, Numerical Algorithms Group Ltd.
*     Sven Hammarling, Numerical Algorithms Group Ltd.
*
*
*     .. External Functions ..
      LOGICAL            LSAME
      EXTERNAL           LSAME
*     .. External Subroutines ..
      EXTERNAL           XERBLA
*     .. Intrinsic Functions ..
      INTRINSIC          MAX
*     .. Local Scalars ..
      LOGICAL            NOTA, NOTB
      INTEGER            I, INFO, J, L, NCOLA, NROWA, NROWB
      DOUBLE PRECISION   TEMP
*     .. Parameters ..
      DOUBLE PRECISION   ONE         , ZERO
      PARAMETER        ( ONE = 1.0D+0, ZERO = 0.0D+0 )
*     ..
*     .. Executable Statements ..
```

```
*
*     Set  NOTA  and  NOTB  as  true if  A  and  B  respectively are not
*     transposed and set  NROWA, NCOLA and  NROWB  as the number of rows
*     and  columns of  A  and the  number of  rows  of  B  respectively.
*
      NOTA  = LSAME( TRANSA, 'N' )
      NOTB  = LSAME( TRANSB, 'N' )
      IF( NOTA )THEN
         NROWA = M
         NCOLA = K
      ELSE
         NROWA = K
         NCOLA = M
      END IF
      IF( NOTB )THEN
         NROWB = K
      ELSE
         NROWB = N
      END IF
*
*     Test the input parameters.
*
      INFO = 0
      IF(      ( .NOT.NOTA                 ).AND.
     $         ( .NOT.LSAME( TRANSA, 'C' ) ).AND.
     $         ( .NOT.LSAME( TRANSA, 'T' ) )      )THEN
         INFO = 1
      ELSE IF( ( .NOT.NOTB                 ).AND.
     $         ( .NOT.LSAME( TRANSB, 'C' ) ).AND.
     $         ( .NOT.LSAME( TRANSB, 'T' ) )      )THEN
         INFO = 2
```

```
      ELSE IF( M  .LT.0               )THEN
         INFO = 3
      ELSE IF( N  .LT.0               )THEN
         INFO = 4
      ELSE IF( K  .LT.0               )THEN
         INFO = 5
      ELSE IF( LDA.LT.MAX( 1, NROWA ) )THEN
         INFO = 8
      ELSE IF( LDB.LT.MAX( 1, NROWB ) )THEN
         INFO = 10
      ELSE IF( LDC.LT.MAX( 1, M     ) )THEN
         INFO = 13
      END IF
      IF( INFO.NE.0 )THEN
         CALL XERBLA( 'DGEMM ', INFO )
         RETURN
      END IF
*
*     Quick return if possible.
*
      IF( ( M.EQ.0 ).OR.( N.EQ.0 ).OR.
     $    ( ( ( ALPHA.EQ.ZERO ).OR.( K.EQ.0 ) ).AND.( BETA.EQ.ONE ) ) )
     $   RETURN
*
*     And if  alpha.eq.zero.
*
      IF( ALPHA.EQ.ZERO )THEN
         IF( BETA.EQ.ZERO )THEN
            DO 20, J = 1, N
               DO 10, I = 1, M
                  C( I, J ) = ZERO
```

```
   10          CONTINUE
   20       CONTINUE
         ELSE
            DO 40, J = 1, N
               DO 30, I = 1, M
                  C( I, J ) = BETA*C( I, J )
   30          CONTINUE
   40       CONTINUE
         END IF
         RETURN
      END IF
*
*     Start the operations.
*
      IF( NOTB )THEN
         IF( NOTA )THEN
*
*           Form  C := alpha*A*B + beta*C.
*
            DO 90, J = 1, N
               IF( BETA.EQ.ZERO )THEN
                  DO 50, I = 1, M
                     C( I, J ) = ZERO
   50             CONTINUE
               ELSE IF( BETA.NE.ONE )THEN
                  DO 60, I = 1, M
                     C( I, J ) = BETA*C( I, J )
   60             CONTINUE
               END IF
               DO 80, L = 1, K
                  IF( B( L, J ).NE.ZERO )THEN
```

```
                     TEMP = ALPHA*B( L, J )
                     DO 70, I = 1, M
                        C( I, J ) = C( I, J ) + TEMP*A( I, L )
   70                CONTINUE
                  END IF
   80          CONTINUE
   90       CONTINUE
         ELSE
*
*           Form  C := alpha*A'*B + beta*C
*
            DO 120, J = 1, N
               DO 110, I = 1, M
                  TEMP = ZERO
                  DO 100, L = 1, K
                     TEMP = TEMP + A( L, I )*B( L, J )
  100             CONTINUE
                  IF( BETA.EQ.ZERO )THEN
                     C( I, J ) = ALPHA*TEMP
                  ELSE
                     C( I, J ) = ALPHA*TEMP + BETA*C( I, J )
                  END IF
  110          CONTINUE
  120       CONTINUE
         END IF
      ELSE
         IF( NOTA )THEN
*
*           Form  C := alpha*A*B' + beta*C
*
            DO 170, J = 1, N
```

```
               IF( BETA.EQ.ZERO )THEN
                  DO 130, I = 1, M
                     C( I, J ) = ZERO
  130             CONTINUE
               ELSE IF( BETA.NE.ONE )THEN
                  DO 140, I = 1, M
                     C( I, J ) = BETA*C( I, J )
  140             CONTINUE
               END IF
               DO 160, L = 1, K
                  IF( B( J, L ).NE.ZERO )THEN
                     TEMP = ALPHA*B( J, L )
                     DO 150, I = 1, M
                        C( I, J ) = C( I, J ) + TEMP*A( I, L )
  150                CONTINUE
                  END IF
  160          CONTINUE
  170       CONTINUE
         ELSE
*
*           Form  C := alpha*A'*B' + beta*C
*
            DO 200, J = 1, N
               DO 190, I = 1, M
                  TEMP = ZERO
                  DO 180, L = 1, K
                     TEMP = TEMP + A( L, I )*B( J, L )
  180             CONTINUE
                  IF( BETA.EQ.ZERO )THEN
                     C( I, J ) = ALPHA*TEMP
                  ELSE
```

```
                     C( I, J ) = ALPHA*TEMP + BETA*C( I, J )
                  END IF
  190          CONTINUE
  200       CONTINUE
         END IF
      END IF
*
      RETURN
*
*     End of DGEMM .
*
      END
```

Appendix B

In Table B.1, we present properties of the matrices given to the DGEMM routine broken down by problem. For each problem, we give the average number of rows in A and C (avg. m), the average number of columns in B and C (avg. n), and the average number of columns in A/rows in B (avg. k). In addition, we calculated the average number of operations per multiplication (derived by summing the product $m \times n \times k$ for each multplication, and dividing by the number of multiplications). We also supply the number of DGEMM calls, and the total number of operations performed by the DGEMM routine over the course of the solution process. We see that anywhere from 13 178 to 309 516 calls are made to DGEMM throughout the course of a solution process. Anywhere from 15 million to over 100 billion operations are executed in the DGEMM routine for our problem set.

Problem	**Avg. m**	**Avg. n**	**Avg. k**	**Avg. Ops per mul**	**Number of Calls**	**Total Ops**
cfd2	231	17	18	473 417	105 082	49 747 605 194
ct20stif	73	12	10	87 151	43 777	3 815 209 327
finan512	108	9	6	83 686	35 558	2 975 706 788
nd3k	831	29	28	1 235 345	13 178	16 279 376 410
nd6k	1406	31	31	2 522 465	39 772	100 323 477 980
nemeth26	42	1	2	122	123 596	15 078 712
pwtk	116	19	16	180 984	111 583	20 194 737 672
600	28	8	7	20 068	76 024	1 525 649 632
700	30	8	7	23 114	104 622	2 418 232 908
800	32	8	7	25 485	136 752	3 485 124 720
900	36	9	8	35 431	173 748	6 156 065 388
1000	36	9	8	36 900	214 411	7 911 765 900
1100	37	9	8	39 406	260 320	10 258 169 920
1200	38	9	8	41 583	309 516	12 870 603 828

Table B.1: Test problem DGEMM matrix properties

Bibliography

[1] A. George and J.W. H. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice Hall, 1981.

[2] Nathaniel John Nystrom, *Bytecode-Level Analysis and Optimizations of Java Classes*, Masters Thesis, 1998, Purdue University.

[3] James Gosling, *The Evolution of Numerical Computing in Java*, http://java.sun.com/people/jag/FP.html, last accessed September 22, 2003.

[4] S. Liang, *The Java Native Interface: Programmer's Guide and Specification*, Addison Wesley Longman, 1999.

[5] A. George, *User Guide for Sparspak90*, Department of Computer Science, Univeristy of Waterloo, 2001.

[6] Y. Saad, H. A. G. Wijshoff, *A Benchmark Package for Sparse Matrix Computations*, Proceedings of the 2nd international conference on Supercomputing, Pages 500-509, 1988.

[7] J.J. Dongarra, *Performance of various computers using standard linear equations software in a FORTRAN environment*, Argonne National Lab. report, MCS-TM 23, 1986.

[8] Y. Saad, *Iterative Methods for Sparse Linear Systems Second Edition*, SIAM, 2003.

[9] M. Hirzel, A. Diwan, M. Hertz, *Connectivity-based garbage collection*, Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Pages 359-373, 2003.

[10] Doolin, D.A., Dongarra, J., Seymour, K. JLAPACK - Compiling LAPACK Fortran to Java. Sci. Prog. 7 (1999), 111-138.

[11] A. George, *On Finding and Analyzing the Structure of the Cholesky Factor*, Proceedings of the NATO Conference on Algorithms for Large Scale Linear Algebraic Systems: State of the Art and Applications in Science and Engineering, University of Gran Canaria, Spain, June 23-July 5 (1996).

[12] J. M. Bull, L. A. Smith, L. Pottage, R. Freeman, *Benchmarking Java against C and Fortran for Scientific applications*, Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande, Pages 97-105, 2001.

[13] B. L. Blount and S. Chatterjee, *An Evaluation of Java for Numerical Computing*, Scientific Programming 7 (2), pages 97-119, 1999.

[14] R. F. Boisvert, J. E. Moreira, M. Philippsen, and R. Pozo, *Java and numerical computing*, IEEE Computing in Science and Engineering, 3(2):18–24, 2001.

[15] J.E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, D. Lawrence, *Java programming for high-performance numerical computing*. IBM System Journal, vol. 39, no. 1, 21–56, 2000.

[16] S.P. Midkiff, J. E. Moreira, and M. Snir, *Optimizing Array Reference Checking in Java Programs*, IBM Journal of Research and Development 41, No. 3, 233-262 (May 1997).

[17] G. E. Forsyth, C. B. Moler, *Computer Solution of Linear Algebraic Systems*, Prentice-Hall Inc., Englewood Cliffs, N.J., 1967.

[18] J. Gosling, B. Joy, G. Steele, G. Bracha, *The Java Language Specification*, http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html, last accessed April 21, 2004.

[19] J.E. Moreira, S.P. Midkiff, *A Comparison of Java, C/C++, and Fortran for Numerical Computing*, IEEE Antennas and Propagation Magazine, 40(5), pages 102-104, Oct. 1998

[20] T. L. Freeman, J. R. Gurd, M. Lujan, J. Migue, *Elimination of Java array bounds checks in the presence of indirection*, Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande, pages 76–85, 2002.

[21] R. F. Boisvert, J. J. Dongarra, R. Pozo, *Developing Numerical Libraries in Java*, ACM 1998 Workshop on Java for High-Performance Network Computing, 1998.

[22] A. Berger, J. Mulvey, E. Rothberg, R. Vanderbei, *Solving multistage stochastic problems using tree dissection*, tech. report SOR-97-07, Program in Statistics and Operations Research.

[23] T. Dickens, *Migrating Legacy Engineering Applications to Java*, OOPSLA 2002 Practioners Reports, ACM Press, 2002.

[24] Sun Microsystems, *http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/Java_HSpot_WP_v1.4.1_1002_1.html*, last accessed August 16th, 2004.