A Note on Native Level 1 BLAS in Java*

Aart J.C. Bik and Dennis B. Gannon
Lindley Hall 215, Computer Science Department, Indiana University
Bloomington, Indiana 47405-4101, USA
ajcbik@cs.indiana.edu

Abstract

In this research note, we explore the potential of extending the Java Application Programming Interface with some mathematical primitives to improve the performance of certain operations in Java programs while maintaining portability. In particular, we show that providing straightforward native implementations of primitives from Level 1 BLAS can already improve the performance substantially. On multi-processors, combining this native Level 1 BLAS with the multi-threading mechanism of Java may even provide a simple and portable way to obtain a Java program that runs faster than compiled serial C code.

1 Introduction

Portability of the Java programming language [13] is obtained by compiling Java programs into architectural neutral instructions (bytecode) of the Java Virtual Machine (JVM) [16], rather than into native machine code. Bytecode runs on any platform that supports an implementation of the JVM. Although the interpretation of bytecode is substantially faster than the interpretation of most high level languages, still a performance penalty must be paid for this portability. Clearly, for Java applications that are computational intensive, it would be desirable to reduce this performance penalty without sacrificing the portability of the language. One approach, for example, is to optimize the Java bytecode [3, 5], either at compile-time (where a machine independent bytecode to bytecode optimization is added as additional phase to the Java compiler), or at run-time (where optimizations that require knowledge of the target machine are applied before execution). Some implementations of the JVM further improve performance by means of 'just-in-time compilation' (JITC), where, at run-time, bytecode is compiled into native machine code.

In this note, we explore a way to speedup certain operations in Java programs using ideas from the mathematical software community. Here, it has been widely accepted that adopting a set of basic routines for problems in linear algebra can help in improving the clarity, portability, modularity, maintenance, robustness, and even the efficiency of mathematical software. The most well-known example of such a set of routines is formed by the Basic Linear Algebra Subprograms [10, ch5]. The original set of vector-vector operations is now commonly referred to as Level 1 BLAS [14, 15]. The set has been extended to Level 2 BLAS [8, 9] and Level 3 BLAS [6, 7] to provide more opportunities to exploit vector processing facilities for matrix-vector operations and memory hierarchies or parallelism for matrix-matrix operations, respectively. Once an efficient implementation of BLAS is available, new mathematical software can be easily build on top of the primitives.

Obviously, a similar approach can be taken for Java by extending the Java API (Application Programming Interface) with an appropriate set of mathematical primitives. In first instance, a Java implementation can be provided for all these primitives to preserve the portability of all Java programs in which the mathematical primitives are used.

*This project is supported by DARPA under contract ARPA F19628-94-C-0057 through a subcontract from Syracuse University.

On a particular machine, however, the performance of all Java software that uses these primitives is simply improved by providing native implementations of the mathematical primitives. Although providing a broad range of highly optimized mathematical primitives would offer the best potential to exploit all characteristic of a particular target machine, this approach would also require the most programming efforts to port the mathematical primitives in the API to different machines. Therefore, in this research note, we explore the potential of extending the API with straightforward native implementations of Level 1 BLAS only. We will see that this extension alone already can improve performance substantially, while combining these native Level 1 BLAS with multi-threading in Java may even provide a simple and portable way to outperform compiled serial C code on multi-processors.

In section 2, we briefly discuss how native methods are integrated in Java. In section 3, we present the results of a series of experiments, followed by conclusions in section 4.

2 Native BLAS

In Java, if the keyword `native` appears as part of a method definition (without an implementation), then this implies that the method is implemented in another language. A set of primitives from BLAS, for instance, can be defined using the following class that provides a shared library loader that will load the actual implementation when required as well as definitions of all primitives in the set:

```
class Blas {

  // Shared Library Loader

  static {
    System.loadLibrary("blas");
  }

  // Level 1 BLAS

  native static double ddot(int n, double[] x, int xoff, int incx,
                                                 double[] y, int yoff, int incy);
  native static void daxpy(int n, double alpha, double[] x, int xoff, int incx,
                                                 double[] y, int yoff, int incy);

  ...
}
```

Here, method definitions of DDOT ($d \leftarrow \vec{x}^T\vec{y}$ in double precision) and DAXPY ($\vec{y} \leftarrow \alpha\vec{x}+\vec{y}$ in double precision) are shown.[1] Alternatively, as stated earlier, Java implementations of the primitives could be given in first instance.

In figure 1, the integration of native methods in Java is illustrated. Using JDK1.0.2, first the source file `Blas.java` is compiled into `Blas.class` using the Java compiler `javac`. Subsequently, the tool `javah` is used to generate a stubs file `Blas.c` and a header file `Blas.h`. The latter file contains C prototypes for all native methods in the class. The actual implementation of these methods is given in a file `BlasImp.c`. Method `ddot()`, for example, can be implemented in a straightforward manner in C as follows, where the macro `unhand()` is used to dereference an object handle:

[1] Because Java does not support passing of arbitrary sub-arrays, offsets into arrays are added as additional parameters (cf. the Java implementation of BLAS in [11]).

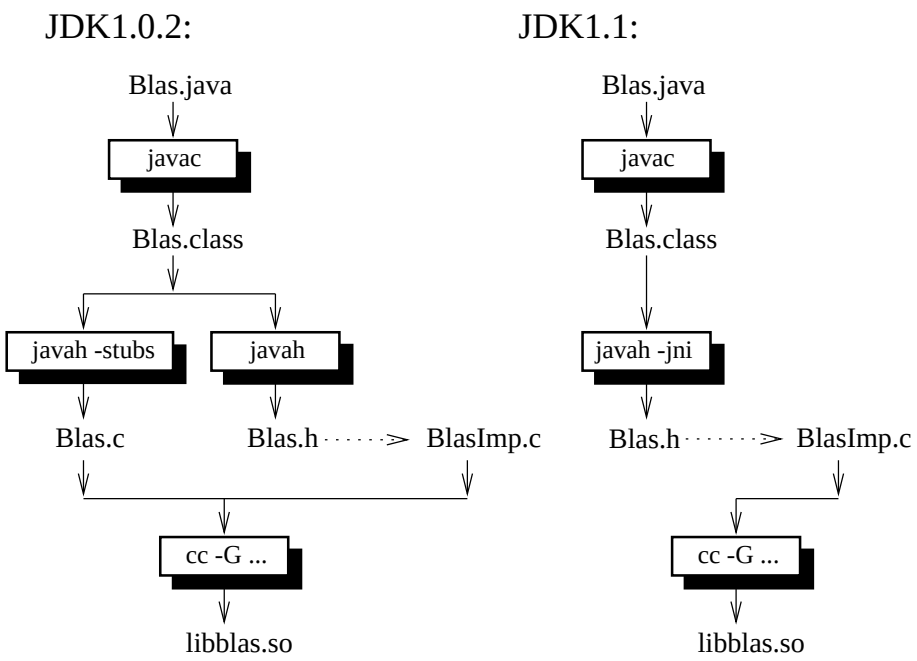

Figure 1: Integration of Native Methods

```
#include <StubPreamble.h>
#include "Blas.h"

double Blas_ddot(struct HBlas *this, long n,
                 HArrayOfDouble *x_, long xoff, long incx,
                 HArrayOfDouble *y_, long yoff, long incy) {
  double *x = unhand(x_)->body;
  double *y = unhand(y_)->body;
  double  d = 0;

  if (n > 0) {
    if ((incx == 1) && (incy == 1)) { /* Unit strides */
      int i;
      for (i=0;i < n; i++)
         d += x[xoff+i]*y[yoff+i];
    }
    else {                        /* Non-Unit strides */
     ...
    }
  }
  return d;
}
```

Eventually, the files `Blas.c` and `BlasImp.c` are compiled into a shared library. The way in which this is done and the naming convention of shared libraries depends on the target architecture (under Solaris, for instance, the shared library is created as:

```
cc -G -O  -I $JAVA_HOME/include -I $JAVA_HOME/include/solaris Blas.c BlasImp.c -o libblas.so
```

As illustrated in the second picture in figure 1, integrating native methods in JDK1.1 is slightly different. Only a header file `Blas.h` which contains the appropriate prototypes is generated using `javah -jni`. The actual implementation of methods is, again, given in a file `BlasImp.c` Method `ddot()`, for example, can be implemented in C as follows, where functions `GetDoubleArrayElements()` and `ReleaseDoubleArrayElements()` are used to obtain and release a double array that cannot be moved by the garbage collector. Eventually, the whole file is compiled into a shared library:

```
#include "Blas.h"
JNIEXPORT jdouble JNICALL Java_Blas_ddot
    (JNIEnv *env, jclass class, jint n, jdoubleArray x_, jint xoff, jint incx,
                                        jdoubleArray y_, jint yoff, jint incy) {
  jdouble *x = (*env)->GetDoubleArrayElements(env, x_, 0);
  jdouble *y = (*env)->GetDoubleArrayElements(env, y_, 0);
  jdouble  d = 0;

  if (n > 0) {
    if ((incx == 1) && (incy == 1)) { /* Unit strides */
      int i;
      for (i=0;i < n; i++)
        d += x[xoff+i]*y[yoff+i];
    }
    else {                          /* Non-Unit strides */
      ...
    }
  }
  (*env)->ReleaseDoubleArrayElements(env, x_, x, 0);
  (*env)->ReleaseDoubleArrayElements(env, y_, y, 0);
  return d;
}
```

If native methods are used, care must be taken with respect to the precision of the computed result, because *Java data types are mapped to the nearest matching data types on the native language side.* More details on integrating native methods in Java can be found on the Web [17].

3 Experiments

In this section, we present the results of a series of experiments that have been conducted on an IBM RS/6000 G30 with four PowerPC 604 processors using the AIX4.2 JDK1.0.2B (with JITC) and the AIX4.2 JDK1.1beta (without JITC), a Sun with two 175MHz. ultra SPARC processors using the Solaris 2.5 JDK1.0.2dp (with JITC), and an SGI Indy 4600 with one 133 MHZ IP22 Processor using the IRIX6.2 JDK1.0.2 (without JITC). All Java and C programs are compiled using the flag '`-O`'. Bytecode is interpreted using the flag '-noasyncgc' and, if available, with 'just-in-time compilation' (JITC) enabled.

3.1 Some Mathematical Operations

In the first series of experiments, we compare the performance a pure Java implementation, a Java implementation that uses native Level 1 BLAS (and possibly parallelism by means of Java multi-threading), and a compiled C implementation of an $\Theta(N)$ DAXPY-operation, an $\Theta(N^2)$ matrix times vector operation, and an $\Theta(N^3)$ matrix times matrix operation.

Below we show the pure Java implementation of the DAXPY operation ($\vec{y} \leftarrow \alpha\vec{x} + \vec{y}$), and the Java implementation that uses a primitive from Level 1 BLAS:

pure Java:

```
for (int i = 0; i < N; i++)
  y[i] += alpha * x[i];
```

Java + Level 1 BLAS:

```
Blas.daxpy(N, alpha, x, 0, 1, y, 0, 1);
```

The inner product Java implementation of matrix times vector and a Java implementation that uses a call to Level 1 BLAS DDOT ($d \leftarrow \vec{x}^T\vec{y}$) are shown below:

pure Java:

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    b[i] += a[i][j] * x[j];
```

Java + Level 1 BLAS:

```
for (int i = 0; i < N; i++)
  b[i] = Blas.ddot(N, a[i], 0, 1, x, 0, 1);
```

Finally, the product of two matrices can be computed using either the pure Java fragment shown below, or a similar Java fragment that uses a call to DAXPY:

pure Java:

```
for (int i = 0; i < N; i++)
  for (int k = 0; k < N; k++)
    for (int j = 0; j < N; j++)
      c[i][j] += a[i][k] * b[k][j];
```

Java + Level 1 BLAS:

```
for (int i = 0; i < N; i++)
  for (int k = 0; k < N; k++)
    Blas.daxpy(N, a[i][k], b[k], 0, 1, c[i], 0, 1);
```

The performance of the fragments for matrix times vector and matrix times matrix with Level 1 BLAS may be further improved by parallelization of the outermost `i`-loop. In earlier work [1, 2], we have shown how loop parallelization can be expressed in Java by means of multi-threading. In this manner the transformed Java program remains portable (i.e. the parallelized version still runs on uni-processors with only a slight overhead), while the programming efforts of the parallelization are reduced substantially with respect to exploiting parallelism in a native language. The versions with a parallel outermost loop are labeled as 'parallel' in the subsequent figures. The C versions of the three mathematical operations are similar to the pure Java implementations.

In figures 2–4 and figures 5–7 we show the execution times for varying values of N on the IBM using the AIX4.2 JDK1.0.2B (with JITC) and the AIX4.2 JDK1.1beta (without JITC), respectively. Here we see that with JITC, providing Level 1 BLAS primitives only suffices to obtain performance that is close to the performance of native C code. Moreover, because IBM's implementation of the JVM supports the actual parallel execution of threads, the parallel versions with native Level 1 BLAS even outperform serial C code. Without JITC, however, the $\Theta(N^3)$ operation still suffers from much overhead, and here it would probably be desirable to provide primitives from Level 2 BLAS as well.

In figures 8–10 the execution times on the Sun using the Solaris 2.5 JDK1.0.2dp (with JITC) are shown. Now, in all cases the performance is even slightly better than the performance of straightforward native C code. Obviously, Sun's implementation of the JVM does not support the actual parallel execution of threads.

In figures 11–13 the execution times on the SGI using the IRIX6.2 JDK1.0.2 (without JITC) are shown. Again, without JITC, the performance of $\Theta(N^3)$ matrix times matrix operations is substantially less than the performance of compiled C code. Obviously, on this uni-processor, no speedup can be expected from loop parallelization.

3.2 Linpack Benchmark

In this section, we present some performance numbers for a Java Linpack benchmark [11] that solves a system of linear equations (i.e. factorization followed by forward and back substitution).

n	JDK1.0.2
500	`Mflops:  3.3 Time:  25.8 s. Norm Res:  5.28`
1000	`Mflops:  3.2 Time: 210.9 s. Norm Res:  9.61`
	JDK1.0.2 + native Level 1 BLAS
500	`Mflops:  8.1 Time:  10.3 s. Norm Res:  4.50`
1000	`Mflops:  7.6 Time:  88.5 s. Norm Res: 11.13`
	JDK1.0.2 + native Level 1 BLAS (parallel factorize)
500	`Mflops:  9.7 Time:   8.6 s. Norm Res:  4.50`
1000	`Mflops: 15.6 Time:  43.0 s. Norm Res: 11.13`

Table 1: Linpack benchmark on the IBM

In tables 1–3, we show the results on the IBM, Sun, and SGI, respectively, for a pure Java implementation of this benchmark and an implementation that uses native implementation of the primitives DDOT, DAXPY, DSCAL and IDAMAX from Level 1 BLAS. For the IBM, we also present the performance of a version in which loop parallelization has been applied to the elimination step in the factorization.

Again, it is clear that providing native Level 1 BLAS can improve the performance substantially. Note, however, that the mapping between Java and C data types caused a slight change in precision of the computed result.

n	JDK1.0.2
500	`Mflops:  8.5 Time:   9.9 s. Norm Res:  5.17`
1000	`Mflops:  8.5 Time:  78.6 s. Norm Res: 10.10`
	JDK1.0.2 + native Level 1 BLAS
500	`Mflops: 18.4 Time:   4.6 s. Norm Res:  5.77`
1000	`Mflops: 18.6 Time:  35.9 s. Norm Res: 10.44`

Table 2: Linpack benchmark on the Sun

n	JDK1.0.2
500	`Mflops:  0.3 Time:  265.4 s. Norm Res:  5.17`
1000	`Mflops:  0.3 Time: 2119.5 s. Norm Res: 10.10`
	JDK1.0.2 + native Level 1 BLAS
500	`Mflops:  7.3 Time:   11.6 s. Norm Res:  5.77`
1000	`Mflops:  7.0 Time:   95.9 s. Norm Res: 10.44`

Table 3: Linpack benchmark on the SGI

4 Conclusions

In this research note, we have explored the potential of extending the Java Application Programming Interface (API) with a native implementation of Level 1 BLAS to improve the performance of certain mathematical operations in Java programs. Because, in first instance, a Java implementation of the primitives can be provided, the portability of Java is maintained. We have shown that for implementations of the Java Virtual Machine (JVM) that support 'just-in-time compilation' (JITC), providing straightforward native Level 1 BLAS already suffices to obtain performance that is close to the performance of native C code for $\Theta(N)$, $\Theta(N^2)$, and $\Theta(N^3)$ mathematical operations. In case JITC is not supported, the performance difference of the latter operations is more profound, and here it would be desirable to provide higher level primitives as well. In addition, we have shown that, on multi-processors, combining native Level 1 BLAS with the multi-threading mechanism of Java may provide a simple and portable way to obtain a Java program that runs faster than compiled serial C code.

In conclusion, although it is obvious that more speedup can be expected by providing highly optimized native implementations of a broad range of mathematical primitives, in which characteristics of the target machine (such as multiple processors, memory hierarchies or vector processing facilities) are fully exploited at native level, providing straightforward native implementations of Level 1 BLAS alone already can help in improving the performance of mathematical operations in Java. In all cases where native methods are used, however, care must be taken with respect to the precision of the computed result, because Java data types are mapped to the nearest matching data types on the native language side.

References

[1] Aart J.C. Bik and Dennis B. Gannon. Automatically exploiting implicit parallelism in Java. *To Appear in a Special Issue of Concurrency, Practice and Experience*, 1997.

[2] Aart J.C. Bik, Juan E. Villacis, and Dennis B. Gannon. `javar` *manual*. Computer Science Department, Indiana University, 1997. This manual and the complete source of `javar` is made available at `http://www.extreme.indiana.edu/hpjava/`.

[3] Zoran Budimlic and Ken Kennedy. Optimizing Java – theory and practice. *To Appear in a Special Issue of Concurrency, Practice and Experience*, 1997.

[4] Bryan Carpenter, Yuh-Jye Chang, Geoffrey C. Fox, Donald Leskiw, and Xiaoming Li. Experiments with HP Java. *To Appear in a Special Issue of Concurrency, Practice and Experience*, 1997.

[5] Michal Cierniak and Wei Li. Optimizing Java bytecodes. *To Appear in a Special Issue of Concurrency, Practice and Experience*, 1997.

[6] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16:1–17, 1990.

[7] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms: Model implementation and test programs. *ACM Transactions on Mathematical Software*, 16:18–28, 1990.

[8] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14:1–17, 1988.

[9] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms: Model implementation and test programs. *ACM Transactions on Mathematical Software*, 14:18–32, 1988.

[10] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. Society for Industrial and Applied Mathematics, 1991.

[11] Jack J. Dongarra et al. *Java Linpack Benchmark*. This benchmark is made available at `http://www.netlib.org/benchmark/linpackjava/`.

[12] Geoffrey C. Fox and Wojtek Furmanski. Java for parallel computing and as a general language for scientific and engineering simulation and modelling. *To Appear in a Special Issue of Concurrency, Practice and Experience*, 1997.

[13] James Gosling, Bill Joy, and Guy Steele. *Java Programming Language*. Addison-Wesley, Reading, Massachusetts, 1996.

[14] C.L. Lawson, R.J. Hanson, D.R. Kincaid, and F.T. Krogh. Algorithm 539: Basic linear algebra subprograms for FORTRAN usage. *ACM Transactions on Mathematical Software*, 5:324–325, 1979.

[15] C.L. Lawson, R.J. Hanson, D.R. Kincaid, and F.T. Krogh. Basic linear algebra subprograms for FORTRAN usage. *ACM Transactions on Mathematical Software*, 5:308–323, 1979.

[16] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[17] Beth Stearns. *Integrating Native Code and Java Programs*. This documents is made available at `http://java.sun.com/nav/read/Tutorial/native1.1/`.

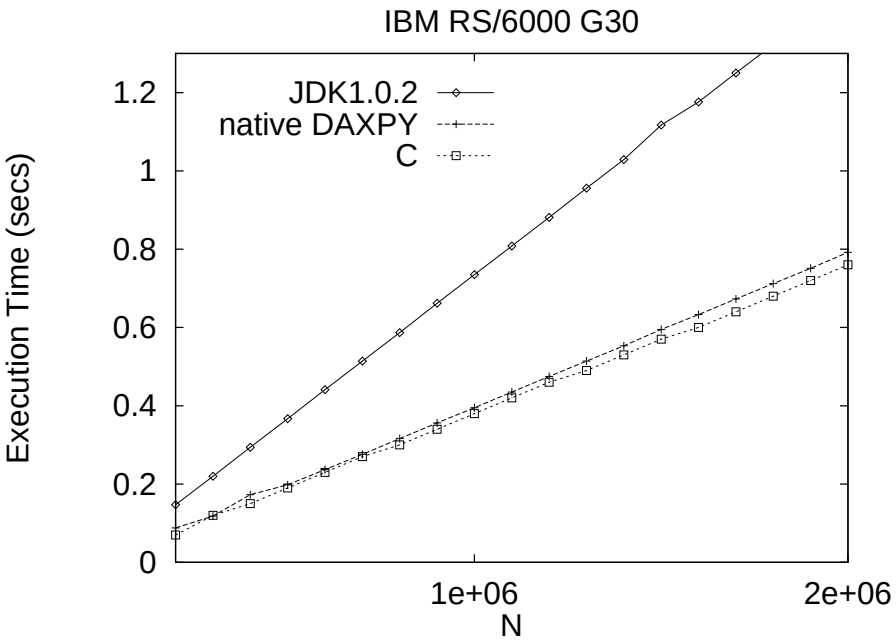

Figure 2: DAXPY on the IBM

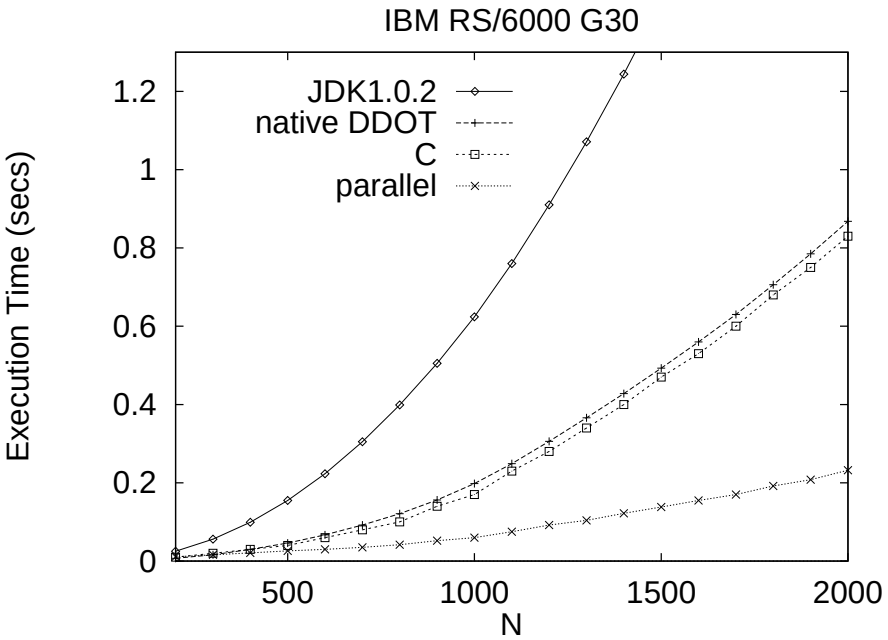

Figure 3: Matrix x Vector on the IBM

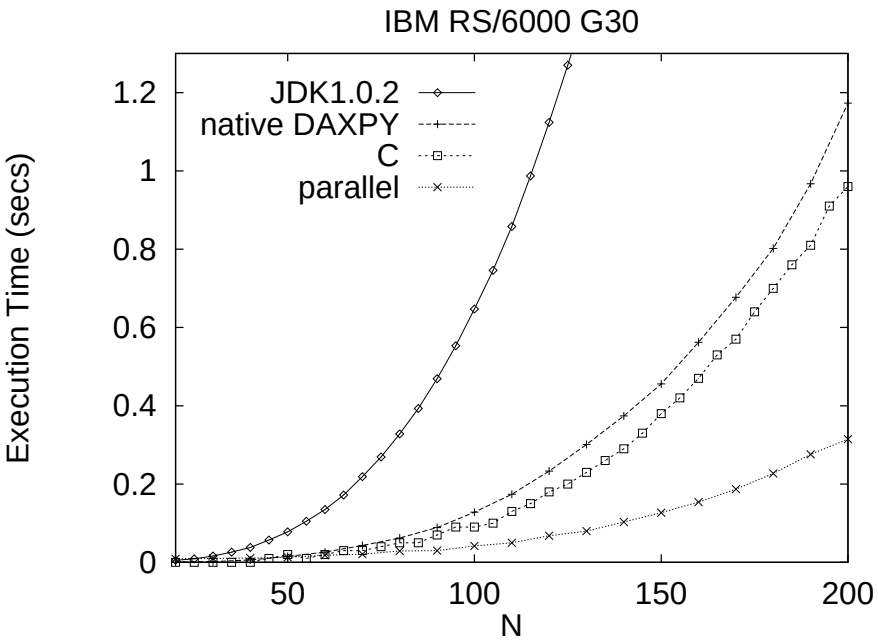

Figure 4: Matrix x Matrix on the IBM

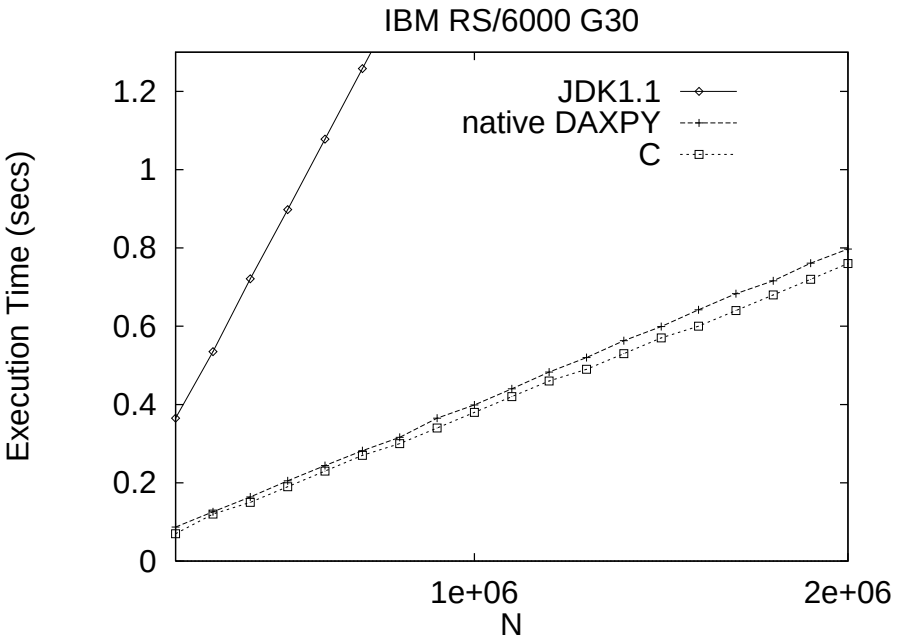

Figure 5: DAXPY on the IBM

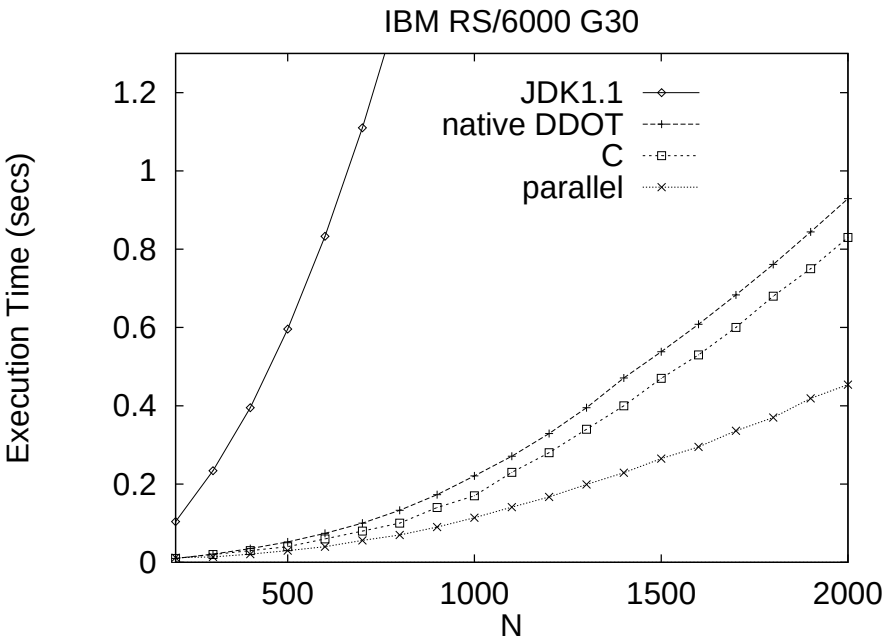

Figure 6: Matrix x Vector on the IBM

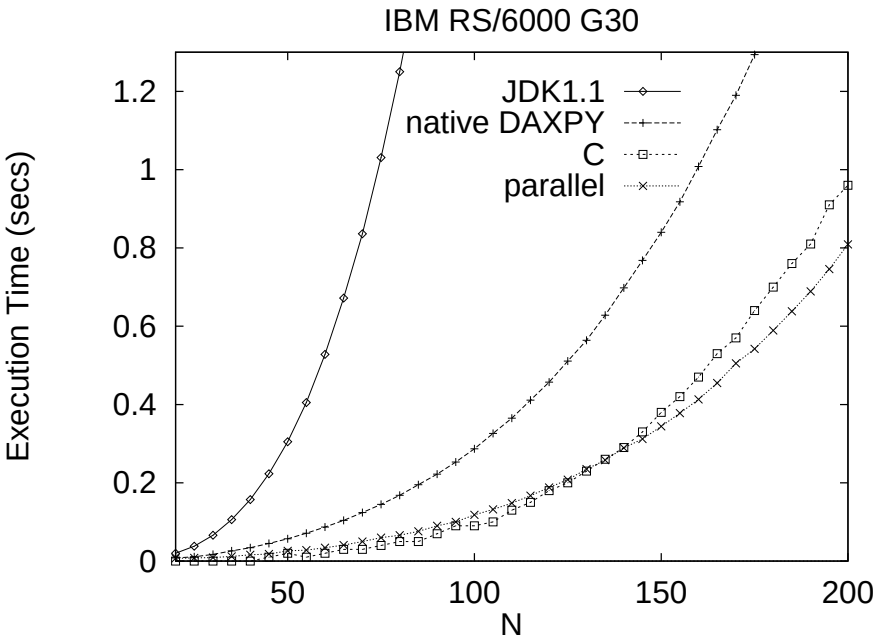

Figure 7: Matrix x Matrix on the IBM

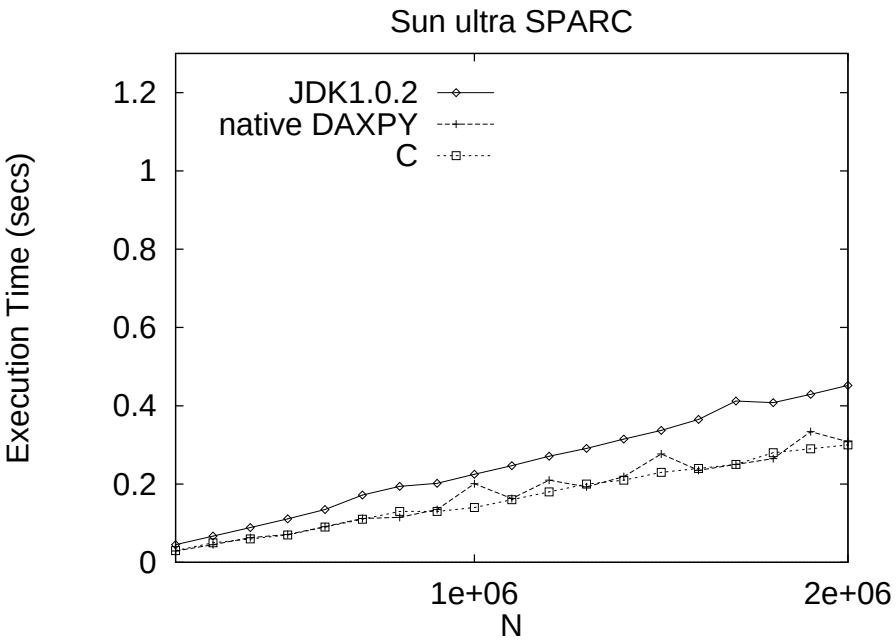

Figure 8: DAXPY on the Sun

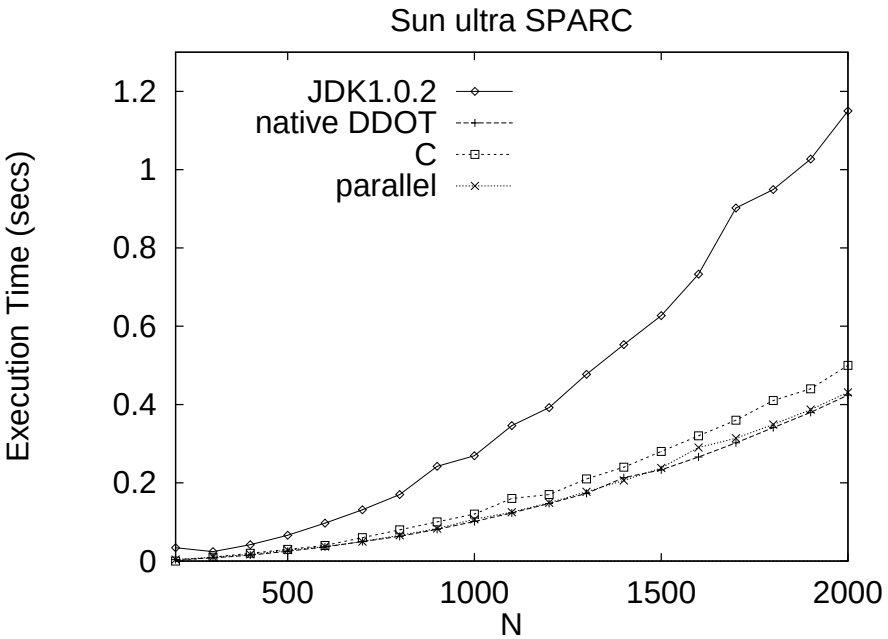

Figure 9: Matrix x Vector on the Sun

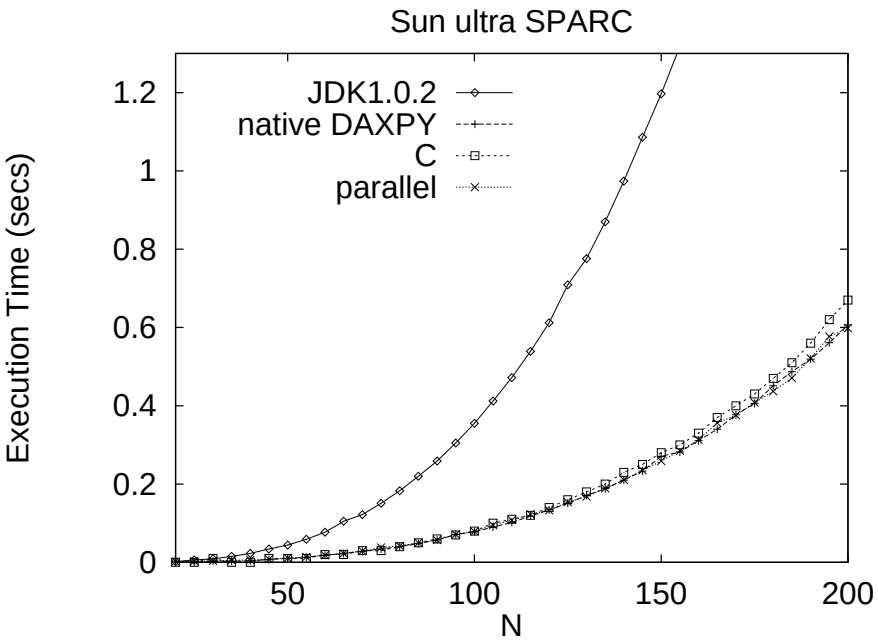

Figure 10: Matrix x Matrix on the Sun

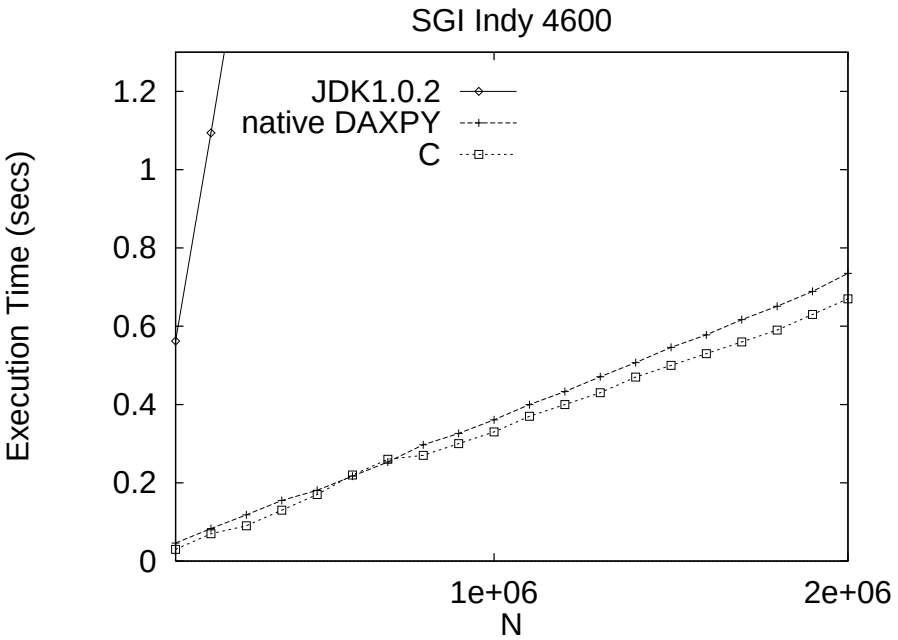

Figure 11: DAXPY on the SGI

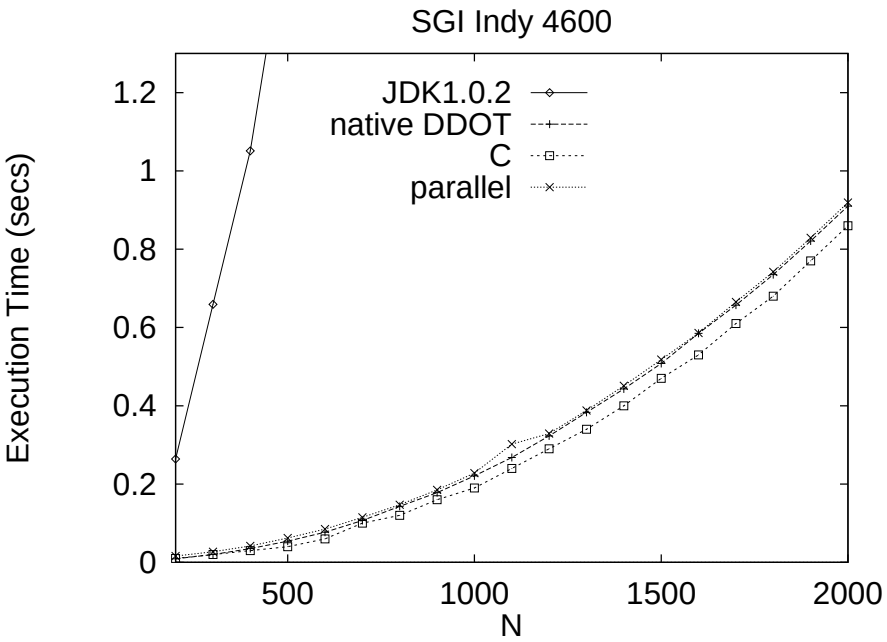

Figure 12: Matrix x Vector on the SGI

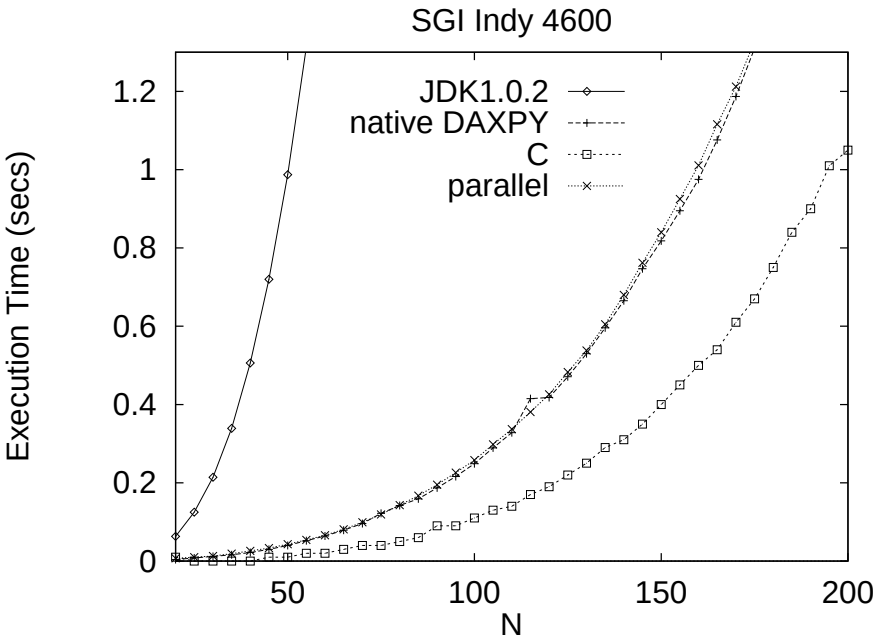

Figure 13: Matrix x Matrix on the SGI