Comparing the performance of JAVA with Fortran and C for numerical computing *

Y. F. Hu[1], R. J. Allan[1] and K. C. F. Maguire[2]
[1]Daresbury Laboratory, CLRC, Daresbury,
Warrington WA4 4AD, United Kingdom
[2]European Southern Observatory
Karl-Schwarzschild-Strasse 2
D-85748 Garching bei Muenchen
Germany

August 16, 2000

Abstract

The performance of Java has been compared with that of Fortran 90 and C on two benchmarks of particular interest to scientific and engineering applications. It was found that in comparison with F90 and C, the I/O and compute performance of Java varies from 30% slower to about 3 times slower, depending on the platforms and the compilers (and the Java Virtual Machines). The best relative performance is achieved on a Pentium II, where it was found that the IBM Java yields code that is about 30-40% slower in computing and I/O performance.

1 Introduction

Java is now widely recognized as a good object-oriented language for writing portable programs quickly. However, its penetration to computationally intensive numerical calculation is still low. One of the main reasons is its poor performance, or the perception of it, for such numerically intensive computing.

However, Java, as a programming language that is supposed to be "written once and run everywhere", is very attractive for scientific and engineering

*This report and the sparse matrix multiplication benchmark is also available at `http://www.dl.ac.uk/TCSC/Staff/Hu_Y_F/JASPA` calculations that involve many researchers using different platforms. With Java's strong connections to Internet technology, the potential of running applications over the Internet, either for server/client side computing, or in terms of using computing resources over the Internet as a Computational Grid, is enormous.

That Java can suffer from performance problems is perhaps not at all surprising. Java is not designed for numerical computing, rather, it is a truly object oriented language which aims to achieve bit-for-bit reproducibility of results on different platforms, safety of execution, and ease of programming and testing. As a result, everything apart from very primitive types is an object, with the associated overhead of handling objects. Furthermore, access to array elements is subject to expensive bound checking and null pointer checking. Array objects are also not assumed to occupy a contiguous section of the memory, which is bad for optimal cache usage. Besides, Java is not allowed to take advantage of some special features of hardware, such as the fused multiply-add instruction found on the IBM POWER architecture.

In addition to the above, compiler technology takes time to mature. Compared with Fortran and C, Java is still relatively new. Java also differs from C and Fortran in that Java codes are first compiled into bytecodes, and then interpreted on any platform using a Java Virtual Machine (JVM). This is similar to what other interpreted languages do, such as Basic and Perl. Java is therefore optimized at run time, rather than at compile time.

There are however already major advances in the Java compiler technology. One of these technologies is the JIT (Just-In-Time) compiler. When a JIT is present, the Java Virtual Machine hands the bytecodes to JIT, which in turn compiles them into native code for the platform, and runs the resulting executables. The JIT is an integral part of the Java Virtual Machine, and is transparent to the users. Since Java is a dynamic language, the JIT is really "just-in-time", and compiles methods on a method by method basis just before they are called. There has also been effort in reducing the amount of array bound check through some clever transformations of code [1].

Not long ago, Java could only achieve say 20% of the performance of Fortran on a good day. With the introduction of new compiler technologies such as JIT and HotSpot (`http://developer.java.sun.com/developer/technicalArticles/Networking/HotSpot/index.html`), there have been an increasing number of reports of Java compilers giving applications with comparable performance to statically compiled languages such as C (e.g., [2]).

It is therefore our intention in this report to have a close look at the

Java performance issue as it stands today, and compare it with Fortran 90 (F90) and C on benchmarks that are important for scientific and engineering applications. This is of course a task that is impossible to achieve fully given the varying kernels dominating different applications. These may range from dense matrix calculations, to sparse matrix operations, to the solution of eigenvalue problems, or even repeated evaluations of elementary or special functions. We therefore decided to restrict our comparison to two benchmarks.

For our first study, we compare the performance of JAVA with C and F90 for sparse matrix based calculations. Sparse matrices appear frequently in large-scale scientific and engineering applications and the ability of JAVA to handle such sparse systems efficiently is of vital importance to the usefulness of this language for these applications. Our sparse benchmark compares the three languages in terms of the speed for sparse matrix multiplications. I/O speed is also tested.

For our second study, we measure the performance of JAVA against C using the SciMark2 benchmark (`http://math.nist.gov/scimark2/`), which contains a variety of applications including FFT, dense LU factorization, and sparse matrix vector products.

It is worth noting that since Java 2 (version 1.2 onwards), there are two floating-point modes – `strictfp` and default. The `strictfp` mode, which applies to classes or methods with the `strictfp` keyword, corresponds to the original (Java 1) floating-point semantics. Although this mode enforces bit-for-bit reproducibility of results across JVMs, it could lead to severe performance deterioration when implemented on Intel Pentium like processors. The registers of these processors operate using IEEE 754's 80-bit double-extended format, therefore under the `strictfp` mode, both the fractional part and the exponent part have to be truncated to IEEE 754 64-bit double format at a great cost. For performance consideration, the default mode was therefore not `strictfp` anymore. All the benchmarks in this report are run under the default mode, although as far as the authors understand, no JVM has implemented `strictfp` mode yet.

2 The Test Platforms

The experiments were carried out on the following systems

- a dual processor Pentium II with 266 MHz CPU and 256 MB RAM, running Linux (Red Hat 6.0)

Table 1: Compilers used

name	note	version	flags
		Pentium II	
abf90	Absoft F90 compiler	1.01	-O
pgf90	Portland group F90 compiler	3.0-3	-O4
gcc	GNU C compiler	2.95.1	-O5
Java (IBM)	IBM JDK	1.3.0	-O
Java (Sun)	Sun JDK	1.3.0 (beta)	-O
		Sun Untra 80	
f90	Sun WorkShop F90 compiler	2.0	-fast
cc	Sun WorkShop C compiler	5.0	-fast
Java	Sun	1.3.0 (beta)	-O
		IBM Power3	
f90	XL Fortran compiler for AIX	7.1.0.0	-O3 -qarch=pwr3 -qtune=pwr3
cc	C for AIX Compiler	4.4.0.0	-O3 -qarch=pwr3 -qtune=pwr3
Java	Java RTE	1.1.8	-O

- a Sun Ultra 80 with 450 MHz CPU and 2GB RAM, running Solaris 2.6
- an IBM Power3 with 375 MHZ CPU (1.5 Gflops peak) and 1GB RAM, running AIX 4.3.3.0.

The compilers and compilation flags are listed in Table 1.

3 The Sparse Matrix Benchmark

In our first study the benchmark we choose is a simple yet important operation: that of multiplying two sparse matrices. This operation appears when forming the normal equations of interior point methods for large scale numerical optimization. It also appears, either explicitly or implicitly, in very large scale unstructured calculations where a multilevel/multigrid scheme is used. For example, in the algebraic multilevel algorithm for large sparse linear systems, the matrix on a coarse grid, A_c, is derived from the matrix on the fine grid, A_f, using the following Galerkin product:

$$A_c = P^T A_f P, \tag{1}$$

where P is the prolongation operator. This is usually one of the most time consuming parts of an algebraic multigrid algorithm. Although in practice the multiplication of the three matrices in (1) is performed in one step, rather than performed twice as a product of two matrices, in this study we only look at the product of two matrices.

Our benchmark (known as JASPA (JAva SPArse benchmark), available at `http://www.dl.ac.uk/TCSC/Staff/Hu_Y_F/JASPA`) therefore has two simple steps:

- read in the matrix A from a file
- multiply the matrix A by itself and store in matrix B ($B = A * A$)

3.1 Test problems

The test matrices are download from the MatrixMarket (`http://math.nist.gov/MatrixMarket`). These are square matrices of size ranging from 2534 to 94069. These matrices are in the MatrixMarket coordinate format. For example, for the following 5×5 sparse matrix with 8 nonzero entries,

$$\begin{pmatrix} 1 & 0 & 0 & 6 & 0 \\ 0 & 10.5 & 0 & 0 & 0 \\ 0 & 0 & .015 & 0 & 0 \\ 0 & 250.5 & 0 & -280 & 33.32 \\ 0 & 0 & 0 & 0 & 12 \end{pmatrix} \tag{2}$$

is stored in the MatrixMarket coordinate as

```
%%MatrixMarket matrix coordinate real general
 5  5  8
    1      1   1.000e+00
    2      2   1.050e+01
    3      3   1.500e-02
    1      4   6.000e+00
    4      2   2.505e+02
    4      4  -2.800e+02
    4      5   3.332e+01
    5      5   1.200e+01
```

Because matrices from MatrixMarket may contain sparse patterns without any entry values, for simplicity all matrix files have been converted to a form with only the sparse pattern. Thus for example the above matrix will be in the form

```
 5  5  8
    1      1
    2      2
    3      3
    1      4
    4      2
    4      4
    4      5
    5      5
```

All the matrix entries are set to one after reading in the sparse pattern.

3.2 I/O considerations

Reading in a matrix in F90 and C is simple. Here is the F90 code fragment

```
  ! read in the row, column sizes and number of nonzeros
  read(input_unit,*) n,m,nz
  allocate(irn(nz),jcn(nz))

  ! read in the row and column indices (sparse pattern)
  do i = 1, nz
     read(input_unit,*) irn(i),jcn(i)
  end do
```

and here is the C code fragment

```
    /* read in the row and columb sizes
       and the number of nonzeros */
    fscanf(fp,"%d %d %d",&n, &m, &nz) ;

    /* read in the matrix */
    irn = (int*) malloc((nz)*sizeof(int));
    jcn = (int*) malloc((nz)*sizeof(int));
    for (i = 0; i < nz; i++) {
      fscanf(fp,"%d %d",&irn[i],&jcn[i]);
    }
```

However the JAVA code for input needs more attention because there is more than one way of writing a JAVA code for reading from the ASCII matrix files, with quite different performances. Initially we tried the `readLine` method of the `BufferReader` class, which processes the matrix file one line at a time. The line is then turned into integers by the `String2int` method. When tested on a dual processors Pentium II running Linux and the Sun JAVA compiler version 1.2.2, this approach was found to be up to 30 times slower than C. After some trial and error, the following approach was finally adopted instead:

```
//open the matrix file
FileInputStream is = new FileInputStream(MatrixFileName);
BufferedReader bis = new BufferedReader(new InputStreamReader(is));
StreamTokenizer st = new StreamTokenizer(bis);

//read in the row and column sizes
st.nextToken(); int n = (int) st.nval;
st.nextToken(); int m = (int) st.nval;
st.nextToken(); int nz = (int) st.nval;

int[] irn = new int[nz];
int[] jcn = new int[nz];

for (int j = 0; j < nz; j++) {
    st.nextToken();
    irn[j] = (int) st.nval;
    st.nextToken();
    jcn[j] = (int) st.nval;
}
```

In this approach the whole file is read in as an *InputStream* and buffered, it is then tokenized to extract the sparse patterns. Using a *BufferedReader* is very important since unbuffered I/O can be very slow.

3.3 The Computational Kernel

Once the matrix is read in, it is converted into the internal data structure. Here the popular compacted sparse row format for storing sparse matrices is used. This comprises two integer arrays ia, ja and one double precision array a, where ia is of dimension $n+1$ and points to the start of each row in the arrays ja and a. The array ja is of dimension nz and contains the column indices, while a is also of dimension nz and contains the matrix entries.

The main kernel in the computational part of the benchmark is the code for the product of two matrices. In F90 this is of the following form, where the product $C = A * B$ is calculated. Each matrix is represented in the compacted sparse row format; for example, matrix A is represented by arrays ia, ja, a, matrix B by arrays ib, jb, b, and product matrix C by arrays ic, jc, c.

```
subroutine spmatmul(n,m,ia,ja,a,ib,jb,b,ic,jc,c)
  ! C = A*B
  integer (kind = myint), intent (in) :: n,m
  integer (kind = myint), intent (in) :: ia(*),ib(*),ja(*),jb(*)
  integer (kind = myint), intent (out) :: ic(*),jc(*)
  real (kind = myreal), intent (in) :: a(*),b(*)
  real (kind = myreal), intent (out) :: c(*)
  integer (kind = myint) :: mask(m),nz,i,j,k,icol,icol_add,neigh
  real (kind = myreal) :: aij
  ! initialise the mask array which is an array that has
  ! non-zero value if the column index already exist, in which
  ! case the value is the index of that column
  mask = 0
  nz = 0
  ic(1) = 1
  do i=1,n
     do j = ia(i),ia(i+1)-1
        aij = a(j)
        neigh = ja(j)
        do k = ib(neigh),ib(neigh+1)-1
           icol_add = jb(k)
           icol = mask(icol_add)
           if (icol == 0) then
              nz = nz + 1
              jc(nz) = icol_add
              c(nz) =  aij*b(k)
              mask(icol_add) = nz     ! add mask
           else
              c(icol) = c(icol) + aij*b(k)
           end if
        end do
     end do
     mask(jc(ic(i):nz)) = 0
     ic(i+1) = nz+1
  end do
end subroutine spmatmul
```

The main characteristic of the above computation is that although the arrays $(ia,\ ja, a)$ are accessed sequentially, the arrays (ib, jb, b) and (ic, jc, c) may be accessed randomly through indirect addressing.

In Java, the kernel looks very similar:

```
static void spmatmul_double(
                            int n, int m,
                            double[] a, int[] ia,int[] ja,
                            double[] b, int[] ib,int[] jb,
                            double[] c, int[] ic,int[] jc)
{
    int nz;
    int i,j,k,l,icol,icol_add;
    double aij;
    int neighbour;

    // extra space for FORTRAN like array indexing
    int[] mask = new int[m+1];

    // starting from one for FORTRAN like array index
    for (l = 1; l <= m; l++) mask[l] = 0;

    ic[0] = 1;
    nz = 0;

    // starting from one for FORTRAN like array indexing
    for (i = 1; i <= n; i++) {
       for (j = ia[i]; j < ia[i+1]; j++){
    neighbour = ja[j];
    aij = a[j];
            for (k = ib[neighbour]; k < ib[neighbour+1]; k++){
                icol_add = jb[k];
                icol = mask[icol_add];
                if (icol ==  0) {
                    jc[++nz] = icol_add;
                    c[nz] =  aij*b[k];
                    mask[icol_add] = nz;
                    }
                    else {
                    c[icol] +=  aij*b[k];
                }
            }
        }
        for (j = ic[i]; j < nz + 1; j++) mask[jc[j]] = 0;
        ic[i+1] = nz+1;
    }

}
```

Notice that because the matrix is stored such that the row and column entries always start from 1, in Java each array will be given one extra element so that the array addressing can start from 1. In C however it is possible to utilize the flexibility of pointer manipulation to avoid this extra element, as the following C code shows.

```
void spmatmul_double(
                     const int* n, const int* m,
                     const double *a, const int *ia,const int *ja,
                     const double *b, const int *ib,const int *jb,
                     double *c, int *ic,int *jc,
                     const int* ind_base)
{
  int nz;
  int i,j,k,icol,icol_add;
  const double *aij;
  const int *neighbour;
  int *mask, *pmask;

  mask = (int*) malloc( (*m) * sizeof(int));
  pmask = mask;

  for (i = 0; i != *m; i++) *pmask++ = 0;
  /* shift the index base for fortran like indexing: *ind_base=1 */
  ib -= *ind_base;
  jb -= *ind_base;
  b -= *ind_base;
  jc -= *ind_base;
  c -= *ind_base;
  mask -= *ind_base;

  aij = a;
  neighbour = ja;
  nz = -1+*ind_base;
  *ic = 1;
  for (i = 0; i != *n; i++) {
    for (j = ia[i]; j != ia[i+1]; j++){
      for (k = ib[*neighbour];k != ib[*neighbour+1];k++){
        icol_add = jb[k];
        icol = mask[icol_add];
        if (icol ==  0)
          {
            jc[++nz] = icol_add;
            c[nz] =  (*aij)*b[k];
            mask[icol_add] = nz;
          }
        else
          {
            c[icol] +=  (*aij)*b[k];
           }
      }
      aij++;
      neighbour++;
    }
    for (j = *ic; j != nz + 1; j++) mask[jc[j]] = 0;
    *(++ic) = nz+1;
  }
  mask += *ind_base;
  free(mask);

}
```

3.4 Performance for sparse matrix multiplication

Table 2 shows the I/O performance of Java compared with F90 and C on the three platforms. In this table and Table 3, the timing for the F90 (in the case of the Pentium it is the Absoft F90) is used as a benchmark. The timings for other compilers are divided by the timings for the F90 compiler, and averaged to give a score in the last row of the tables. With this measure, the F90 compiler always comes out with a score of 1. The lower the score, the better the algorithm.

On the Pentium system, the I/O performance of both versions of Java are about 25-40% worse than C, but are about three times better than the Absoft F90! The disappointing I/O performance of the Absoft compiler is however not inherent to F90. Using the Portland Group F90 compiler, the I/O performance is close to that of C. However, as we will see later, the computing performance of the Portland Group F90 compiler is rather disappointing. This was known from an earlier experience [3]. I/O performance of Java (Sun) is slightly (about 10%) better than that of Java (IBM).

On the Sun Ultra 80, it is surprising that the I/O of the C version is 70% slower than F90, while the Java took 3.5 times the I/O time of the F90.

On the IBM Power3, the F90 I/O took the least time, followed by Java and C, which are both about 60-75% slower.

One annoying feature of Java is that all Java Virtual Machines assume a certain fixed heap size (this being 30 MB on the IBM platform in our experiment), and for large problems one has to specify the amount of memory needed explicitly using a flag `-mx<size>`, which can be inconvenient since in our experiments we do not know in advance the amount of memory needed.

The compute performance for the sparse matrix multiplications is compared in Table 3. On the Pentium platform, the Absoft F90 compiler performed the best. It is interesting that the C version of the matrix multiplication performed just as well. The Java (IBM) version runs about 27% slower. The big disappointment is the Portland F90, which needed 75% more time (this is reduced to about 67% on another Linux system which has the newer 3.1-3 version of Portland F90)! Java (Sun) is around 35% slower than Java (IBM), and 70% slower than the Absoft F90.

On the Sun Ultra 80, the F90 and C versions have almost the same performance, but the Java version does not perform well at all, requiring on average 2.4 times the CPU time!

On the IBM Power3, the C version is 30% slower than the F90 version. The Java version is about 2.9 times slower! Note however from Table 1 that the Java compiler used is version 1.1.8, rather than the latest version 1.3 (beta). This is because the IBM Power3 used in this experiment runs under AIX 4.3.3.0. To install the Java version 1.3 (beta) would require an operating system upgrade to or above AIX 4.3.3.10, which we were unable to obtain permission to do. We would expect that with a newer Java compiler and JVM, the gap of computing performance may be closer. It is also worth noting that prior to benchmarking on the Power3 platform, we have

Table 2: Comparing the I/O performance (in seconds) of Java with F90 and C for reading in sparse matrices in MatrixMarket coordinate form as ASCII files.

Pentium II

matrices	n	nz	abf90	pgf90	gcc	Java (IBM)	Java (Sun)
af23560.mtx	23560	484256	13.779	4.426	3.490	4.821	4.441
bcsstk30.mtx	28924	1036208	29.921	9.564	7.530	9.445	9.336
e40r0000.mtx	17281	553956	15.200	5.037	3.930	5.278	4.913
fidap011.mtx	16614	1091362	30.345	9.787	7.720	10.673	10.090
fidapm11.mtx	22294	623554	20.024	5.678	4.470	5.898	5.641
memplus.mtx	17758	126150	3.745	1.096	0.880	1.840	1.112
qc2534.mtx	2534	463360	13.360	3.891	3.110	3.766	3.697
s3dkt3m2.mtx	90449	1921955	62.245	18.034	14.180	18.549	18.061
score			1.000	0.306	0.242	0.341	0.304

Sun Ultra 80

matrices	n	nz	F90	C	Java
af23560.mtx	23560	484256	0.989	1.720	3.560
bcsstk30.mtx	28924	1036208	2.044	3.720	7.672
e40r0000.mtx	17281	553956	1.106	1.910	3.966
fidap011.mtx	16614	1091362	2.109	3.760	7.741
fidapm11.mtx	22294	623554	1.263	2.210	4.552
memplus.mtx	17758	126150	0.293	0.410	0.906
qc2534.mtx	2534	463360	0.896	1.450	3.090
s3dkt3m2.mtx	90449	1921955	3.831	7.100	4.386
score			1.000	1.711	3.564

IBM Power3

matrices	n	nz	F90	C	Java
af23560.mtx	23560	484256	1.970	3.510	3.247
bcsstk30.mtx	28924	1036208	4.320	7.550	7.061
e40r0000.mtx	17281	553956	2.300	4.020	3.624
fidap011.mtx	16614	1091362	4.480	7.770	7.118
fidapm11.mtx	22294	623554	2.700	4.510	4.168
memplus.mtx	17758	126150	0.488	0.887	0.776
qc2534.mtx	2534	463360	1.755	3.190	2.671
s3dkt3m2.mtx	90449	1921955	8.280	14.310	13.528
score			1.000	1.756	1.592

experimented on a PowerPC Silver processor. It was found that the I/O performance of C and Java versions are no more than 25% slower than the F90 version, while in terms of the compute performance, C and Java are about 17% and 85% slower, respectively. In view of these, we do not understand why Java performed so poorly on the Power3. (Or: This may indicate that Java is not able to utilise the extra floating point pinelines that are available on the Power3 and the Sun Ultra80, but not available on the PowerPC and the Pentium II processors)

4 The SciMark2 Benchmark

SciMark 2.0 (`http://math.nist.gov/scimark2/`) is a Java benchmark for scientific and numerical computing. It measures several computational kernels and reports a composite score in approximate Mflops/s. This benchmark was developed at the US National Institute of Standards and Technology (NIST). Part of the benchmark can also be found in the Java Grande Forum Benchmark Suite (`http://www.epcc.ed.ac.uk/javagrande/javag.html`). This benchmark contains codes on FFT, SOR (Successive Over-Relaxation over a 2D grid), Monte-Carlo integration, Sparse matmult (Sparse matrix vector multiplications) and LU factorization. We have chosen this benchmark because the same benchmark is available both in Java and C, allowing us to compare the two languages. There are many other Java benchmarks available, see `http://www.epcc.ed.ac.uk/javagrande/links.html`.

Table 4 contains the result of this benchmark. The compiler flag used with gcc is `-O2 -funroll-loops`, which is the flag that comes with the makefile of this benchmark. The loop unrolling does seem to improve the performance of the LU factorization and the sparse matrix vector multiplications, although it had minimal effect on our previous sparse matrix multiplication benchmark, and was therefore not used for that. We used two settings of the benchmark: SMALL and LARGE. When the SMALL setting is chosen, the problems tend to be able to fit into the cache. The LARGE setting is useful in measuring the capability of the memory subsystem since the size of the benchmark at that setting is designed to be much bigger than most low-level caches ($>$ 2MB).

On the Pentium system, for the SMALL setting, Java (IBM) is about 30% slower ($45.80/35.50 - 1$) than C, while for larger problems it is about 40% slower. The Java (Sun) again does not perform as well as Java (IBM), and is on average 33% slower than Java (IBM) on SMALL setting and 12%

Table 3: Comparing the computing performance (in seconds) of Java with F90 and C for multiplying two sparse matrices.

Pentium II							
matrices	n	nz	abf90	pgf90	gcc	Java (IBM)	Java (Sun)
af23560.mtx	23560	484256	1.765	3.584	1.820	2.182	2.824
bcsstk30.mtx	28924	1036208	5.034	8.600	5.780	6.484	9.319
e40r0000.mtx	17281	553956	2.790	4.870	3.100	3.618	5.093
fidap011.mtx	16614	1091362	8.997	14.110	10.730	12.358	16.948
fidapm11.mtx	22294	623554	3.232	5.896	3.380	3.899	5.202
memplus.mtx	17758	126150	2.631	4.719	2.390	2.863	3.757
qc2534.mtx	2534	463360	10.266	12.508	12.760	13.348	19.072
s3dkt3m2.mtx	90449	1921955	5.159	10.778	5.600	6.948	8.329
score			1.000	1.747	1.096	1.267	1.709

Sun Ultra 80					
matrices	n	nz	F90	C	Java
af23560.mtx	23560	484256	1.081	1.070	2.529
bcsstk30.mtx	28924	1036208	3.323	3.510	8.530
e40r0000.mtx	17281	553956	1.862	1.940	4.751
fidap011.mtx	16614	1091362	5.972	6.560	15.195
fidapm11.mtx	22294	623554	2.063	2.080	4.428
memplus.mtx	17758	126150	1.656	1.440	2.793
qc2534.mtx	2534	463360	5.958	6.780	16.263
s3dkt3m2.mtx	90449	1921955	3.368	3.340	8.849
score			1.000	1.024	2.399

IBM Power3					
matrices	n	nz	F90	C	Java
af23560.mtx	23560	484256	0.380	0.482	1.085
bcsstk30.mtx	28924	1036208	1.100	1.435	3.264
e40r0000.mtx	17281	553956	0.598	0.783	1.796
fidap011.mtx	16614	1091362	1.865	2.580	5.524
fidapm11.mtx	22294	623554	0.730	0.907	1.935
memplus.mtx	17758	126150	0.490	0.573	1.339
qc2534.mtx	2534	463360	1.665	2.360	5.428
s3dkt3m2.mtx	90449	1921955	1.220	1.575	3.513
score			1.000	1.298	2.914

slower on the LARGE setting.

On the Sun Ultra 80, Java is about 170% slower than C on the SMALL setting and 103% slower on the LARGE setting!

On the IBM Power3, Java is about 2.19 times slower than C on the SMALL setting and 1.97 times slower on the LARGE setting! As in Section 3, we did run the SciMark2 on a PowerPC Silver platform as well. It was found that on the IBM PowerPC Silver processor, Java was about 42% slower than C on the SMALL setting and only 24% slower on the LARGE setting. Again we do not understand why the performance of Java is so poor on the Power3.

5 Conclusions

In this report the performance of Java has been compared with that of Fortran 90 and C on two benchmarks of particular interest to scientific and engineering applications. It was found that in comparison with F90 and C, the I/O and compute performance of Java various from 30% slower to about 3 times slower, depending on the platforms and the compilers (and the JVMs). The best relative performance is achieved on a Pentium II, where it was found that the IBM Java (Version 1.3) yields code that is about 30-40% slower in computing and I/O performance. Given the special advantages of Java, we felt that this slightly inferior performance of JAVA may be acceptable for many applications. In any case the performance of Java seen in this work is certainly a big improvement over the situation only 1-2 years ago when it was generally perceived that Java could only achieve, say, 20% the speed of Fortran and C.

A number of issues make it difficult for Java to be used in large scientific and engineering applications, these include the lack of efficient multidimensional array, the lack of complex number support, no operator overloading, inability to take advantage of fused multiply-add and associativity of operations in compiler optimizations, the lack of a Math Library that produces the same results on all Java platform, and the difficulties in interfacing Java with other languages. There is currently many activities to address these issues. These include some closely related/intersected forums and activities, such as the Java Grande Forum (`http://www.javagrande.org`), the JN-T (Java Numerical Toolkit) project (`http://math.mist.gov/jnt`), which includes proposals for Java BLAS interface, and the JNL (Java Numerical Library, `http://www.vni.com/products/wpd/jnl`). It can be expected that compiler technology and support libraries will continue to improve. We

Table 4: Results (in Mflops/s) of the SciMark2 benchmark for Java and C.

Pentium II

	SMALL			LARGE		
Applications	gcc	Java (IBM)	Java (Sun)	gcc	Java (IBM)	Java (Sun)
FFT	39.93	24.70	14.97	5.80	6.75	5.42
SOR	79.49	74.22	60.15	50.29	38.09	33.85
MonteCarlo	10.69	4.17	3.69	10.65	4.16	3.69
Sparse matmult	41.80	28.42	20.63	19.05	15.64	13.33
LU	57.08	24.70	14.97	27.71	6.75	5.42
Composite Score	45.80	35.50	26.63	22.70	16.24	14.50

Sun Ultra 80

	SMALL		LARGE	
Applications	C	Java	C	Java
FFT	63.38	17.27	10.71	7.32
SOR	135.32	54.66	89.96	44.87
MonteCarlo	17.25	5.82	16.99	5.88
Sparse matmult	48.91	22.40	30.48	17.27
LU	94.03	17.27	52.78	7.32
Composite Score	71.78	25.71	40.18	19.78

IBM Power3

	SMALL		LARGE	
Applications	C	Java	C	Java
FFT	117.22	78.37	10.89	11.31
SOR	147.32	94.19	142.47	91.72
MonteCarlo	22.60	5.07	22.60	5.09
Sparse matmult	120.92	71.89	122.49	75.16
LU	341.33	78.37	244.20	11.31
Composite Score	149.88	68.32	108.53	55.17

believe we are closer to a time when new scientific and engineering applications could equally well be written in Java to benefit from its advantages, without the need to worry about the loss of performance.

References

[1] S. P. Midkiff, J. E. Moreira and M. Snir, Optimizing Array Reference Checking in JAVA Programs, IBM Systems Journal, 37 (409-453), 1998.

[2] C. Rijk, Binaries vs Byte-codes, Ace's Hardware, June 27, 2000. (`http://www.aceshardware.com/Spades/read.php?article_id=153`).

[3] K. Maguire and B. Searle, Low cost HPC?, in Proceeding of the 9th Daresbury Machine Evaluation Workshop, R.J. Allan, M.F. Guest, B.G. Searle, K. Maguire (Eds), Daresbury Machine Evaluation Workshop, CLRC Daresbury Laboratory, Daresbury, 26-27 November, 1998.