Implementation of NAS Parallel Benchmarks in Java

Michael Frumkin[*], Matthew Schultz[*], Haoqiang Jin[*], and Jerry Yan[†]
Numerical Aerospace Simulation Division
NASA Ames Research Center
{frumkin,mshultz,hjin,yan}@nas.nasa.gov

Abstract

A number of features make Java an attractive but a debatable choice for High Performance Computing (HPC). In order to gauge the applicability of Java to the Computational Fluid Dynamics (CFD) we have implemented NAS Parallel Benchmarks in Java. The performance and scalability of the benchmarks point out the areas where improvement in Java compiler technology and in Java thread implementation would move Java closer to Fortran in the competition for CFD applications.

1. Introduction

The portability, expressiveness and safety of the Java language supported by rapid progress in Java compiler technology have created interests in the HPC community to evaluate Java technology on computationally intensive tasks. Java threads and networking capabilities well position Java for programming on Shared Memory Parallel (SMP) computers and on computational grids. On the other hand, Java safety, lack of light weight objects and intermediate byte code compilation stage create a number of challenges in achieving high performance of Java code. The challenges are met by work on implementation of efficient Java compilers [9],[5] and by extending Java with classes implementing the data types used in HPC [10].

In this paper, we report an implementation of the NAS Parallel Benchmarks (NPB) [1] in Java. The benchmarks are accepted by HPC community as an instrument for evaluating performance of parallel computers, compilers and tools (in [8] mentioned that "Parallel Java versions of Linpack and NAS Parallel Benchmarks would be particularly interesting"). The implementation of the NPB in Java builds a base for watching the progress of Java technology, for evaluating Java as a choice for programming aerospace applications and allows to identify the areas where improvement in Java compilers would give the most gain in performance of the Computational Fluid Dynamics (CFD) codes written in Java.

[*]MRJ Technology Solutions, Inc. M/S T27A-2, NASA Ames Research Center, Moffett Field, CA 94035-1000.
[†]NAS Division. M/S T27A-2, NASA Ames Research Center, Moffett Field, CA 94035-1000.

Our implementation NPB in Java is derived from the optimized NPB3.2-serial, [7] versions written in Fortran, except IS written in C. The NPB3.2-serial version was previously used by us for development of HPF [3] and OpenMP [7] versions. We start with an evaluation of Fortran to Java conversion options by comparing performance of basic CFD operations. The most efficient options are then used to translate the Fortran to Java. Then we parallelize NPB using Java threads and master-workers load distribution model. Finally we profile the code on SGI Origin2000 and on SUN Enterprise10000 machines and analyze its performance.

2. NAS Parallel Benchmarks

NAS Parallel Benchmarks (NPB) were derived from CFD codes [1]. They were designed to compare the performance of parallel computers and are recognized as a standard indicator of computer performance. NPB consists of five kernels and three simulated CFD applications. The five kernels represent the computational core of five numerical methods used in CFD applications. The simulated CFD applications mimic some data movement and computations found in full CFD codes.

An algorithmic description of the benchmarks (pencil and paper specification) was given in [5] and is referred to as NPB-1. A source code implementation of most benchmarks (NPB-2.0) was described in [2]. The latest release of NPB-2.3 contains MPI source code for all of benchmarks and a stripped-down serial version (NPB-2.3-serial). The serial version was intended to be used as a starting point for parallelization tools and compilers and for other types of parallel implementations. For completeness of discussion we outline the seven benchmarks (except for embarrassingly parallel, EP) that have been implemented in Java.

BT is a simulated CFD application that uses an implicit algorithm to solve 3-dimensional (3-D) compressible Navier-Stokes equations. The finite differences solution to the problem is based on an Alternating Direction Implicit (ADI) approximate factorization that decouples the x, y and z dimensions. The resulting system is Block Tridiagonal of 5x5 blocks and is solved sequentially along each dimension.

SP is a simulated CFD application that has a similar structure to BT. The finite differences solution to the problem is based on a Beam-Warming approximate factorization that decouples the x, y and z dimensions. The resulting system of Scalar Pentadiagonal linear equations is solved sequentially along each dimension.

LU is a simulated CFD application that uses symmetric successive over-relaxation (SSOR) method to solve a seven-block-diagonal system resulting from finite-difference discretization of the Navier-Stokes equations in 3-D by splitting it into block Lower and Upper triangular systems.

FT contains the computational kernel of a 3-D fast Fourier Transform (FFT). FT performs three series of one-dimensional FFTs, one series for each dimension.

MG uses a V-cycle Multi Grid method to compute the solution of the 3-D scalar Poisson equation. The algorithm works iteratively on a set of grids that are made between the coarse and fine grids. It tests both short and long distance data movement.

CG uses a Conjugate Gradient method to compute an approximation to the smallest eigenvalue of a sparse unstructured matrix. This kernel tests unstructured computations and communications by using a diagonally dominated matrix with randomly generated locations of entries.

IS performs sorting of integer keys using linear time Integer Sorting algorithm based on computation of the key histogram. IS is the only benchmark written in C.

Our implementation is based on the optimized version of NPB-2.3-serial as described in [7].

3. Fortran to Java Translation

Java is a more expressive language than Fortran. This implies a simple translation from Fortran to Java as well as a challenge in achieving the performance of Java code matching the performance of Fortran code. There are two general options in translating Fortran code into Java: literal translation and object oriented translation. In the literal translation the procedural structure of application is kept intact, arrays are translated to Java arrays, complex numbers are translated into (Re,Im) pairs and no other objects are used except the objects having the methods corresponding to the original Fortran subroutines. The object oriented translation translates multidimensional arrays, complex numbers, matrices and grids into appropriate classes and changes the code structure from the procedural style to the object oriented style. Advantage of the literal translation is that mapping of the original code to the translated code is direct and the potential overhead for access and modification of corresponding data is less than in the object oriented translation. On the other hand, the object oriented translation results in a better structured code and allows to advise the compiler in special treatment of particular classes, for example using semantic expansion (inlining), [9], [11]. Since we are interested in high performance code we chose the literal translation as resulting in a faster code.

In order to compare efficiency of different options of the literal translation and to form a baseline for estimation of the quality of our implementation of the benchmarks we chose a few basic CFD operations and implemented them in Java. Relative performance of different implementations of basic operations gives us a guide for Fortran-to-Java translation. As basis operations we chose the operations we have used for building HPF performance model [3]:

- loading/storing array elements
- filtering an array with a local kernel (the kernel can be a first or second order star-shaped stencil as in BT, SP and LU, or compact 3x3x3 stencil as in the smoothing operator in MG);
- matrix vector multiplication of a 3D array of 5x5 matrices a 3D array of five-dimensional vectors (manipulation with 3D arrays of five-dimensional vectors is a common CFD operation);
- performing a reduction sum of 4D array elements.

We used two ways to implement these operations: with linearized arrays and with preserving the number of array dimensions. The version with preserving array dimension was 2-8 times slower than the linearized version on SGI Origin2000 (Java 3.1), on Sun Enterprise10000 (Java 1.2.2) the ratio was within 2-5. So we have decided to translate Fortran arrays into linearized Java arrays and present profiling data for the linearized translation only. The performance of serial and multithreaded implementations are compared with the Fortran implementation. The results on SGI Origin2000 are summarized in Table 1.

TABLE 1. Execution time in seconds of basic CFD operations on SGI Origin 2000 for f77, Java serial and Java multithreaded implementations (grid size 81x81x100). The column number is the number of threads.

Operation	f77	Serial	1	2	4	8	12	16
1. Assignment (10 iterations)	0.127	1.145	1.253	0.585	0.298	0.211	0.158	0.125
2. First Order Stencil	0.060	0.513	0.533	0.283	0.142	0.099	0.057	0.048
3. Second Order Stencil	0.069	0.767	0.793	0.411	0.205	0.151	0.125	0.067
4. Matrix vector multiplication	0.624	7.221	7.271	3.824	2.036	1.228	0.846	0.684
5. Reduction Sum	0.048	0.653	0.686	0.379	0.201	0.139	0.082	0.081

We can suggest some conclusions from the profiling data.

- Java serial code is factor of 8.6 (First Order Stencil) to 13.6 (Reduction Sum) times slower than the corresponding Fortran operations.
- Thread overhead (serial column versus 1 thread column) contributes only a few percent to the execution time.
- The external loop of the operations is completely parallel and speedup with 16 threads is around 11 for computationally expensive operations (2,3 and 4) and is around 9 for less intensive operations (1 and 5).

For more detailed analysis of the basic operations we used an SGI profiling tool `perfex`. `Perfex` uses 32 hardware counters to count issued/graduated integer and floating point instructions, load/stores, primary/secondary cache misses. The profiling with `perfex` shows that the Java/Fortran performance correlates well with the ratio of the total number of executed instructions in the two codes. Also, the Java code executes as twice as many floating point instructions as the Fortran code, confirming that Just-In-Time (JIT) compiler does not use the "`madd`" instruction since the last is not compatible with Java rounding error model, [9].

Once we chose a literal translation with array linearization of the Fortran to Java we automate the translation with using `emacs` regular expressions. For example, to translate Fortran array

```
REAL*8 u(5,nx,ny,nz)
u(m,i,j,k)=...
```

into Java array:

```
double u[]=new double[5*nx*ny*nz];
int usize1=5,
        usize2=usize1*nx,
        usize3=usize2*ny;
    u((m-1)+(i-1)*usize1+(j-1)*usize2+(k-1)*usize3)=...
```

we translated the declaration by hand and translated the references to array elements by using the macro

```
arrayname(\([^,]+\),\([^,]+\),\([^,]+\),\([^)]\)) =>
arrayname[((\1-1)+(\2-1)*size1+(\3-1)*size2+(\4-1)*size3].
```

Similarly, `DO` loops were converted to Java `for` loops using the macro

```
do[ ]+\([-+a-z0-9]+\)[ ]*=[ ]*\([-+a-z0-9]+\)[ ]*,[ ]*\(.+\)
=> for(\1=\2;\1<=\3;\1++){.
```

Several Fortran constructs were changed to Java through context free replacement. These include all boolean operators, all type declarations (except character arrays which were converted to Java strings), some `if-then-else` statements, comments, the `call` statement, and line continuation marker. The semiautomatic translation allowed us to translate about 70% of Fortran code to Java. In general, even the literal translation requires parsing the Fortran code and translation of the parse tree to a Java equivalent, for example for labeled `DO` loops, `common`, `format`, and `IO` statements.

In general, each benchmark has the base class and derived main and workers classes. The base class contains all global and common variables as members. The main class contains one method per each Fortran subroutine, including `main`. There is one worker class per each parallelizable Fortran subroutine (see discussion in the next section). The main class has two additional methods: `runBenchmark()` executed in serial mode and `run()` executed in parallel mode. `runBechmark()` calls all methods exactly in the same sequence as in the original Fortran code. The `run()` method manages the worker threads in the parallel mode. Commonly used functions `Timer` and `Random` are implemented as separate classes and are imported into each benchmark. All the benchmarks are united into the NPB package.

4. Using Java Threads for Parallelization

A significant appeal of Java for parallel computing steams from presence of threads as part of Java language. On a multiprocessor machine the Java Virtual Machine (JVM) can assign different threads to different processors and speed up execution of the job if the work is well divided between threads. Conceptually, Java threads are close to the OpenMP threads, so we used the OpenMP version of the benchmarks, implemented at NASA Ames Research Center, [7] as a prototype for multithreading.

The base class (hence all other classes) of each benchmark was derived from class `java.lang.Thread`, so all benchmark objects are implemented as threads. The instance of the main class was designated as the master to control the synchronization of the worker objects. Workers were switched between blocked and runnable states with `wait()` and `notify()` methods of the `Thread` class.

The initialization of the threads and partitioning the work between them are performed in the main class. The partition is accomplished by specifying the starting and ending iterations of the outer `for` loops of each worker. The master thread dispatches the job to each worker, starts the workers and then waits until all workers are finished (see Figure 1). Each worker thread is then started and immediately goes into a blocked state on the condition that the variable done is true, then it performs the work and notifies the master that the work is done. The while loop around the wait call prevents an arbitrary notify call from waking a thread before its time. All CFD codes are placed in the step method.

```
Master's code

for(i=0;i<num_threads;i++)
  worker[i].done=false;
for(i=0;i<num_threads;i++)
  synchronized(worker[i]){
    worker[i].notify();
  }
for(i=0;i<num_threads;i++)
  while(!worker[i].done){
    try{wait();}
   catch(InterruptedException ie){}
  }
```

```
Worker's code

while(done){
  try{wait();}
  catch(InterruptedException ie){}
}
step();
done=true;
synchronized(master){master.notify();}
```

FIGURE 1. Master-Worker Thread Synchronization.

The described model of thread synchronization is applicable only if there is no dependence between workers: each worker processes the job dispatched by the master independently on other workers. Such dependences, however, exist in LU, where the computations are pipelined. We have implemented the pipelined computations with relay-race thread synchronization. The master thread starts all workers but waits only for the last worker to finish (the relay-race mechanism guarantees that all other workers have finished already). The workers are synchronized between themselves with the job relay, as shown in Figure 2.

```
while(true){
  while(done)
    try{wait();}
    catch(InterruptedException ie){}
  for(k=1;k<nz;k++){
    if(id>0)
      while(todo<=0)
        try{wait();}
        catch(InterruptedException ie){}
    step(k);
    todo--;
    if(id<num_threads-1)
      synchronized(worker[id+1]){
        worker[id+1].todo++;
        worker[id+1].notify();
      }
  }
  done=true;
  if(id==num_threads-1) synchronized(master){master.notify();}
}
```

FIGURE 2. Worker relay-race synchronization for pipelineing computations.

5. The Performance

We have tested the benchmarks on classes S,W and A for different problem size; the performance is shown for the class A as the largest of tested classes. The full performance results for IS, CG and MG will be included in the final version of the paper. The tests were performed on two shared memory machines: SGI Origin2000 (195 MHz, 24 processors) and SUN Enterprise 10000 (333 MHz, 16 processors). On the Origin we used Java version "3.1 (Sun 1.1.5)" (in the final version of the paper we will compare its performance with performance of SGI latest Java 3.1.1). On the Enterprise we used Java 1.1.3 (we also tested the latest Java 1.2.2, but its scalability was significantly worse than that of Java 1.1.3). The performance results are summarized in Table 2 and Table 3, and the profiles are shown in Figure 3 and Figure 4 respectively.

TABLE 2. Benchmarks time in seconds on SGI Origin2000 (195 MHz, 24 processors), Java version "3.1 (Sun 1.1.5)". The column number is the number of threads.

Thread number	Serial	1	2	4	8	9	12	16
BT.A Java	14605.2	16447.6	8448.4	4375.6	2550.9	2230 2	1700.6	1367.4
BT.A f77-OpenMP	1396.0	1477.8	749.0	378.6	-	177.9	-	106.1
SP.A Java	9948.0	10293.7	5640.4	3001.4	1699.3	1690.3	1361.3	1254.9
SP.A f77-OpenMP	1224.3	1227.1	646.0	350.4	175.0	160.8	-	91.4
LU.A Java	14684.8	17585.7	10330.0	4901.6	2511.1	2335.2	1797.7	1655.9
LU.A f77-OpenMP	1248.7	1234.4	585.6	336.2	184.4	-	-	96.1
FT.A Java	1088.1	1135.5	657.1	336.4	171.9	-	145.7	105.9
FT.A f77-OpenMP	113.1	114.5	60.2	30.9	16.2	-	-	8.6
IS.A Java	17.0	[a]	[a]	19.4	14.5	-	[a]	[a]
IS.A C-Serial	7.9	-	-	-	-	-	-	-
CG.A Java	182.2	[a]	[a]	[a]	[a]	-	[a]	[a]
CG.A f77-OpenMP	46.0	47.7	28.6	14.0	4.8	-	-	2.5
MG.A Java	[a]	[a]	[a]	[a]	[a]	-	-	[a]
MG.A f77-OpenMP	47.3	47.6	27.1	13.7	7.0	-	-	3.9

a. The number will be supplied in the final version of the paper.

TABLE 3. Benchmarks time in seconds on SUN Enterprise10000 (333 MHz, 16 processors), Java version "1.1.3"[a]. The column number is the number of threads.

	Serial	1	2	4	8	9	12	16
BT.A Java	13609.5	14671.3	7381.7	3846.3	2305.0	2042.7	1782.7	1762.2
SP.A Java	10235.8	11108.1	5692.9	3409.3	2095.5	1899.1	1862.1	1671.2
LU.A Java	12344.5	13578.9	6843.3	3765.7	2077.3	1892.7	1730.2	1745.4
FT.A Java	1104.6	1318.8	674.7	384.2	342.7	-	353.4	363.3
IS.A Java	21.1	25.4	13.7	8.5	10.6	-	13.9	11.8
CG.A Java	203.8	[b]	[b]	[b]	[b]	-	[b]	[b]
MG.A Java	[b]	[b]	[b]	[b]	[b]	-	[b]	[b]

a. Performance tests with Java1.2.2 showed that serial and single thread performance improved by 5%-10% however 12 threads performance became worse by 50%-100%

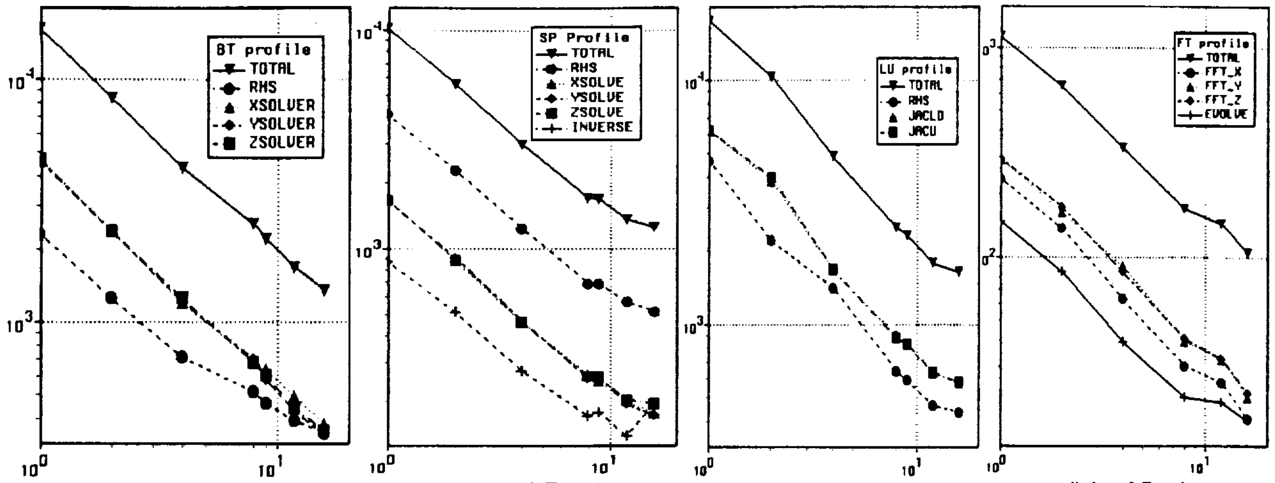

FIGURE 3. Benchmarks profile on SGI Origin2000 (24 proc. machine). The horizontal axis shows number of threads, the vertical axis shows the execution time in seconds.

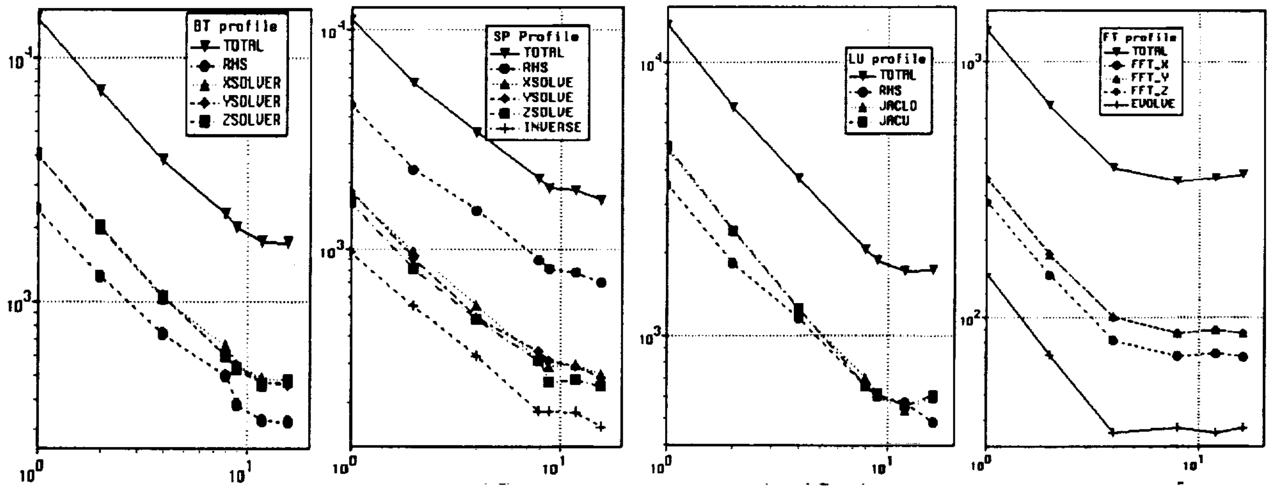

FIGURE 4. Benchmarks profile on SUN Enterprise 10000 (16 proc. machine). The horizontal axis shows number of threads, the vertical axis shows the execution time in seconds.

We can suggest the following conclusions from the performance results. There are 2 groups: benchmarks BT, SP, LU and FT working on structured grids and benchmarks IS and CG involving unstructured computations. For the first group the serial Java/Fortran execution time ratio is within interval 8-12 which is close to 8-14 interval for the basic CFD operations (Section 3). It means that our implementation of the benchmarks does not introduce additional performance overhead to the overhead of the basic operations. For the second group Java/NPB execution time ratio is within 2-4 interval. This separation in the two groups may be explained by the fact that the f77 compiler optimizes the regular computations much better than the Java compiler.

The benchmarks working on structured grids are heavily involve the basic CFD operations and any performance improvement of the basic operations would essentially improve performance of the benchmarks. Such improvement can be achieved in three directions. Firstly, an improvement in JIT must reduce the ratio of Java/Fortran instructions (which currently is factor of 10) for executing the basic operations. Secondly, the Java rounding error should be compatible with well accepted for HPC computations the "madd" instruction. Thirdly, in all benchmarks working on structured grids the array sizes and loop bounds are constants and simple compiler optimization can move bound checking out of the loop (cf. [9]) without compromising the code safety.

Multithreading of the Java benchmarks introduces overhead about 10%. The speedup of BT, SP and LU with 16 threads is in the range 6-12 (efficiency 0.38-0.75). The low efficiency of FT on SUN Enterprise is explained by inability of JVM to use more than 4 processors to run applications requiring significant amount of memory (FT.A uses about 350 Mb). An artificial increase in the memory use for other benchmarks also resulted in drop of scalability for more than 4 threads. The lower scalability of LU can be explained by the fact that it performs the thread synchronization inside a loop over one grid dimension, thus introducing higher overhead due to thread relay-racing mechanism *. The low scalability of IS was expected since the amount of work performed by each thread is small relative to other benchmarks. The scalability of IS would improve for larger classes.

The profiles of different subroutines are shown in Figure 3 and Figure 4. All subroutines, except RHS, show similar degradation of the efficiency, which is close to the 50%-60% for16 threads and close to the efficiency of the basic CFD operations. The lower efficiency of RHS can be explained by the fact that it processes larger number of independently synchronized threaded nests and the nests are smaller, resulting in larger multithreading overhead than in other subroutines.

6. Related Work

In our implementation of NPB only Java threads were used. The University of West-

*By the same reason OpenMP version of LU has less parallel efficiency than other benchmarks.

minster's Performance Engineering Group at the School of Computer Science used the Java JNI (Java Native Interface) to create a system dependent Java MPI library. They also used this library to implement the NAS benchmarks FT and IS using javaMPI [6]. The Westminster version of javaMPI can be compiled on any system with Java 1.0.2 and LAM 6.1.

The University of Adelaide's Distributed and High Performance Computing Group, has also released the NAS benchmarks EP and IS (with FT, CG and MG under development), [8] along with many other benchmarks in order to test Java's suitability for grand applications.

7. Conclusions

Although the performance of the implemented benchmarks in Java is not comparable to Fortran and C at this time, using the performance enhancing methods detailed in [9], serial performance could be improved to near Fortran-like performance. Efficiency of parallelization with threads is about 0.5 for up to 16 threads and is lower than efficiency of parallelization with OpenMP, MPI and HPF. However, with several groups working on MPI for Java, improvements in parallel performance and scalability seem inevitable as well. The attraction of Java as a numerically intensive applications language is primarily driven by its ease of use, portability and high expressiveness, which, in particular, allows to express parallelism. If it is made to run faster through methods that have already been researched extensively, such as high order loop transformations, semantic expansion and a wider availability of traditionally optimized native compilers, together with an implementation of multidimensional arrays and complex numbers it could be an attractive environment for HPC applications.

8. References

[1] D. Bailey, J. Barton, T. Lasinski, and H. Simon (Eds.), *The NAS Parallel Benchmarks,* NAS Technical Report RNR-91-002, NASA Ames Research Center, Moffett Field, CA, 1991, http://www.nas.nasa.gov/Software/NPB/.

[2] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, M. Yarrow. *The NAS Parallel Benchmarks 2.0.* Report NAS-95-020, Dec. 1995. http://science.nas.nasa.gov/Software/NPB.

[3] M. Frumkin, H. Jin, J. Yan. *HPF Implementation of* NPB2.3. Proceedings of ISCA 11th International Conference on Parallel and Distributed Computing Systems, Chicago, Il, September 2-4, 1998, 8 pp.

[4] M. Frumkin, H. Jin, J. Yan. *Implementation of NAS Parallel Benchmarks in High Performance Fortran*. CDROM version of IPPS/SPDP 1999 Proceedings, April 12-16, 1999, San Juan, Puerto Rico, 10 pp.

[5] *The GNU Compiler for the Java^{tm} Programming Language*. http://sourceware.cygnus.com/java.

[6] V. Getov, S. Flynn-Hummel, S. Mintchev. *High-Performance Parallel Programming in Java: Exploiting Native Libraries*. Proceedings of the 1998 ACM Workshop on Java for High-Performance Network Computing, 10 pp., http://perun.hscs.wmin.ac.uk/JavaMPI.

[7] H. Jin, M. Frumkin, J. Yan. *The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance*. NAS Technical Report RNR-991-011, NASA Ames Research Center, Moffett Field, CA, 1999, http://www.nas.nasa.gov/Software/NPB/.

[8] J.A. Mathew, P.D. Coddington and K.A. Hawick. *Analysis and Development of Java Grande Benchmarks*. Proc. of the ACM 1999 Java Grande Conference, San Francisco, June 1999, 9 pp., http://www.cs.ucsb.edu/conferences/java99/program.html.

[9] S.P. Midkiff, J.E. Moreira, M. Snir. *Java for Numerically Intensive Computing: from Flops to Gigaflops*. Proceedings of FRONTIERS'99, pp. 251-257, Annapolis Maryland, February 21-25, 1999.

[10] S.P. Midkiff, J.E. Moreira, M. Gupta, F. Artigas, *Numerically Intensive Java*. http://www.alphaWorks. ibm.com/tech/ninja. Analysis and Development of Java Grande Benchmarks.

[11] P. Wu, S. Midkiff, J. Moreira and M. Gupta. *Efficient Support for Complex Numbers in Java*. Proc. of the ACM 1999 Java Grande Conference, San Francisco, June 1999, 10 pp., http://www.cs.ucsb.edu/conferences/java99/program.html.