Implementation of the NAS Parallel Benchmarks in Java

Michael A. Frumkin, Matthew Schultz, Haoqiang Jin, and Jerry Yan
NASA Advanced Supercomputing (NAS) Division
NASA Ames Research Center, Moffett Field, CA 94035-1000
{frumkin,hjin,yan}@nas.nasa.gov

NAS-02-009

Abstract

Several features make Java an attractive choice for High Performance Computing (HPC). In order to gauge the applicability of Java to Computational Fluid Dynamics (CFD), we have implemented the 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 position 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 an interest in the HPC community to evaluate Java on computationally intensive problems [11]. Java threads, RMI, and networking capabilities position Java well for programming on Shared Memory Parallel (SMP) computers and on computational grids. On the other hand issues of safety, lack of light weight objects, intermediate byte code interpretation, and array access overheads create challenges in achieving high performance for Java codes. The challenges are being addressed by work on implementation of efficient Java compilers [12, 13] and by extending Java with classes implementing the data types used in HPC [12].

In this paper, we describe an implementation of the NAS Parallel Benchmarks (NPB) [1] in Java. The benchmark suite is accepted by the HPC community as an instrument for evaluating performance of parallel computers, compilers, and tools. We quote from [10] "Parallel Java versions of Linpack and NAS Parallel Benchmarks would be particularly interesting". The implementation of the NPB in Java builds a base for tracking the progress of Java technology and for evaluating Java as a choice for programming scientific applications, and for identifying the areas where improvement in Java compilers would make the strongest impact on the performance of scientific codes written in Java.

Our implementation of the NPB in Java is derived from the optimized NPB2.3-serial version [9] written in Fortran (except IS, written in C). The NPB2.3-serial version was previously used by us for the development of the HPF [5] and OpenMP [9] versions of the NPB. We start with an evaluation of Fortran to Java conversion options by comparing performance of basic Computational Fluid Dynamics (CFD) operations. The most efficient options are then used to translate the Fortran to Java. We then parallelize the resulting Java code by using Java threads and the master-workers load distribution model. Finally, we profile the benchmarks and analyze the performance on five different machines: IBM p690, SGI Origin2000, SUN Enterprise10000, Intel Pentium-III based PC, and Apple G4 Xserve. The implementation is available as the NPB3.0-JAV package from www.nas.nasa.gov.

2 The NAS Parallel Benchmarks

The 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. The NPB suite consists of five kernels and three simulated CFD applications. The five kernels represent computational cores of numerical methods routinely used in CFD applications. The simulated CFD applications mimic data traffic and computations found in full CFD codes.

An algorithmic description of the benchmarks (pencil and paper specification) was given in [1] and is referred to as NPB-1. A source code implementation of most benchmarks (NPB-2) was described in [2]. The latest release NPB-2.3 contains MPI source code for all the 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. Recently, OpenMP [9] and HPF [5] versions of the benchmarks have been developed. These benchmarks were released, along with an optimized serial version, as a separate package called Programming Baselines for NPB (PBN). For completeness of this paper, we outline the seven benchmarks 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 **B**lock **T**ridiagonal system of 5x5 blocks is solved sequentially along each dimension.

SP is a simulated CFD application. It differs from BT in the factorization of the discrete Navier-Stokes operator. It employes the Beam-Warming approximate factorization that decouples the x, y, and z dimensions. The resulting system of **S**calar **P**entadiagonal linear equations is solved sequentially along each dimension.

LU is also a simulated CFD application. It uses the symmetric successive over-relaxation (SSOR) method to solve the discrete Navier-Stokes equations by splitting it into block **L**ower and **U**pper triangular systems.

FT contains the computational kernel of a 3-D Fast **F**ourier **T**ransform (FFT). Each FT iteration performs three sets of one-dimensional FFTs, one sets for each dimension.

MG uses a V-cycle **M**ulti **G**rid 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 coarsest and the finest grids. It tests both short and long distance data movement.

CG uses a **C**onjugate **G**radient method to compute approximations to the smallest eigenvalues of a sparse unstructured matrix. This kernel tests unstructured computations and communications by manipulating a diagonally dominant matrix with randomly generated locations of entries.

IS performs sorting of integer keys using a linear-time **I**nteger **S**orting algorithm based on computation of the key histogram. IS is the only benchmark written in C.

3 Fortran to Java Translation

Java is a more expressive language than Fortran. This eases the task of translating Fortran code to Java. However, matching the performance of Fortran versions is still a challenge. In the *literal translation*, the procedural structure of the application is kept intact, the 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. The 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 the corresponding data is smaller than in the object oriented translation. On the other hand, the object oriented translation results in a better structured code and allows advising the compiler of special treatment of particular classes, for example, using semantic expansion [12, 13]. Since we are interested in high performance code we chose the literal translation.

In order to compare efficiency of different options in 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. The relative performance of different implementations of the basic operations gives us a guide for Fortran-to-Java translation. As the basic operations we chose the operations that we have used to build the HPF performance model [6]:

- 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 a compact 3x3x3 stencil as in the smoothing operator in MG);
- a matrix vector multiplication of a 3-D array of 5x5 matrices by a 3-D array of 5-D vectors; (a routine CFD operation);
- a reduction sum of 4-D array elements.

We implemented these operations in two ways: by using linearized arrays and by preserving the number of array dimensions. The version that preserves the number of array dimension was 1.5-2 times slower than the linearized version as on the SGI Origin2000 (Java 1.1.8) as on the Sun Enterprise10000 (Java 1.1.3). So we decided to translate Fortran arrays into linearized Java arrays; hence, we present the profiling data for the linearized translation only. The performance of the serial and multithreaded implementations are compared with the Fortran implementation. The results on the SGI Origin2000 are summarized in Table 1.

Table 1. The execution times in seconds of the basic CFD operations on the SGI Origin2000; the grid size 81x81x100.

	f77	Java 1.1.8						
			Number of Threads					
Operation	Serial		1	2	4	8	16	32
1. Assignment (10 iterations)	0.327	1.087	1.256	0.605	0.343	0.264	0.201	0.140
2. First Order Stencil	0.045	0.373	0.375	0.200	0.106	0.079	0.055	0.061
3. Second Order Stencil	0.046	0.571	0.572	0.289	0.171	0.109	0.082	0.072
4. Matrix vector multiplication	0.571	4.928	6.178	3.182	1.896	1.033	0.634	0.588
5. Reduction Sum	0.079	0.389	0.392	0.201	0.148	0.087	0.063	0.072

We can offer some conclusions from the profiling data:

- Java serial code is a factor of 3.3 (Assignment) to 12.4 (Second Order Stencil) slower than the corresponding Fortran operations;
- Thread overhead (serial column versus 1 thread column) contributes no more than 20% to the execution time;
- The speedup with 16 threads is around 7 for the computationally expensive operations (2,3 and 4) and is around 5-6 for less intensive operations (1 and 5).

For a more detailed analysis of the basic operations we used an SGI profiling tool called `perfex`. The `perfex` uses 32 hardware counters to count issued/graduated integer and floating point instructions, load/stores, and primary/secondary cache misses etc. The profiling with perfex shows that the Java/Fortran performance ratio correlates well with the ratio of the total number of executed instructions in these two codes. Also, the Java code executes twice as many floating point instructions as the Fortran code, confirming that the Just-In-Time (JIT) compiler does not use the "madd" instruction since that is not compatible with the Java rounding error model [11].

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

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

into the 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 then 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 via 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, and the `call` statement. The semiautomatic translation allowed us to translate about 70% of Fortran code to Java. In general, any translation requires parsing the Fortran code and translating the parse tree to a Java equivalent, for example, for labeled DO loops, common, format, and IO statements.

We structured the code in the following way. Each benchmark has a 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 for each Fortran subroutine, including main. There is one worker class for each parallelizable Fortran subroutine (see the discussion in the next section). The main class has two additional methods: `runBenchmark()` executed in the serial mode and `run()` executed in the parallel mode. The `runBenchmark()` method calls all methods exactly in the same sequence as in the original Fortran code. The `run()` method is used to start threads in the parallel mode. The commonly used functions `Timer`, `Random`, and `PrintResults` are implemented as separate classes and are imported into each benchmark. All the benchmarks are packaged in the NPB3˙0˙JAV package.

4 Using Java Threads for Parallelization

A significant appeal of Java for parallel computing stems from the presence of threads as part of the Java language. On a shared memory multiprocessor, 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 among the threads. Conceptually, Java threads are close to OpenMP threads, so we used the OpenMP version of the benchmarks [9] as a prototype for the multithreading.

The base class (and, 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 is designated to be the master that controls the synchronization of the worker objects. The workers are switched between blocked and runnable states with `wait()` and `notify()` methods of the `Thread` class.

In all benchmarks, except MG, the work per thread is the same in all iterations. Hence, in these benchmarks, the initialization of the threads and partitioning the work among them is performed in the main class. The partitioning is accomplished by specifying the starting and ending iterations of the outer loop for each worker. The master thread dispatches the job to each worker, starts the workers, and then waits until all workers are finished (see 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 assigned 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 worker's step method. In MG, the load per thread depends on the size of the grid used in this iteration. Hence, in MG, each thread uses the `GetWork()` method to obtain the loop boundaries before it performs the step method.

This model of thread synchronization is applicable only to independent workers: each worker processes the job dispatched by the master, independent of other workers. This is the case for all the benchmarks except LU where there is a pipelined dependence between workers. We implemented the pipelined computations with a 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 among themselves with the job relay, as shown in 2.

5 Performance and Scalability

We have tested the benchmarks on the classes S, W, and A; the performance is shown for class A as the largest of the tested classes. The tests were performed on three shared memory machines: IBM p690 (1.3 GHz, 32

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){}
}
GetWork(); //MG only
step();
done=true;
synchronized(master){master.notify();
```

Figure 1. The master-worker thread synchronization.

```
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 pipelining computations. Here id is the thread number.

processors), SGI Origin2000 (250 MHz, 32 processors), and SUN Enterprise 10000 (333 MHz, 16 processors). On the IBM p690 we used Java 1.3.0, on the SGI Origin we used Java 1.1.8, and on the SUN Enterprise we used Java 1.1.3. On the SUN Enterprise we also tested Java 1.2.2, but its scalability was significantly worse than that of Java 1.1.3. The performance results are summarized in Tables 2-4. For comparison, we include the Fortran-OpenMP results in Tables 2 and 3.

Also for reference we ran the benchmarks on a Linux PC (933 MHz, 2 PIII processors, Java 1.3.0), Table 5 and on 1 node of Apple Xserve (1GHz, 2 G4 processors, Java 1.3.1), Table 6.

5.1 Comparison to Fortran Performance

We offer the following conclusions from the performance results. There are two groups: benchmarks BT, SP, LU, FT, and MG 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 the interval 8.3-10.8, which is within the 8.2-12.5 interval for the computationally intensive basic CFD operations (operations 2-4 from Table 1), indicating that our implementation of the benchmarks adds little performance overhead to the overhead of the basic operations.

Table 2. Benchmark times in seconds on IBM p690 (1.3 GHz, 32 processors).

		Number of Threads					
	Serial	1	2	4	8	16	32
BT.A Java1.3.0	511.5	614.4	307.1	160.2	80.8	41.5	22.4
BT.A f77-OpenMP	161.4	254.1	129.5	66.1	34.8	17.6	10.5
SP.A Java1.3.0	407.1	427.5	214.2	111.1	58.7	30.8	33.2
SP.A f77-OpenMP	142.6	141.2	72.3	37.8	18.7	10.2	6.2
LU.A Java1.3.0	615.9	645.8	322.5	168.5	90.5	46.2	28.5
LU.A f77-OpenMP	144.0	145.9	70.0	32.9	16.7	8.9	6.0
FT.A Java1.3.0	54.4	46.9	28.8	15.0	8.6	5.61	4.83
FT.A f77-OpenMP	10.8	11.0	5.5	2.7	1.4	0.76	0.55
IS.A Java1.3.0	1.60	1.70	1.04	0.83	0.76	0.79	2.50
IS.A C-OpenMP	1.36	1.87	1.02	0.55	0.35	0.27	0.40
CG.A Java1.3.0	8.75	8.16	4.55	2.44	1.50	1.37	1.79
CG.A f77-OpenMP	6.22	6.21	3.13	1.64	0.83	0.41	0.46
MG.A Java1.3.0	14.55	14.44	7.76	4.15	2.39	1.80	1.70
MG.A f77-OpenMP	6.95	6.84	3.34	1.56	0.86	0.55	0.44

For the second group, the Java/Fortran execution time ratio is within the 3.11-7.2 interval. The separation into two groups may be explained by the fact that the f77 compiler optimizes regular-stride computations much better than the tested Java compilers.

The benchmarks working on structured grids heavily involve the basic CFD operations and any performance improvement of the basic operations would directly affect performance of the benchmarks. Such improvement can be achieved in three ways. First, JIT needs to reduce the ratio of Java/Fortran instructions (which for Java 1.1.8 is a factor of 10) for executing the basic operations. Second, the Java rounding error model should allow the "madd" instruction to be used. Third, in all benchmarks working on structured grids, the array sizes and loop bounds are constants, and simple compiler optimization should be able to lift bounds checking out of the loop [11] without compromising code safety.

Our performance results apparently are in sharp contrast to the results reported by the Java Grande Benchmarking Group [3]. In that paper it was reported that on almost all Java Grande Benchmarks, the performance of a Java version is within a factor of 2 of the corresponding C or Fortran versions. To resolve the discrepancy in the performance we obtained the jgf˜2.0 from the `www.epcc.ed.ac.uk/javagrande` website. Since the Fortran version was not available on the website we literally translated the Java version to Fortran and ran both versions on multiple platforms. The results are summarized in Table 7. We have also included results of the LINPACK version of the LU decomposition. From the table we can conclude that the algorithm used in `lufact` benchmark performs very poorly relative to LINPACK. The reason for this is that `lufact` is based on BLAS1, having poor cache reuse. As a result, the computations always wait for data (cache misses), which obscures the performance comparison between the Java and Fortran versions. Note that our Assignment base operation exhibits about the same Java/Fortran performance ratio as the `lufact` benchmark.

5.2 Scalability of Multithreaded Java Codes

Singlethreaded Java benchmarks sometimes run faster than the serial versions. That can be explained by the fact that in the singlethreaded version the data layout is more cache friendly. Overall the multithreading introduces an overhead of about 10%-20%. The speedup of BT, SP, and LU with 16 threads is in the range of 6-12 (efficiency 0.38-0.75). The low efficiency of FT on SUN Enterprise is explained by the inability of the JVM to use more than 4 processors to run applications requiring significant amounts of memory (FT.A uses about 350 MB). An artificial increase in the memory use for other benchmarks also resulted in a drop of scalability. 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 a thread relay-racing mechanism. The low scalability of IS was expected since

Table 3. **Benchmark times in seconds on SGI Origin2000 (250 MHz, 32 processors).**

		Number of Threads					
	Serial	1	2	4	8	9	16
BT.A Java 1.1.8	9136.3	8332.5	4806.0	2645.7	1413.7	1278.0	838.4
BT.A f77-OpenMP	1028.0	983.6	519.5	275.4	143.7	133.0	81.4
SP.A Java 1.1.8	7137.4	7111.0	3789.8	2333.8	1705.2	1581.2	1188.2
SP.A f77-OpenMP	944.7	850.8	504.5	259.9	147.6	133.0	88.5
LU.A Java 1.1.8	9686.8	9967.4	5600.9	3475.8	2247.8	-	1502.2
LU.A f77-OpenMP	1104.8	926.9	439.7	236.4	132.5	121.1	75.72
FT.A Java 1.1.8	656.0	630.8	361.1	174.9	110.6	-	63.8
FT.A f77-OpenMP	82.3	74.8	41.1	21.1	11.7	10.9	7.1
IS.A Java 1.1.8	10.7	12.7	15.0	17.9	11.8	-	17.9
IS.A C-OpenMP	4.9	4.9	2.7	1.5	1.0	-	0.9
CG.A Java 1.1.8	105.0	112.4	114.2	53.9	52.6	-	23.1
CG.A f77-OpenMP	39.7	35.6	21.8	10.2	3.6	3.2	2.7
MG.A Java 1.1.8	254.0	263.7	189.3	108.4	70.8	-	45.0
MG.A f77-OpenMP	36.4	36.8	23.0	12.7	7.7	6.4	4.1

Table 4. **Benchmark times in seconds on SUN Enterprise10000 (333 MHz, 16 processors).**

		Number of Threads						
	Serial	1	2	4	8	9	12	16
BT.A Java1.1.3	13609.5	14671.3	7381.7	3846.3	2305.0	2042.7	1782.7	1762.2
SP.A Java1.1.3	10235.8	11108.1	5692.9	3409.3	2095.5	1899.1	1862.1	1671.2
LU.A Java1.1.3	12344.5	13578.9	6843.3	3765.7	2077.3	1892.7	1730.2	1745.4
FT.A Java1.1.3	1104.6	1318.8	674.7	384.2	342.7	-	353.4	363.3
IS.A Java1.1.3	22.9	29.4	15.7	9.0	8.4	-	8.9	13.6
CG.A Java1.1.3	203.8	215.3	111.6	69.0	47.6	-	40.8	36.4
MG.A Java1.2.2	438.9	494.7	244.8	138.5	87.1	-	72.6	68.7

the amount of work performed by each thread is small relative to other benchmarks, hence, the data movement overheads eclipse the gain in processing time.

Our tests of CG benchmark on the SGI Orgin2000 showed virtually no performance gain until 8 processors were used; (similar observations are valid for IS). Even with a large number of threads (10-16), only a few processors were used(2-4). To investigate this problem, we used "top -T" which allows monitoring the individual `Posix` threads of an application. With this utility, we found that the JVM seemed to be ignoring our thread creation and running all the threads in one or two Posix threads. The fact that all the other benchmarks ran each thread in a separate Posix thread suggested that the problem was peculiar to CG. CG's work load is much smaller than the work load of the computationally intensive benchmarks. Based on this, we hypothesized that the JVM was attempting to optimize CPU usage by running the threads serially on a few processors instead of using one processor per thread. In order to test this, we put an initialization section into the benchmark which performed a large number of floating point operations in each thread, in the hope that the JVM would create more Posix threads to handle the high CPU load. With this change in the code, the JVM created all threads for executing the initialization section. When the actual computations did start, JVM used a separate CPU for each thread. As the number of threads increased, the work load on each CPU decreased somewhat. However, by initializing the thread load, we were able to get a visible speedup of CG see Table 3. On the Linux PIII PC we did not obtain any speedup when using 2 threads. The reason for this will be farther investigated.

Table 5. Benchmark times in seconds on Linux PC (933 MHz, 2 PIII processors).

		Number of Threads	
Java1.3.0	Serial	1	2
BT.A	8007.8	8007.7	8083.2
SP.A	3543.9	4198.7	4201.9
LU.A	5887.9	7151.7	7140.7
FT.A	411.0	493.0	494.4
IS.A	9.1	9.4	9.8
CG.A	116.8	75.8	77.0
MG.A	195.0	170.3	188.2

Table 6. Benchmark times in seconds on Apple Xserve (1 GHz, 2 G4 processors).

		Number of Threads	
Java1.3.0	Serial	1	2
BT.A	2043.15	2120.87	1185.97
SP.A	1377.56	1487.30	845.91
LU.A	17779.24	19075.83	9883.85
FT.A	179.71	161.31	95.93
IS.A	7.08	7.59	6.00
CG.A	51.62	48.69	28.08
MG.A	59.94	60.12	36.07

Table 7. Java Grande LU benchmark [4]. The Fortran version was directly derived from `lufact`. The performance of the LINPACK version of the LU decomposition (DGETRF, based on MMULT, and having good cache reuse) is shown for reference. The execution time is in seconds. (The classes A,B and C employ 500x500, 1000x1000 and 2000x2000 matrices respectively).

	Java			f77			Linpack		
Machine/Platform	A	B	C	A	B	C	A	B	C
SUN UltraSparc/Java 1.4.0	3.13	27.78	250.3	0.36	8.11	104.0	0.423	3.448	29.93
SGI Origin2000/Java 1.1.8	3.05	28.10	266.7	0.70	7.94	86.3	0.207	1.710	13.78
Sun E10000/Java 1.1.3	3.86	48.92	512.0	1.41	29.90	395.6	0.522	4.411	48.55
IBM POWER4/Java 1.3.0	0.27	2.60	21.3	0.17	2.19	17.6	0.031	0.237	1.74

6 Related Work

In our implementation we parallelized the NAS Parallel Benchmarks using Java threads. The University of Westminster'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` [8]. 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) [10], along with many other benchmarks in order to test Java's suitability for grand challenge applications.

The Java Grande Forum have developed a set of benchmarks [4] reflecting various computationally intensive applications which likely will benefit from use of Java. The performance results reported in [3] relative to C and Fortran are significantly more favorable to Java than ours.

7 Conclusions

Although the performance of the implemented NAS Parallel Benchmarks in Java is lagging far behind Fortran and C at this time, by using the performance enhancing methods detailed in [11, 13], the serial performance can be improved to near Fortran-like performance. From our performance results it follows that the IBM Java compiler is the leader in this direction. Efficiency of parallelization with threads is about 0.5 for up to 16 threads and is lower than the efficiency of parallelization with OpenMP, MPI, and HPF on SGI and SUN machines. However, on the IBM machine, the scalablity of the Java code is as good as that of OpenMP, and in average the performance of the Java code is within a factor of 3 of that of Fortran.

With several groups working on MPI and OpenMP for Java, improvements in parallel performance and scalability seem likely as well. The attraction of Java as a numerically intensive applications language is primarily driven by its ease of use, universal portability, and high expressiveness which, in particular, allows expressing parallelism. If Java code 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 programming environment for HPC applications. The NPB3˙0˙JAV package is avialable from www.nas.nasa.gov/Software/NPB.

Acknowledgment. The authors appreciate help of IBM staff members Charles Grassl and Luiz DeRose in understanding of the p690 architecture, and of Steve Sullivan from Apple for providing access to the Apple Xserve.

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] J.M. Bull, L.A. Smith, L. Pottage, R. Freeman. *Benchmarking Java against C and Fortran for Scientific Applications.* Joint ACM Java Grande - ISCOPE 2001 Conference, Palo Alto, CA, pp. 975-105.

[4] *Java Grande Benchmarks.* http://www.epcc.ed.ac.uk/computing/research˙activities/java˙grande/index˙1.html.

[5] 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.

[6] 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.

[7] *The GNU Compiler for the Java™ Programming Language.* http://sourceware.cygnus.com/java.

[8] 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.

[9] H. Jin, M. Frumkin, J. Yan. *The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance.* NAS Technical Report RNR-99-011, NASA Ames Research Center, Moffett Field, CA, 1999, http://www.nas.nasa.gov/Software/NPB/.

[10] 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.

[11] 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.

[12] 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.

[13] J. E. Moreira, S. P. Midkiff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence. *Java programming for high-performance numerical computing.* IBM Systems Journal, Vol. 39, n. 1, 2000. http://www.research.ibm.com/journal/sj/391/moreira.html.