Benchmarking Parallel Java

Master’s Project Report

Asma’u Sani Mohammed

9/9/2013
Committee Chair: Prof. Alan Kaminsky
Reader: Prof. James Heliotis
Observer: Dr. Hans Peter Bischof
Department of Computer Science
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
ABSTRACT

Parallel computing has become popular amongst the scientific community and areas where large data computation is needed. The need for computationally efficient parallel middleware has become increasingly important as hardware architectures evolve to allow parallelism. My project examined the effectiveness of the Parallel Java API by implementing the OpenMP version of the NAS Parallel Benchmark (NPB) in Parallel Java. The translation from FORTRAN OpenMP to Parallel Java will be discussed in this report, as well as implementation and performance details in comparison with FORTRAN OpenMP. Benchmarking Parallel Java allows us to understand the performance and scalability of the API, in particular the SMP architecture. Four of the NPBs were implemented, namely the Conjugate Gradient, Block Tridiagonal, Scalar Pentadiagonal and Lower and Upper Triangular System. These benchmarks were run on a 48-core machine with varying thread numbers in order to calculate speedup, running time, and efficiency. Our results will give insight to areas within the Parallel Java directives that could potentially be improved to achieve better performance.
# Table of Contents

Abstract .......................................................................................................................... 1

1.0. Introduction ............................................................................................................. 3

2.0. The NAS Parallel Benchmark .................................................................................. 3

3.0. Brief Overview of the Parallel Java Construct ....................................................... 3
    3.1. Core Parallel Java Constructs ............................................................................ 4

4.0. Overview of Translating OpenMp to Parallel Java ................................................. 5

5.0. Benchmark Implementations and performance analysis ........................................ 5
    5.1. The CG Benchmark .......................................................................................... 5
        5.1.1. Problem Sizes ......................................................................................... 6
        5.1.2. Implementation Details of CG ................................................................. 6
        5.1.4. Performance ............................................................................................. 11
    5.2. The BT Benchmark .......................................................................................... 14
        5.2.1. Problem size Definition ......................................................................... 15
        5.2.2. Changes made To Array Structure in Translating OpenMp to Java ............ 15
        5.2.3. The parallel Structure of The BT Benchmark ........................................... 16
        5.2.4. Performance of BT Benchmark ............................................................... 17
    5.3. The SP Benchmark .......................................................................................... 19
        5.3.1. Problem Sizes ......................................................................................... 19
        5.3.2. Translating the SP Benchmark from OpenMp to Java .............................. 20
        5.3.3. Performance ............................................................................................. 23
        5.3.3.1. Array access optimization ................................................................. 23
    5.4. The LU Benchmark .......................................................................................... 25
        5.4.1. Problem sizes .......................................................................................... 26
        5.4.2. Translation from OpenMp to Parallel Java .............................................. 26
        5.4.3. Performance ............................................................................................. 28

6.0. Related Work ......................................................................................................... 30

7.0. Lessons Learned .................................................................................................... 31
    7.1. Preliminary Work .............................................................................................. 31
    7.2. Parallel Java And OpenMp ................................................................................. 31
        7.2.1. Array Access .......................................................................................... 31
        7.2.2. Thread Startup Overhead and Nested Parallelism .................................... 31
        7.2.3. End of Loop Synchronization ................................................................. 31

8.0. Conclusion and Future Work .................................................................................. 32

9.0. References ............................................................................................................ 33
1.0. INTRODUCTION

The level of parallelism supported by parallel computers varies significantly, from symmetric multiprocessor, distributed computing and multi-core computing. Various libraries exist for programming parallel computers. Each library or API makes assumptions based on the underlying computer architecture. This wide availability of computing systems and difference in computer architecture makes it a challenge to objectively assess the performance of these parallel middleware libraries. Parallel benchmark codes such as the SPEC HPC 2002 Benchmarks, the OpenMP Benchmarks, the Pallas MPI Benchmarks, and NPB; have been developed to aid in comparing the performance of parallel middleware across different architectures. NPB is an Open source benchmark developed by NASA’s Advanced Supercomputing (NAS) division. The NPB suite is among the most highly used open source benchmarks out there. My project involved porting the OpenMP implementation of the NPB 3 specification [3] to Parallel Java [8]. These Benchmarks are written in FORTRAN, which I translated to Parallel Java. The report was generated for comparison between the performance of the OpenMP and Parallel Java versions. The performance was measured based on execution time. In order to make an objective comparison between the OpenMP and Parallel Java benchmark implementations, the OpenMP implementation was run on the same 48-core machine that the Parallel Java benchmarks were run on. Both benchmark implementations were tested on 1, 2, 3, 4, 6, 8, 12, 16, 24, 32 and 48 cores. Given that the number of CPUs and the data size cover a large range of values (1 to 48 CPUs), we decided to set the intervals between the numbers of processors to increase exponentially in order to give an even coverage of the data.

2.0. THE NAS PARALLEL BENCHMARK

NPB was developed to aid the development of advanced super computers capable of performing complex computations orders of magnitude faster than conventional supercomputers in the early 90s [1]. It was an attempt to bridge the gap between hardware advances and advances made in parallel computing. The initial benchmarks were designed as “pencil and paper” descriptions of algorithms. These algorithms had enough details for one to implement it but general enough to allow a certain level of flexibility based on different languages. The benchmarks have been implemented in different API’s including MPI, OpenMP, FORTRAN and Java. The versions range from NPB-1 to NPB-3.2 with NPB-3.2 being the latest version that carries the Java and OpenMP implementations.

NPB-3 has a total of eleven benchmarks, seven of which are implemented in the OpenMP version. Three of the seven, namely, Block Tridiagonal (BT), Scalar Pentdiagonal (SP) and Lower and Upper triangular systems (LU) are simulated Computational Fluid Dynamics (CFD) applications that solve a synthetic system of nonlinear partial differential equations (PDE) using three different algorithms. The MultiGrid (MG) solves a three-dimensional discrete Poisson equation using the V-cycle multigrid method. The Fast Fourier Transform (FT) uses the Fast Fourier Transform to solve a 3-D partial differential equation. Embarrassingly Parallel (EP) generates pairs of Gaussian random variates according to a specific scheme. The final of the seven is the Conjugate Gradient (CG), which estimates the smallest eigenvalue of a matrix using the conjugate gradient method as a subroutine for solving systems of linear equations.

3.0. BRIEF OVERVIEW OF THE PARALLEL JAVA CONSTRUCT

Parallel Java is a Java-based API that defines constructs, which allow a programmer to write sections of their code to be executed in parallel. The API supports Shared Memory Programming (SMP) as well as Cluster Parallel
In SMP parallel programs, data are stored within one memory area therefore the programmer would have to ensure that the right constructs are called to avoid race conditions and data synchronization issues. The benchmarks discussed in this document are implemented using the SMP model therefore this overview will focus on defining the Parallel Java constructs associated with the SMP programming model.

### 3.1. **Core Parallel Java Constructs**

As an example of a block of code defined to be run in parallel using the Parallel Java core constructs, Figure i shows a piece of code from the Conjugate Gradient Benchmark. This piece of code is aimed at giving context as we discuss the different constructs involved in writing an SMP parallel block in Parallel Java. The code serves only as an example of the use of the Parallel Java construct. Details of the Conjugate Gradient Benchmark will be discussed in later sections.

```java
team.execute(new ParallelRegion(){
    public void run() throws Exception{
        execute(1, lastcol-firstcol-1, new IntegerForLoop(){
            double d2 = 0.0;
            public void run(int first, int last){
                for(int j = first; j<=last; j++){
                    d2 += p[j]*q[j];
                }
            }
            public void finish(){
                // adding up all our collapsed sums
                sum2.addAndGet(d2);
            }
        });
    }
});
```

**Figure i. Parallel Java Construct**

**ParallelTeam:** The parallel team provides a team of threads for executing a block of code in parallel. We can specify the number of threads to be used in a parallel team and if nothing is defined the default takes the maximum number of threads available to the environment.

**ParallelRegion:** In order to define a certain part of code as an area to be executed by multiple threads, we define an instance of a parallelRegion. The `execute` clause that follows allows us to set the boundaries for which work is divided and shared amongst the participating threads in the parallel team. The call made to the execute method is done by the main thread however each thread within the parallel team calls the run method to start execution of their tasks. The `finish` method is used for operations that occur after the threads are done from the run method; it is called by the main thread and so serves as a synchronization mechanism within the threads.

**IntegerForLoop:** This describes a parallel loop that will be executed inside the parallel region. The IntegerForLoop class inherits from the ParallelForLoop base class with a loop index of type `int`. Scheduling within the threads is done using an IntegerSchedule defined within the IntegerForLoop.
4.0. Overview of Translating OpenMP to Parallel Java

This section only gives a brief overview of the initial work done in this project. The intricate details of the implementations will be discussed in later sections within the document as we deep dive into each benchmark.

The closest in architecture and SMP parallel construct amongst the implementations of the NAS Parallel Benchmarks is the OpenMP FORTRAN implementation. Therefore in order to get better understanding of the algorithms involved and possible optimizations, the OpenMP Fortran benchmarks were studied thoroughly and a comparison done on the performance difference.

The first step taken was in understanding the details of the algorithms behind the benchmarks. The Scalar Pentadiagonal (SP), Block Tridiagonal (BT) and Lower and Upper triangular systems (LU) are CFD codes derived from aerophysics applications [3]. These three benchmark algorithms, amongst the four implemented, took the most amount of effort to decipher. The next steps taken were in understanding FORTRAN 77 and OpenMP parallel constructs. This involved looking into some basic FORTRAN 77 constructs used in the benchmarks such as multi-dimensional arrays, implicit and explicit constructs variable declaration and initialization, and do-loops. The basic OpenMP directives are not large in number and as a result were relatively easy to pick up and follow.

Before translating the code from FORTRAN to Parallel Java, the OpenMP benchmarks were run using the GNU debugger. This gave me the ability to step through the benchmark implementations therefore fortifying my understanding of the FORTRAN 77 library as well as OpenMP parallel construct.

The Parallel Java implementation tried to keep the imperative structure of the OpenMP implementations. However, as will be discussed in later sections, a few changes were made to the procedural structure in order to incorporate the object-oriented nature of Parallel Java. The naming of objects and variables were kept the same however in the Parallel Java implementation.

In the translation of multi-dimensional arrays from FORTRAN to Java in the SP, BT and LU benchmarks, the use of linearized arrays was adopted as opposed to multi-dimensional arrays in Java. This will be discussed in more detail in section 6.2 when we analyze the BT benchmark implementation.

5.0. Benchmark Implementations and Performance Analysis

This section will take a detailed look into the implementation of the CG, BT, SP and LU benchmarks. The performance of the Parallel Java implementation will be compared to that of OpenMP in this section as well. All four benchmarks were run on a 48 core SMP Ubuntu 4.4.3 2100MHz system having a per processor cache size of 64 KB.

5.1. The CG Benchmark

The CG Benchmark uses a conjugate gradient method to find the largest eigenvalue of a symmetric positive definite sparse matrix. The conjugate gradient method is an iterative approach to solving large partial differential equations where we attempt to find the solution to the linear equation.
As an iterative method, we select different conjugate vectors to use in obtaining an approximation of the solution to our partial differential equation. Each approximation should fall within the accuracy specified by epsilon, ε. ε is set at “1.0E-10” for all problem sizes. There are two different problem sizes reported in this section, the class A and the class B. These are the two larger problem sizes that give a better performance comparison. Section 6.1.1 describes the problem sizes in more details.

5.1.1. PROBLEM SIZES
Each problem size defines six variables that make up the complexity and size of the benchmark. The first variable “na” defines the size of the sparse matrix A. “nonzer” defines the number of non-zeros, while the number of outer iterations (Eigen value estimates) is defined by “niter”. The Eigen value shift, λ, is defined as “shift” and “zeta_verify_value” defines the value of the approximate solution that is used in verifying the computed approximate solution. Table 1.0 shows the two classes A and B, and their values for each of the variables mentioned.

<table>
<thead>
<tr>
<th>Class A</th>
<th>Class B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Na = 14000</td>
<td>Na = 75000</td>
</tr>
<tr>
<td>Nonzer = 11</td>
<td>Nonzer = 13</td>
</tr>
<tr>
<td>Shift = 20</td>
<td>Shift = 60</td>
</tr>
<tr>
<td>Niter = 15</td>
<td>Niter = 75</td>
</tr>
<tr>
<td>Rcond = 0.1</td>
<td>Rcond = 0.1</td>
</tr>
<tr>
<td>zeta_verify_value = 17.130235054029</td>
<td>zeta_verify_value = 22.712745482631</td>
</tr>
</tbody>
</table>

5.1.2. IMPLEMENTATION DETAILS OF CG
In finding a solution to equation 1.0 the Conjugate Gradient algorithm first makes a call to the makea routine. makea is responsible for generating the sparse matrix A denoted as small letter “a”. Next we call the conjugate gradient routine once, consisting of 7 main steps, in order to initialize all code and data page tables. Then we compute zeta and find the norm of z. below is a highlight of the 7 steps involved in the conjugate gradient routine:

- compute q where q = A.p
- compute p.q
- compute alpha = rho / (p.q)
- compute z = z + alpha*p and r = r - alpha*q
- compute rho = r.r by first summing squares of r elements locally
- compute p = r + beta*p
- and then Compute residual norm explicitly: \[ ||r|| = ||x - A.z|| \]

In converting these 7 steps from OpenMP FORTRAN to Parallel Java, OpenMP parallel constructs were translated to their Parallel Java equivalent. An analysis of the two implementations will be discussed in the sub-sections that follow.
5.1.3. The Conjugate Gradient Loop

5.1.3.1. Computing $q=A.p$

Figure 1.0 shows the implementations of the OpenMP and Parallel Java loops for computing $q=A.p$. The OpenMp syntax for declaring a parallel region is given in line 1 of the OpenMp code snippet. The clause “private” refers to variables that are local to each thread participating in the parallel loop (denoted by !$omp parallel) while the “shared” clause refers to variables that are shared amongst the threads. “!$omp do” declares the start of the parallel loop. The number of iterations from 1 to lastrow-firstrow+1 is divided among the threads participating in the team.

The equivalent Parallel Java declaration is shown below this. To declare a block of code as a parallel region, we create a parallel Team by calling on the ParallelTeam class and then within the execute method we declare a parallelRegion. The Parallel Java equivalent of the OpenMp “private” clause for declaring per thread variables can be seen on line 5 and 6 of the Parallel Java implementation. This is achieved by declaring the interval 1 and lastrow-firstrow+1 within the execute clause (line 4 of the Parallel Java implementation) right before we create an instance of the IntegerForLoop which would be responsible for scheduling the task accordingly.

---

**OpenMP implementation**

1. !$omp parallel default(shared) private(j,k,sum,alpha,beta)
2. !$omp& shared(d,rho0,rho)
3. !$omp do
4. do $j=1,lastrow-firstrow+1$
5. $sum = 0.d0$
6. do $k=rowstr(j),rowstr(j+1)-1$
7. $sum = sum + a(k)*p[colidx(k)]$
8. enddo
9. $q(j) = sum$
10. Enddo

**Parallel Java implementation**

1. try{
2. new ParallelTeam().execute(new ParallelRegion(){
3. public void run() throws Exception{
4. execute(1, lastrow-firstrow+1, new IntegerForLoop(){
5. int j, k;
6. double sum;
7. public void run(int first, int last){
8. for($j = first; j<=last; j++){  
9. $sum = 0.0;$
10. for( $k = rowstr[j]; k<=rowstr[j+1]-1; k++$){
11. $sum++ a[k]*p[colidx[k]];$
12. }
13. $q[j] = sum;$
14. }}}}
5.1.3.2. **Compute residual norm explicitly:**

In order to compute the residual norm “∥r∥ = ∥x - A.z∥” first we form the partition submatrix “A.z”, which we call “r”. Then we compute the norm by subtracting r from x and collapsing all the sums into one sum. In order to compute the sum we need to combine the local sums computed by each thread into one solution. This is done through a process called reduction. OpenMp uses the “do reduction” clause to indicate a computation that needs to be done on a list of variables that are passed into the reduction routine. On line 12 of Figure 1.1 of the OpenMp implementation, the reduction variable specified is sum and the operation to be performed is addition. Each thread gets assigned a private variable sum. Once the reduction is complete; the reduced sum is then applied to all the sum private variables belonging to the threads and also written to a global sum variable.

Reduction in Parallel Java is done a little differently. First we declare a variable of the shared variable type, in our case we have declared sum2 as a `SharedDouble`. This is the variable that will hold all the collapsed sums. The `addAndGet(double val)` method of the `SharedDouble` object, as is seen on line 37 of Figure 1.1 is responsible for the addition of the per thread sums. `addAndGet(double val)` adds the value “d2” (individual thread local sum) to the existing value of sum and returns the new value. There is a performance cost associated with reduction, which will be discussed in section 6.1.4 when we analyze performance of the CG benchmark.

**OpenMp implementation**

```
1.   sum = 0.0d0
2.   !$omp parallel default(shared) private(j,d)
3.   !$omp& shared(sum)
4.   !$omp do
5.      do j=1,lastrow-firstrow+1
6.         d = 0.d0
7.         do k=rowstr(j),rowstr(j+1)-1
8.            d = d + a(k)*z(colidx(k))
9.         enddo
10.    r(j) = d
11.   enddo
12.  !$omp do reduction(+:sum)
13.   do j=1, lastcol-firstcol+1
14.      d = x(j) - r(j)
15.      sum = sum + d*d
16.   enddo
17.  !$omp end parallel
```
Parallel Java Implementation

1. try{
2.  team.execute(new ParallelRegion()){
3.  public void run() throws Exception{
4.    execute(1, lastrow-firstcol+1, new IntegerForLoop{
5.      double d2;
6.    public void run(int first, int last){
7.      for(int j = first; j<=last; j++){
8.        d2 = 0.0;
9.        for(int k = rowstr[j]; k<=rowstr[j+1]-1; k++){
10.       d2+= a[k]*z[colidx[k]];
11.     }
12.     r[j] = d2;
13.   }
14.  }
15. });
16. }
17. });
18. }
19. catch(Exception e){
20.  e.printStackTrace();
21. }

22. try{
23.  team.execute(new ParallelRegion()){
24.    public void run() throws Exception{
25.      execute(1, lastrow-firstcol+1, new IntegerForLoop{
26.        double d2 = 0.0;
27.      public void run(int first, int last){
28.        for(int j = first; j<=last; j++){
29.         d2 += (x[j]-r[j]) * (x[j]-r[j]);
30.      }
31.    }
32.    public void finish(){
33.      // adding up all our collapsed d2 sums
34.      sum2.addAndGet(d2);
35.    }
36.  });
37.  }
38. }
39. }
40. catch(Exception e){
41.   e.printStackTrace();
42. }
43. }

Figure 1.1 Compute residual norm explicitly
5.1.3.3. Computing Zeta and the norm of z

Once we have computed and returned the sum derived from the conjugate gradient loop (that is, the Euclidean norm of the residual vector “r”), we need to compute zeta and the norm of z. Zeta is the sum of the Eigenvalue λ (shift) and the reciprocal of the value derived from summing up the product of the two vectors x and z (the solution to the linear equation Az=x). That is:

\[
\text{zeta} = \text{shift} + 1/(x \cdot z) \tag{2.0}
\]

In solving for zeta, we need to break the equation into two separate steps, one of which is to compute the value of (x.z) and the other to normalize z by computing (z.z). Once we have these two norms, we can then easily compute the value of zeta. Figure 1.2 shows the implementation of these two steps and how zeta is obtained. A few extra steps were taken in the Parallel Java implementation in order to solve for equation (2.0). Two variables, thread_norm1 and thread_norm2 were created for each thread. These two variables hold the per thread normalization of (x.z) and (z.z). The per-thread sums are then reduced in the finish function into the variables sum2 and sum3. The two sums, sum2 and sum3 are then copied into norm_temp1 and norm_temp2 respectively and zeta is calculated. This is shown in Figure 1.2 lines 26-31 of the Parallel Java implementation.

**OpenMP Implementation**

```cpp
1. norm_temp1 = 0.0d0
2. norm_temp2 = 0.0d0
3. !$omp parallel do default(shared) private(j)
4. !$omp& reduction(+:norm_temp1,norm_temp2)
5. do j=1, lastcol-firstcol+1
6.   norm_temp1 = norm_temp1 + x(j)*z(j)
7.   norm_temp2 = norm_temp2 + z(j)*z(j)
8. enddo
9. norm_temp2 = 1.0d0 / sqrt(norm_temp2)
10. zeta = shift + 1.0d0 / norm_temp1
```

**Parallel Java Implementation**

```java
1. norm_temp1 = 0.0;
2. norm_temp2 = 0.0;

3. try{
4.   team.execute(new ParallelRegion(){
5.     public void run() throws Exception{
6.         execute(1, lastcol-firstcol+1, new IntegerForLoop(){
7.             double thread_norm1 = 0.0;
8.             double thread_norm2 = 0.0;
9.             public void run(int first, int last){
10.                for(int j = first; j<=last; j++){
11.                   thread_norm1 += x[j]*z[j];
12.                   thread_norm2 += z[j]*z[j];
13.                }
14.            }
15.        }
16.    }
17. }
18. }
```

```
14. } 
15. public void finish(){
16.    sum2.addAndGet(thread_norm1);
17.    sum3.addAndGet(thread_norm2);
18. }
19. };
20. }
21. }
22. }
23. catch(Exception e){
24.   e.printStackTrace();
25. }

26. norm_temp1 = sum2.get();
27. norm_temp2 = sum3.get();
28. sum2.set(0); sum3.set(0);
29. norm_temp2 = 1.0/Math.sqrt(norm_temp2);
30. temp = norm_temp2;
31. zeta = shift + 1.0/norm_temp1;

Figure 1.2 computing zeta

5.1.4. PERFORMANCE
The total running time of the CG benchmark comprises of three sub running time computations. These times are; the time it takes to run the benchmark method, having the highest running time, followed by the running time for the initialization method and finally the time for computing the conjugate gradient method. This section will explain the performance of the CG benchmark by analyzing the different parallel sections within the benchmark method.

5.1.4.1. THE BENCHMARK METHOD
The Parallel Java running time of the benchmark method is considerably more than that of the OpenMp implementation. This could be in part as a result of the way the two libraries handle reduction.

In Parallel Java, reduction is done using a shared double, which uses a lock free atomic compare and set. The method addAndGet first takes a copy of the existing value of the shared double, and converts it to a long. Then it sums the new and old values and converts those to a long type as well. The new summed value is then returned after verifying that the summation happened correctly through an atomic compareAndSet method provided in the Java AtomicLong class. This process happens sequentially meaning that only one thread can go into the addAndGet method at a time.

The OpenMp approach to reduction varies from Parallel Java. At first each thread is assigned work and a local variable for the summation just as in Parallel Java. However, when the thread gets to the reduction construct, each thread will have a pointer to an array indexed by the thread_id. Each thread then puts its local sum into the array at an index equal to the thread id. Once all the threads have put their values in the array, the master thread then iterates through the array and sums up all the values.

This is important to note because what it means for performance is that the threads do not have to execute sequentially in order to add their local sum variables to the global sum, with OpenMp part of this happens in
parallel. The indexing of the array by thread_ids ensures that the operation is thread-safe and no variables will be overwritten as the threads write concurrently into the summation array. Although the actual addition of the values in the summation array itself happens using only the master thread, a part of the operation has been done in parallel, which improves performance as compared to the Parallel Java approach to reduction.

5.1.4.2. **The Initialization Step**

Initialization here involves calling the procedure to make the sparse matrix A, initializing the vector x to 1 and shifting the column index values. *makea* method is computed in part parallel and in part sequential as is in the OpenMp implementation. The part done in parallel involves initializing colidx to zero but the actual generation of the matrix "a" is done sequentially. Within the initialization step we also perform one sweep of computing zeta in which a reduction is done as explained in the section above. The cost to performance can be in the creation of a parallel block as compared to its creation in OpenMp. This would be discussed in more detail when we look at the performance of the SP Benchmark.

A summarization of the performance difference can be seen in table 1.1 where the graphs show a comparison between the performance of the OpenMp Benchmark and Parallel Java. The efficiency and EDSF are also shown which are defined as the following:

**Efficiency:** a value between 0 and 1 that defines how the speedup varies with increasing number of processors. This helps to determine how well the program is utilizing the processors in doing the programs work.

**EDSF:** A measurement of the sequential fraction of a parallel program which can be calculated as $F = \frac{(K \times T(N, K) - T(N, 1))}{(K \times T(N, 1) - T(N, 1))}$

Table 1.1 Performance Graphs for OpenMp and Parallel Java CG Benchmark

<table>
<thead>
<tr>
<th>Parallel Java</th>
<th>OpenMp Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="#" alt="Running Time vs. Processors" /></td>
<td><img src="#" alt="Running Time vs. Processors" /></td>
</tr>
<tr>
<td><img src="#" alt="N = Class B" /></td>
<td><img src="#" alt="N = Class B" /></td>
</tr>
<tr>
<td><img src="#" alt="N = Class A" /></td>
<td><img src="#" alt="N = Class A" /></td>
</tr>
<tr>
<td><img src="#" alt="Processors, K" /></td>
<td><img src="#" alt="Processors, K" /></td>
</tr>
</tbody>
</table>
### Parallel Java

**Speedup vs. Processors**

![Graph showing speedup vs. processors for Parallel Java with classes A and B labeled.](image)

**Efficiency vs. Processors**

![Graph showing efficiency vs. processors for Parallel Java with classes A and B labeled.](image)

**EDSF vs. Processors**

![Graph showing EDSF vs. processors for Parallel Java with class A and B labeled.](image)

### OpenMP Fortran

**Speedup vs. Processors**

![Graph showing speedup vs. processors for OpenMP Fortran with classes A and B labeled.](image)

**Efficiency vs. Processors**

![Graph showing efficiency vs. processors for OpenMP Fortran with classes A and B labeled.](image)

**EDSF vs. Processors**

![Graph showing EDSF vs. processors for OpenMP Fortran with class A and B labeled.](image)
Table 1.2. Running Time Values for OpenMp and Parallel Java for CG

<table>
<thead>
<tr>
<th>CG Parallel Java</th>
<th>A</th>
<th>4.43</th>
<th>2.66</th>
<th>2.14</th>
<th>1.45</th>
<th>1.26</th>
<th>1.15</th>
<th>1.29</th>
<th>1.3</th>
<th>1.51</th>
<th>1.8</th>
<th>2.28</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG Open Mp</td>
<td>A</td>
<td>3.41</td>
<td>1.71</td>
<td>1.13</td>
<td>0.82</td>
<td>0.54</td>
<td>0.41</td>
<td>0.3</td>
<td>0.23</td>
<td>0.18</td>
<td>0.16</td>
<td>0.15</td>
</tr>
<tr>
<td>CG Parallel Java</td>
<td>B</td>
<td>214.7</td>
<td>110.4</td>
<td>77.17</td>
<td>65.43</td>
<td>49.59</td>
<td>37.327</td>
<td>33.32</td>
<td>42.11</td>
<td>50.772</td>
<td>40.32</td>
<td>40.30</td>
</tr>
<tr>
<td>CG Open Mp</td>
<td>B</td>
<td>230.6</td>
<td>112.9</td>
<td>64.5</td>
<td>49.84</td>
<td>32.6</td>
<td>25.45</td>
<td>16.54</td>
<td>12.55</td>
<td>8.96</td>
<td>7.04</td>
<td>5.59</td>
</tr>
</tbody>
</table>

5.2. THE BT BENCHMARK

The BT benchmark solves a system of 5 non-linear partial differential equations (PDEs). These PDEs are modified from the realistic system of equations in order to make them suitable for benchmarking. This is represented by the equation given in figure 2.0 as defined in the NAS Parallel Benchmark spec, where E, F, G, T, V, W and H are (5 x 1) vectors with each element in the vector being a function of U. U on the other hand is a vector function that forms the orthogonal coordinate system in three-dimensional space. U is defined as $U = [U^1, U^2, U^3, U^4, U^5]^T$ and $\xi, \eta$ and $\zeta$ represent the X, Y, and Z directional factors respectively.

$$\frac{\partial U}{\partial \tau} = \frac{\partial E(U)}{\partial \xi} + \frac{\partial F(U)}{\partial \eta} + \frac{\partial G(U)}{\partial \zeta} + \frac{\partial T(U, U_\xi)}{\partial \xi} + \frac{\partial V(U, U_\eta)}{\partial \eta} + \frac{\partial W(U, U_\zeta)}{\partial \zeta} + H(U, U_\xi, U_\eta, U_\zeta). \quad (\tau, \xi, \eta, \zeta) \in D \times D$$

Figure 2.0 BT Partial Differential Equation.

The linear system of equations is dissolved into an implicit part (LHS) and an explicit part (RHS) using the Alternating Direction Implicit Factorization Scheme. The BT benchmark uses an iterative method to numerically compute the solution for $\Delta U^0$ which is then used to compute the $\Delta U^{n+1}$ and that solution used to compute $\Delta U^{n+2}$ and so on. The pseudocode below describes the functional breakdown of this iterative method.

From i to maxSteps

I. Initialize RHS
II. Solve the system of linear equations for $U_1$ along the x-axis
\[
\{ I - \Delta \tau [D_e(A)^n + D_e^2(N)^n] \} \Delta U_1 = \text{RHS}
\]

III. Solve the system of linear equations for $U_2$ along the y-axis
\[
\{ I - \Delta \tau [D_y(B)^n + D_y^2(Q)^n] \} \Delta U_2 = \Delta U_1
\]

IV. Solve the system of linear equations for $U_3$ along the z-axis
\[
\{ I - \Delta \tau [D_z(C)^n + D_z^2(S)^n] \} \Delta U_3 = \Delta U_2
\]

V. Update the solution for the current time-step
\[
a. \quad [U^{n+1}]_{i,j,k} = [U^n]_{i,j,k} + [\Delta U^n]_{i,j,k}, \quad \text{for} \quad (i,j,k) \in D_h
\]

We have analyzed, on a high level, the steps mentioned above in section 6.2. The BT benchmark and the SP benchmark share a lot of similarities in the PDEs and nature of parallelism therefore we go into deeper implementation details in section 6.3 when we start to discuss the SP benchmark. This will avoid repetition in describing the behavior of the parallel blocks, as BT and SP are almost identical in parallel structure.

5.2.1. Problem size definition

In determining the scalability and performance of the BT Benchmark, we used the problem sizes defined by the NAS parallel benchmarks. No changes were made to the algorithm to conform to any one particular problem size. This is to allow an accurate representation of the performance as problem sizes increased and number of processors varied. Varying the problem size of the BT benchmark is done by increasing or decreasing the number of iterations performed in computing the x, y and z directional sweeps as well as the time-step. Table 2.0 gives the problem sizes defined for the BT benchmark.

Table 2.0. BT problem sizes

<table>
<thead>
<tr>
<th></th>
<th>Problem size = 12 x 12 x 12</th>
<th>Problem size = 24 x 24 x 24</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Time-step = .01</td>
<td>Time-step = .0008</td>
</tr>
<tr>
<td>A</td>
<td>Problem size = 64 x 64 x 64</td>
<td>Problem size = 102 x 102 x 102</td>
</tr>
<tr>
<td></td>
<td>Time-step = .0008</td>
<td>Time-step = .0003</td>
</tr>
</tbody>
</table>

5.2.2. Changes made to array structure in translating OpenMP to Java

As in the Java implementation of the NAS Parallel benchmark, the use of linearized arrays was employed in the Parallel Java implementation. Though the dimensions were maintained, all FORTRAN arrays were flattened to a corresponding n x 1 dimensional equivalent. Linearization was achieved by multiplying the size of each dimension in a multi-dimensional array to derive a length for a one-dimensional array. Scalar constants were used in keeping track of the boundaries of each dimension in an array. For example where the OpenMP implementation defines the forcing, rhs and U matrix as follows:

```java
double precision
forcing (5, 0:IMAX/2*2, 0:JMAX/2*2, 0:KMAX/2*2),

u       (5, 0:IMAX/2*2, 0:JMAX/2*2, 0:KMAX/2*2),

rhs      (5, 0:IMAX/2*2, 0:JMAX/2*2, 0:KMAX/2*2)
```

Parallel Java defines it as follows:
In addition, a few changes were made to the class structure of the BT Parallel Java implementation. These changes are not performance based but rather made for convenience and to incorporate some of the object oriented structure of Java. Adi, Print_results and Setconstants were included in the BT.java class rather than being classes on their own. The two subroutines, error_norm and rhs_norm were made methods in the verify.java class whereas they were subroutines in error.f in OpenMp implementation.

5.2.3. The Parallel Structure of The BT Benchmark

In this section we look at a high level description of the algorithm defined in section 6.2.

The solution of the systems of linear equations defined in steps ii, iii, and iv in section 6.2 are called the x-solve, y-solve and z-solve respectively in the benchmark implementation. Below is a breakdown of the system of linear equation:

1. Parallel Loop start(k=start to end)
   a. For(i=1 to problem_size-2)
      i. For(j=1 to jsize)
         1. Compute the indices for storing the tri-diagonal matrix
         ii. end loop
         iii. form left hand side in x, y or z direction
         iv. performs gaussian elimination on this cell
         v. begin inner most do loop, do all the elements of the cell unless last
         vi. subtract A*lhs_vector(j-1) from lhs_vector(j)
             rhs(j) = rhs(j) - A*rhs(j-1)
         vii. B(j) = B(j) - C(j-1)*A(j)
         viii. multiply c(i,j,k) by b_inverse and copy back to c
              ix. multiply rhs(i,1,k) by b_inverse(i,1,k) and copy to rhs
              x. rhs(jsize) = rhs(jsize) - A*rhs(jsize-1)
              xi. B(jsize) = B(jsize) - C(jsize-1)*A(jsize)
              xii. multiply rhs[jsize] by b_inverse[jsize] and copy to rhs
              xiii. back solve
      b. end loop
   2. end parallel block

The code snippet in Figure 2.1 shows parts of this implementation in Parallel Java. Although the work is initially divided on the topmost level loop, this is not an indication of the amount of work that the threads have to perform. One would need to take a close look at the algorithm itself as it contains numerous large running loops within. Therefore the work performed by each thread can be seen as growing polynomially when we increase the problem size due to the number of nested loops within these three core functions (x-solve, y-solve and z-solve). We will analyze the performance of the different problem sizes in the section that follows.
team.execute(new ParallelRegion(){
    public void run() throws Exception{
        execute(1, grid_points[2]-2, new IntegerForLoop(){
            int i, j, k, n, m;
            public void run(int first, int last){
                for(k = first; k<last; k++){
                    for(j=1; j<grid_points[1]-2; j++){
                        for(i = 0; i<=isize; i++){
                            tmp1 = rho_i[i+j*jsize1+k*ksize1];
                            tmp2 = tmp1 * tmp1;
                            tmp3 = tmp1 * tmp2;
                        }
                    }
                }
            }
        });
    }
});

Figure 2.1 code snippet from BT x-solve

5.2.4. PERFORMANCE OF BT BENCHMARK

There was a significant difference in performance between the Parallel Java implementation and OpenMp implementation of the BT benchmark. Perhaps one of the fundamental reasons why Fortran outperforms the Parallel Java implementation lies in the structure of the languages themselves and their compilers. The majority of this benchmark is made up of array-based computations and here is where Fortran arguably beats a lot of programming languages. Java in itself has no real concept of multi-dimensional arrays but instead uses an array of arrays. Using the flattened out array described in section 6.2.2 came with a cost of performing mathematical computations to determine array indexes. Further investigation would be needed to prove that better performance is gained when using flattened arrays versus multidimensional arrays or vice versa, this is mentioned in future work. The use of flattened arrays here is however enough to be an overhead cost which can cause a lag in performance as the OpenMp benchmark is not subjected to these mathematical computations.

Although the JIT compiler is optimized for array bound checking [6] based on static indices, not all bound checks are eliminated. Given the high amount of array access and reassignments done in the x-solve, y-solve and z-solve methods, the cost of array bound checking could add up. This could be a contributing factor to the performance difference with Fortran OpenMp on GCC compiler where no array bound checks are done.
Table 2.1. Performance Graphs for OpenMp and Parallel Java BT Benchmark

<table>
<thead>
<tr>
<th>Parallel Java</th>
<th>OpenMp Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Running Time vs. Processors</strong></td>
<td><strong>Running Time vs. Processors</strong></td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td><strong>Speedup vs. Processors</strong></td>
<td><strong>Speedup vs. Processors</strong></td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
<tr>
<td><strong>Efficiency vs. Processors</strong></td>
<td><strong>Efficiency vs. Processors</strong></td>
</tr>
<tr>
<td><img src="image" alt="Graph" /></td>
<td><img src="image" alt="Graph" /></td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Parallel Java</th>
<th>OpenMp Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="EDSF vs. Processors" /></td>
<td><img src="image2.png" alt="EDSF vs. Processors" /></td>
</tr>
</tbody>
</table>

Table 2.2. Running Time Values for OpenMp and Parallel Java for BT.

<table>
<thead>
<tr>
<th>BT Parallel Java</th>
<th>A</th>
<th>2164.441</th>
<th>2195.028</th>
<th>1168.339</th>
<th>1120.434</th>
<th>1046.584</th>
<th>1033.877</th>
<th>1072.921</th>
<th>1011.493</th>
<th>982.223</th>
<th>980.032</th>
<th>709.639</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT OpenMp</td>
<td>A</td>
<td>119.84</td>
<td>60.98</td>
<td>42</td>
<td>32.13</td>
<td>21.78</td>
<td>15.74</td>
<td>12.59</td>
<td>8.24</td>
<td>6.6</td>
<td>5.16</td>
<td>5.18</td>
</tr>
</tbody>
</table>

5.3. The SP Benchmark

SP defines a solution of multiple, independent systems of nondiagonally dominant scalar pentadiagonal equations [1]. This is a computationally intensive CFD application much like BT. The Beam-warning approximate factorization is used to breakdown a system of equations into the x, y and z coordinates, and a solution is computed along each of these coordinates. The detailed algorithm can be found in the NAS technical document [1].

Just as in the BT benchmark, most of the structure of the code was maintained in sync with the OpenMp implementation. The naming convention was followed for functions and variables in the most part with changes being made to new variables introduced as a result of finer implementation needs. Certain methods were however moved around to make the code more readable. These changes are minimal and do not seem to hinder performance. One of the changes made is the Adi method was included in the sp.java class rather than in a class of its own. Another change is the two subroutines, error_norm and rhs_norm in error.f were made methods in the verify.java class and finally the Setconstants subroutine was moved to the sp.java class.

5.3.1. Problem Sizes

The SP problem sizes vary only in the size of the 3 arrays grid_points1, grid_points2 and grid_points3. The number of iterations performed by the beam-warming algorithm (represented by the variable “niter”) remains the same in
all the class sizes and has a value of 400. The table below gives the values of the variables for the 5 different class sizes starting from the smallest to the largest.

Table 3.0. SP problem sizes

<table>
<thead>
<tr>
<th></th>
<th>problem_size=12</th>
<th>problem_size=36</th>
<th>problem_size=64</th>
<th>problem_size=102</th>
</tr>
</thead>
</table>

5.3.2. TRANSLATING THE SP BENCHMARK FROM OPENMP TO JAVA

In this section, we will go into the implementation details of the “adi” method, which forms the core of the beam warming algorithm in the SP Benchmark. We compare some of the parallel code blocks within adi with the OpenMp implementation.

5.3.2.1. A BREAKDOWN OF THE ADI METHOD.

The “adi” function is responsible for calling a one-time iteration and comprises of five major computations given below:

1. Compute RHS
2. Solving the equation in the X-dimension
3. Solving the equation in the Y-dimension
4. Solving the equation in the Z-dimension and
5. Adding the three solutions.

The first four operations form the most computationally intensive part of the SP benchmark. We would therefore analyze these four in the sections that follow, in order to gain an understanding of the performance benefits or drawbacks that Parallel Java has compared to OpenMp.

5.3.2.2. PARALLEL JAVA IMPLEMENTATION OF COMPUTE RHS:

The first step in computing RHS is to compute the reciprocal of density, kinetic energy and the speed of sound. This calculation requires iterating through the array “U” (which holds the solution to the PDEs) to compute the reciprocals mentioned, then storing it in the arrays us, vs and ws which correspond to speed, velocity and time respectively.

The absence of dependencies on previously computed values of us, vs and ws gives us the ability to parallelize this computation at the highest loop level in a three level nested for loop. This gives better scalability as we increase the number of processors. A similar approach was taken to the OpenMp parallelization of this loop, where the topmost for loop is parallelized to give the best performance.

Parallelizing in this manner yields better performance for many reasons. One of which is the overhead of initializing a parallel Region. When a thread comes to a parallel Region, Parallel Java determines if this thread is the
first to arrive at the parallel block by iterating through each thread and using a counter to determine if it is the first to arrive. Next the thread sets the scheduler for each thread and assigns the chunk sizes. This setup overhead is something that we would rather do only once per parallel block. The larger the number of threads in our team the longer it would take to iterate through and set them all which would hinder scalability.

Another preference for moving the parallel Region to the toposfor loop is in maximizing the amount of work each thread does at a go. In Figure 3.0, notice that the work divided in the parallel Region is the size of the array \textit{grid\_points} which we saw earlier from the problem size definition in Table 3.0 can go up to 102.

```
  team.execute(new ParallelRegion(){
    public void run() throws Exception{
      execute(0, grid_points[2]-1, new IntegerForLoop(){
        int i, j, k;
        double aux, rho_inv;
        public void run(int first, int last){
          for(k=first; k<=last; k++){
            for(j=0; j<=grid_points[1]-1; j++){
              for(i=0; i<=grid_points[0]-1; i++){
                rho_inv = 1.0d0/u(1,i,j,k);
                rho_i(i,j,k) = rho_inv;
                us(i,j,k) = u(2,i,j,k) * rho_inv;
                vs(i,j,k) = u(3,i,j,k) * rho_inv;
                ws(i,j,k) = u(4,i,j,k) * rho_inv;
                square(i,j,k) = 0.5d0* (u(2,i,j,k)*u(2,i,j,k) + u(3,i,j,k)*u(3,i,j,k) +u(4,i,j,k)*u(4,i,j,k) ) * rho_inv;
                qs(i,j,k) = square(i,j,k) * rho_inv;
              }
            }
          }
        }
      }
    }
  }
```

Figure 3.0. A parallel Block from Compute RHS (Parallel Java)

This pattern repeats throughout the compute RHS function where the outermost loop is parallelized to increase scalability and efficiency. The same is applied in the OpenMp implementation as can be seen in Figure 3.1 below.

```
!$omp do schedule(static)
  do  k = 0, grid_points(3)-1
    do  j = 0, grid_points(2)-1
      do  i = 0, grid_points(1)-1
        rho_inv = 1.0d0/u(1,i,j,k)
        rho_i(i,j,k) = rho_inv
        us(i,j,k) = u(2,i,j,k) * rho_inv
        vs(i,j,k) = u(3,i,j,k) * rho_inv
        ws(i,j,k) = u(4,i,j,k) * rho_inv
        square(i,j,k) = 0.5d0* (u(2,i,j,k)*u(2,i,j,k) + u(3,i,j,k)*u(3,i,j,k) +u(4,i,j,k)*u(4,i,j,k) ) * rho_inv;
        qs(i,j,k) = square(i,j,k) * rho_inv;
      end do
    end do
  end do
```

aux = c1c2*rho_inv* (u(5,i,j,k) - square(i,j,k))
speed(i,j,k) = dsqrt(aux)

end do
5.3.2.3. Solving the Equation in the X-Direction (XSolve):

In this section, we will discuss the x-solve, y-solve and z-solve together because these three functions were parallelized following the same pattern. The examples that will be given are from the x-solve method and it will be enough to explain for us the details of the parallel structure for all three functions mentioned.

The x-solve, y-solve and z-solve perform the tridiagonal matrix algorithm also known as the Thomas Algorithm. This involves doing a forward elimination and a backward substitution. The forward elimination only deals with the first row of the tridiagonal entries, while the backward elimination solves the last two diagonal rows. This is an iterative approach that is easily parallelized. In parallelizing the forward elimination, we started our parallel block on the outer loop in a 2-level deep nested loop as seen in figure 3.2.

```java
public void x_solve(){
    if (timer) timer.timerStart(t_xsolve);

    for(k=1; k<=nz2; k++){
        for(i=1; i<=ny2; i++){
            // first fill the lhs for the u-eigenvalue
            for(i=0; i<=grid_points[0]-1; i++){
                ru1 = c3c4*rho_i[i]*jsize2+k*ksize2;
                cv[i] = us[i+j]*jsize2+k*ksize2;
                rhon[i] = dmax1(dx2+con43*ru1,
                                dx5+c1c5*ru1,
                                dx1);
            }

            // perform the forward elimination
            for(i=0; i<=grid_points[0]-3; i++){
                i1 = i + 1;
                i2 = i + 2;
                fac1 = 1.0 / lhs[2+i*jsize4];
                lhs[3+i*jsize4] = fac1*lhs[3+i*jsize4];
                lhs[4+i*jsize4] = fac1*lhs[4+i*jsize4];

                for(m=0; m<3; m++){
                    rhs[m+i*jsize1+j]*jsize1+k*ksize1] =
                    fac1*rhs[m+i*jsize1+j]*jsize1+k*ksize1];
                }

                lhs[2+i*jsize4] = lhs[2+i*jsize4] -
                                lhs[1+i*jsize4]*lhs[3+i*jsize4];
            }

            for(m=0; m<3; m++){
                // perform the forward elimination
                for(i=0; i<=grid_points[0]-3; i++){
                    i1 = i + 1;
                    i2 = i + 2;
                    fac1 = 1.0 / lhs[2+i*jsize4];
                    lhs[3+i*jsize4] = fac1*lhs[3+i*jsize4];
                    lhs[4+i*jsize4] = fac1*lhs[4+i*jsize4];

                    for(m=0; m<3; m++){
                        // perform the forward elimination
                        for(i=0; i<=grid_points[0]-3; i++){
                            i1 = i + 1;
                            i2 = i + 2;
                            fac1 = 1.0 / lhs[2+i*jsize4];
                            lhs[3+i*jsize4] = fac1*lhs[3+i*jsize4];
                            lhs[4+i*jsize4] = fac1*lhs[4+i*jsize4];
                    }
                    ...
                }
            }
        }
    }

    if (timer) timer.timerEnd(t_xsolve);
}
```

Figure 3.1. A parallel Block from Compute RHS (OpenMp)
Figure 3.2. Snippet of a sequential and parallel implementation of x-solve.

Figure 3.2 shows a side-by-side snippet of the sequential and parallel implementation of x-solve. From the structure of the loops shown in this figure, it is apparent that parallelizing the outermost loop would yield the best performance as mentioned earlier. Computation done within the innermost loop is independent of previously computed values outside the scope of the thread in execution, therefore parallelizing in this way does not run us into concurrency problems.

5.3.2.4. Adding the Three Solutions

Adding the three equations is done in the add function. All computed values in the rhs array are added to the “U” array. The add function is a straightforward parallelization at the topmost loop similar to x-solve discussed above and contains no reduction or final clauses due to the independence in the work to be performed by each thread.

5.3.3. Performance
5.3.3.1. Array Access Optimization

In Fortran, many of the arrays are optimized and transformed by default by the Intel Fortran optimizer. The OpenMp implementation of multi dimensional arrays makes use of the column-major-order of Fortran. This order improves array access performance and specifies that arrays be accessed with the leftmost subscript being the one that varies more rapidly with a stride of one [7]. This is quite different from the Parallel Java implementation of the array access. Parallel Java implementation of SP uses the flattened arrays mentioned in section 6.2.2. This implementation however, bears a tradeoff in the computation done in order to get the value of the index for the equivalent position of a variable in the one-dimensional array. An example is shown below in Figure 3.3, a snippet from the x-solve function where we fill the factors (u+c) and (u-c).

Notice how the three-dimensional array speed is accessed here, in order to get the index we perform two multiplications (j*size2 and k*size2) as well as some additions. These mathematical operations are not done using the Long type and so can be extremely costly. Even with compiler optimizations this does cost us a bit in terms of performance because of the numerous places this has to be done by every thread for every nested loop.

```java
for (i=1; i<=nx2; i++) {
    lhs[0+i*jsize4] = lhs[0+i*jsize4];
    lhs[1+i*jsize4] = lhs[1+i*jsize4] - dttx2 * speed[i-1]+j*size2+k*size2]...
}
```
Figure 3.3. Sample computation done to retrieve indexes

Table 3.1. Performance Graphs for OpenMp and Parallel Java SP Benchmark

<table>
<thead>
<tr>
<th>Parallel Java</th>
<th>OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Running Time vs. Processors</strong></td>
<td><strong>Running Time vs. Processors</strong></td>
</tr>
<tr>
<td><img src="image1" alt="Graph" /></td>
<td><img src="image2" alt="Graph" /></td>
</tr>
<tr>
<td><strong>Speedup vs. Processors</strong></td>
<td><strong>Speedup vs. Processors</strong></td>
</tr>
<tr>
<td><img src="image3" alt="Graph" /></td>
<td><img src="image4" alt="Graph" /></td>
</tr>
<tr>
<td><strong>Efficiency vs. Processors</strong></td>
<td><strong>Efficiency vs. Processors</strong></td>
</tr>
<tr>
<td><img src="image5" alt="Graph" /></td>
<td><img src="image6" alt="Graph" /></td>
</tr>
</tbody>
</table>
5.3.3.2. **The Nature of the Parallel Work**

*x*-solve, *y*-solve and *z*-solve are heavy computational functions. The chunk of the loop given to each thread is minimal whereas the work done within the loop by each thread is a lot. For example, in *y*-solve, the integerForLoop size division is at most 102 (Problem size B), which is then divided into the number of threads. Where we have about 48 threads one can see that the chunk of the array given to each thread is only about a size of three to work on yet the work within the loop itself is heavy.

Table 2.2. Running Time Values for OpenMp and Parallel Java for SP.

<table>
<thead>
<tr>
<th>SP Parallel Java</th>
<th>A</th>
<th>196.61</th>
<th>185.04</th>
<th>124.79</th>
<th>106.63</th>
<th>80.70</th>
<th>71.27</th>
<th>65.83</th>
<th>61.88</th>
<th>68.00</th>
<th>67.01</th>
<th>70.26</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP OpenMp</td>
<td>A</td>
<td>96.19</td>
<td>48.84</td>
<td>33.22</td>
<td>25.08</td>
<td>17.24</td>
<td>12.51</td>
<td>11.21</td>
<td>7.04</td>
<td>5.98</td>
<td>5.06</td>
<td>4.99</td>
</tr>
</tbody>
</table>

5.4. **The LU Benchmark**

LU solves 5 X 5 sparse matrix blocks of lower and upper triangular systems. This is another CFD application that solves the Navier-Stokes equation using the symmetric successive over-relaxation (SSOR) method. The Navier-Stokes equation is used to describe fluid motion by applying the sum of a viscous term and a pressure term as fluid stress to Newton’s second law.

**The LU Decomposition**

Given a matrix A and assuming we are able to write A as a product of two matrices, one of which is a lower triangular matrix and the other an upper triangular matrix. We can express A as equation 4.0 below:
\[ A = L \cdot U \] (4.0)

Now if we write matrix \( A \) as a system of linear equations, we get equation 4.1 and 4.2 below:

\[ Ax = b \] (4.1)

\[ (L \cdot U) x = b \] (4.2)

To solve equation 4.2, we can break it down into solving two logical equations 4.4 and 4.5 given below. The LU benchmark seeks to solve the breakdown of equation 4.4 and 4.5.

\[ L \cdot y = b \] (4.4)

\[ U \cdot x = y \] (4.5)

The actual implementation of the LU benchmark does not transform \( A \) into lower and upper triangular matrices but rather implements a symmetric successive over-relaxation (SSOR) numerical scheme to solve a regular-sparse, block lower and upper triangular system. SSOR solves the left hand side of the equation for \( x \), using previously computed values of \( x \) on the right hand side. In finding the solution for the lower and upper triangular decompositions we do a forward elimination and a backward substitution. We can summarize the steps taken within the SSOR computation in the LU benchmark as follows:

1. Form the lower triangular part of the jacobian matrix
2. Perform the lower triangular solution
3. Form the strictly upper triangular part of the jacobian matrix
4. Perform the upper triangular solution
5. Update the variables

In later sections, we will take a deeper dive into each of the steps above and analyze the details of the Parallel Java implementation.

5.4.1. Problem sizes

The LU benchmark defines different problem sizes using three sets of variables or parameters. The first is \( isize \), which defines the number of grid points in each dimension. The time step is defined by \( dt_{default} \) while \( inorm_{default} \) defines the frequency with which the norm is printed out to the console during iterations. \( itmax_{default} \) defines the quantity of time steps that have been taken. Table 4.0 shows the value of these variables for the W and A problem size.

Table 4.0. LU problem sizes

<table>
<thead>
<tr>
<th>Class W</th>
<th>Class A</th>
</tr>
</thead>
<tbody>
<tr>
<td>( isize1=isize2=isize3=33 );</td>
<td>( isize1=isize2=isize3=64 );</td>
</tr>
<tr>
<td>( itmax_{default}= 300 );</td>
<td>( itmax_{default}= 250 );</td>
</tr>
<tr>
<td>( inorm_{default}=300 );</td>
<td>( inorm_{default}=250 );</td>
</tr>
<tr>
<td>( dt_{default}=0.015 );</td>
<td>( dt_{default}=2 );</td>
</tr>
</tbody>
</table>

5.4.2. Translation from OpenMP to Parallel Java

This section will discuss the implementation details of the SSOR algorithm. We will take each step mentioned in section 6.4 and look into the Parallel Java implementation.
5.4.2.1. Forming the Lower and Upper Triangular Matrices (Steps 1 and 3)

In forming the lower and upper triangular matrices, we first form the block diagonal \(d\) and then we form the first second and third sub-diagonals, \(a\), \(b\) and \(c\) in that order. Forming these matrices does not require previously computed values from either of them therefore there are no dependencies tied to this operation. This allows us to parallelize this at the highest loop level and eliminates the need for setting thread barriers.

The solution to the partial differential equations is done using a 3-dimensional grid; there will be three level deep loops involved in forming the corresponding \(d\), \(a\), \(b\) and \(c\) matrices. Each of these loops go from 0 to the number of grid points (isize) defined, which varies with the problem size. Given that the grid size itself is not a large number it may appear that the workload division is minimal, however a three level deep loop increases this workload in an exponential way. Figure 4.0 is a Parallel Java code snippet showing the workload division performed within the \textit{jacld} method responsible for computing the lower triangular matrix.

```
  team.execute(new ParallelRegion(){
    public void run() throws Exception{
      execute(jst, jend-1, new IntegerForLoop(){
        int i, j;
        double r43 = ( 4.0 / 3.0 );
        double c1345 = c1 * c3 * c4 * c5;
        double c34 = c3 * c4;
        double tmp1, tmp2, tmp3;
        public void run(int first, int last){
          for(j=first; j<=last; j++){
            for(i=ist-1; i<=iend-1; i++){
              tmp1 = rho_i[i+j*jsize3+k*ksize3];
              tmp2 = tmp1 * tmp1;
              tmp3 = tmp1 * tmp2;
              
              d[0+0*isize4+i*jsize4+j*ksize4] = 1.0
              + dt * 2.0 * ( tx1 * dx1+ ty1 * dy1+ tz1 * dz1 );
              d[0+1*isize4+i*jsize4+j*ksize4] = 0.0;
              ...
            }
          }
        }
      });
    }
  });
```

Figure 4.0 Snippet from the \textit{Jacld} method.

5.4.2.2. Computing the Surface Integral

After the \textit{ssor} has been computed, we compute the surface integrals, which requires thread synchronization in the summation of the solutions. This requires the values computed from the solutions of the partial differential equation to be collapsed into a single value representing the solution to the partial differential equation previously discussed. In order to achieve this, we first divide up the matrix between the threads and assign to each thread a local array to enter in their summations. Then we do a reduction to add the thread local summations into one solution. Here a barrier action is not used but rather the \textit{finish()} clause at the end of the threads iteration. The code snippet below shows this.
Parallel Java

```java
team.execute(new ParallelRegion(){
    public void run() throws Exception{
        execute(jbeg-1, jfin-1, new IntegerForLoop(){
            int i, j, k;
            public void run(int first, int last){
                for(j=first; j<=last; j++){
                    for(i=ibeg-1; i<=ifin-1; i++){
                        ...
                    }
                }
                public void finish(){
                    // adding up all our collapsed d2 sums
                    for(j=jbeg-1; j<=jfin1-1; j++){
                        for(i=ibeg-1; i<=ifin1-1; i++){
                            frc1 = frc1 + ( phi1[i+j*size5] + phi1[(i+1)+j*size5] + phi1[i+(j+1)*size5] + phi1[(i+1)+(j+1)*size5] + phi2[i+j*size5] );
                        }
                    }
                }
            }
        }
    }
}
```
which gives it a performance edge over Parallel Java. The finish method in Parallel Java is called by the main thread and is executed only after the threads have all exited the run method. This is therefore done sequentially.

Mathematical Computation

Just as in the BT and SP benchmarks, mathematical equations performed in order to retrieve values at an array index bears a performance cost for Parallel Java. The same would apply to LU here where we do countless number of mathematical multiplications and additions without first converting the values to a Long type.

Table 4.2 performance summary of LU
Parallel Java | OpenMp
--- | ---

![Efficiency vs. Processors](image1)

![Efficiency vs. Processors](image2)

![EDSF vs. Processors](image3)

![EDSF vs. Processors](image4)

Table 2.2. Running Time Values for OpenMp and Parallel Java for LU.

<table>
<thead>
<tr>
<th></th>
<th>Parallel Java</th>
<th></th>
<th>OpenMp</th>
</tr>
</thead>
<tbody>
<tr>
<td>LU</td>
<td>W</td>
<td>463.241</td>
<td>256.412</td>
</tr>
<tr>
<td>LU</td>
<td>W</td>
<td>14.70</td>
<td>8.80</td>
</tr>
</tbody>
</table>

6.0. RELATED WORK

The Java implementation of the NAS parallel Benchmarks using Java threads showed similar performance differences between their implementation of the benchmark in Java and the OpenMp and MPI implementation.
The best reported result of the CFD benchmarks was achieved on an IBM p690 (1.3 GHz, 32 processors)[2], where a Java/FORTRAN execution ratio of 8.3-10.8 was achieved.

7.0. LESSONS LEARNED

7.1. PRELIMINARY WORK

Much time was spent in stepping through the SP and CG OpenMP benchmarks using the GNU FORTRAN debugger in order to understand the benchmark algorithms. An easier and more efficient approach was to do a static analysis of the code and look up FORTRAN syntax and Symantec directly.

7.2. PARALLEL JAVA AND OPENMP

7.2.1. ARRAY ACCESS

In the benchmarks BT, SP and LU, we used multi-dimensional arrays, but memory is inherently one-dimensional. FORTRAN uses the column-major order in traversing multidimensional arrays where the first array index varies fastest. When we write out to an array in FORTRAN using the array name, as was done in all four benchmarks, FORTRAN writes this out in the order mentioned. This is efficient because memory will be traversed in sequence as the array is written out. In order for us to come close to this type of traversal, we flattened the Parallel Java multidimensional arrays. This came with a cost of performing an enormous amount of mathematical computations, which could create the performance lag we see between the OpenMp and Parallel Java benchmarks.

7.2.2. THREAD STARTUP OVERHEAD AND NESTED PARALLELISM

In our benchmark implementation, Parallel Teams were created only once during the lifetime of the running benchmark. However, multiple Parallel Regions were created within the benchmark so we will look at the lesson learnt in the difference between starting up a parallel block in OpenMp versus Parallel Java.

In Parallel Java, there are two important steps worth mentioning which are, the step to determine the first thread to arrive at the parallel block and the step to setup the scheduler object for each thread. This is of importance to us because these two steps require iterating through the number of threads in the team. Therefore increasing the number of threads bears us a little more setup cost.

OpenMp bears some cost as well with regards to thread creation. The master thread spawns the threads needed in a parallel team. Arguably, an advantage OpenMp has over Java is that each thread created by the master thread has the ability to spawn other threads in the event of a nested parallel region. That thread then becomes the master of the threads it spawned. This is a feature that is not available in Parallel Java. If one tried to execute a parallel block within a parallel block, the Parallel Team class treats this as an exception and declares that a Parallel Region is already in execution. Perhaps there is a benefit to having the ability to have nested parallelism and this is something that would be mentioned in future work.

7.2.3. END OF LOOP SYNCHRONIZATION

When a thread comes to the end of execution of its parallel chunk of work and exits the run method, Parallel Java uses a semaphore to coordinate a wait for all threads in a team to finish their execution of the Parallel Region before the main thread enhances to the finish block. This is no different from the OpenMp end-of-loop synchronization. All threads executing a parallel loop denoted by #pragma omp will block at the end of the loop until all threads in the team have completed their chunk of work. The difference lies in the way the two libraries
handle reduction. As discussed in earlier sections, OpenMp handles reduction in a semi-parallel manner whereas Parallel Java handles reduction in a sequential manner.

8.0. CONCLUSION AND FUTURE WORK

Parallel Java is by no doubt miles ahead of using the native Java code to do parallelism and has elegantly re-engineered native Java to allow complex thread creation and synchronization with ease to the developer. As the JIT compiler continues to improve and get better at code optimization at the compiler level, one can expect to see even more astonishing results from the Parallel Java API.

It is important to continue to look for ways to improve Parallel Java as the new wave of cloud computing and multi-threaded computing is taking place. A simple example is in improving the amount of code needed to construct a parallel block. From the examples of code given in this document, one can see that there is a lot more code needed to construct a parallel block using Parallel Java than is needed when using OpenMp.

As discussed in performance and efficiency analysis of the benchmarks, the CFD benchmarks, SP, LU and BT did not perform as efficient as the OpenMP implementation. We have therefore compiled a list of future work that can be done in order to improve performance of these benchmarks in Parallel Java. These include but are not limited to the following:

- Restructuring array access to take advantage of further improvements of the Java JIT compiler and Java Native Code Compilers especially in array bound checking.
- Restructure and reduce the number of temporary objects being used in the parallel loops within the benchmarks.
- Convert all variables to a Long type when performing mathematical operations. This would especially improve performance in the BT, SP and LU benchmarks.
- Use more tools to analyze memory usage and areas where code could be optimized.
- Look into the possibility of implementing the BT, SP and LU benchmarks in cluster Parallel Java to compare performance with SMP. This would give us more insight into the effect of cache sharing on these benchmarks.
- Look into the possibility of nested parallelism and how it may benefit Parallel Java from a performance standpoint.
- Investigate the Parallel Java performance difference using java multidimensional arrays instead of the flattened array structure used in the current implementation.
9.0. References


