Benchmarking Parallel Java
Benchmarking Parallel Java
Benchmarking Parallel Java

What is Parallel Computing?

Suppose you want to build the Empire State Building, and you have work that can be done independent of each other. If you hire 100 workers to do one of these jobs, you will be done a whole lot faster maybe a 100 times faster.

In the simplest sense, parallel computing is the ability to break down computational problems into chunks that can be solved concurrently using multiple processors.

The NAS Parallel Benchmarks

- Aid in comparing parallel middleware
- Open Source
- Among the most highly used open source benchmarks.

Relevance in Today’s World

- Computational Fluid Dynamics
- Call Centers
- Parallel Java sits on top of Java

Hypothesis

Parallel Java is effective in parallelizing the NAS Parallel Benchmarks. This will be gauged by porting the OpenMP version of the NPB to PJ and compare PJ’s performance based on the calculated speedup, steep and efficiency of the standard benchmarks to the performance of the OpenMP version.

How Does Parallel Java Fit into all this?

- More hardware showing up for parallel computing
- Better programming models
- Educating ourselves better on parallelism
- Better tools (e.g., Parallel Java, OpenMP)
- Parallel Java sits on top of Java
Masters Project
Asma'u Sani Mohammed
5/12/2013
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
What is Parallel Computing?

Suppose you want to build the Empire State Building, and you have work that can be done independent of each other. If you hire 100 workers to do one of these jobs, you will be done a whole lot faster maybe a 100 times faster.

In the simplest sense, parallel computing is the ability to break down computational problems into chunks that can be solved concurrently using multiple processors.
Relevance in Today's World

Computational Fluid Dynamics

Call Centers
How Does Parallel Java Fit into all this?

- More hardware showing up for parallel computing
- Better programming models
- Educating ourselves better on parallelism
- Better tools (e.g. Parallel Java, OpenMp)
- Parallel Java sits on top of Java
Hypothesis

Parallel Java is effective in parallelizing the NAS Parallel Benchmarks. This will be gauged by porting the OpenMP version of the NPB to PJ and compare PJ's performance based on the calculated speedup, sizeup and efficiency of the standard benchmarks to the performance of the OpenMp version.
The NAS Parallel Benchmarks

- Aid in comparing parallel middleware
- Open Source
- Among the most highly used open source benchmarks.

<table>
<thead>
<tr>
<th>EP</th>
<th>Embarrassingly Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>Conjugate Gradient</td>
</tr>
<tr>
<td>BT</td>
<td>Block Tridiagonal</td>
</tr>
<tr>
<td>SP</td>
<td>Scalar Pentdiagonal</td>
</tr>
<tr>
<td>LU</td>
<td>Lower and Upper triangular systems</td>
</tr>
<tr>
<td>MG</td>
<td>MultiGrid</td>
</tr>
<tr>
<td>FT</td>
<td>Fast Fourier Transform</td>
</tr>
</tbody>
</table>
Among the most highly used open source benchmarks.

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EP</td>
<td>Embarrassingly Parallel</td>
</tr>
<tr>
<td>CG</td>
<td>Conjugate Gradient</td>
</tr>
<tr>
<td>BT</td>
<td>Block Tridiagonal</td>
</tr>
<tr>
<td>SP</td>
<td>Scalar Pentdiagonal</td>
</tr>
<tr>
<td>LU</td>
<td>Lower and Upper triangular systems</td>
</tr>
<tr>
<td>MG</td>
<td>MultiGrid</td>
</tr>
<tr>
<td>FT</td>
<td>Fast Fourier Transform</td>
</tr>
</tbody>
</table>
Implementation

The CG Benchmark uses a conjugate gradient method to find the largest eigenvalue of a symmetric positive-definite sparse matrix. An attempt to find the solution to the linear equation

\[ A x = \beta \]

**Complexity and size variance of the CG Benchmark**

- \( n \) — define the size of the sparse matrix \( A \)
- \( m \) — the number of linear equations
- \( d \) — the number of components
- \( k \) — the size of the vector \( \beta \)
- \( \alpha \) — the eigenvalue

**Computing \( q = A p \)**

1. \( \text{OpenMP} \) implementation
2. \( \text{Parallel} \) implementation
3. \( \text{Serial} \) implementation
4. \( \text{compute residual norm} \)
5. \( \text{compute} \)
6. \( \text{compute} \)
7. \( \text{compute} \)
8. \( \text{compute} \)
9. \( \text{compute} \)
10. \( \text{compute} \)

**The Conjugate Gradient Routine**

- Compute \( q = A p \)
- Compute \( p \)
- Compute \( \alpha = \beta / (q^T p) \)
- Compute \( x = x + \alpha p \) and \( x = x + \alpha p \)
- Compute \( r = r - \alpha p \)
- Compute residual norm: \( ||e|| = ||x - A x|| \)

**Computing \( z \) and the norm of \( z \)**

Once we have computed and returned the sum derived from the compute gradient loop (that is, the Euclidean norm of the residual vector \( r \)), we need to compute \( x \) and the norm of \( z \). The norm of \( z \) is the norm of the eigenvector \( \lambda \).

\[ x = A x \]
The CG Benchmark uses a conjugate gradient method to find the largest eigenvalue of a symmetric positive definite sparse matrix. An attempt to find the solution to the linear equation

\[ Az = x \]
Complexity and size variance of the CG Benchmark

Each problem size defines six variables:
- na – defines the size of the sparse matrix A
- nonzero – the number of non zeros
- niter – the number of outer iterations
- shift – the eigen value λ
- zeta_verify_value – the value of the approximate solution

<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>( nonzero = 11 )</td>
<td>( nonzero = 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>
- shift – the eigen value $\lambda$
- zeta_verify_value – the value of the approximate solution

<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>$nonzero = 11$</td>
<td>$nonzero = 13$</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>
The Conjugate Gradient Routine

- compute $q$ where $q = A.p$
- compute $p.q$
- compute $\alpha = \rho / (p.q)$
- compute $z = z + \alpha \cdot p$ and $r = r - \alpha \cdot q$
- compute $\rho = r.r$
- compute $p = r + \beta \cdot p$
- compute residual norm: $||r|| = ||x - A.z||$
Computing $q = A\cdot p$

OpenMP implementation

1. !$omp parallel default(shared)
2. private(j,k,sum,alpha,beta)
3. !$omp& shared(d,rho0,rho)
4. !$omp do
5. do j=1,lastrow-firstrow+1
6. sum = 0.d0
7. do k=rowstr(j),rowstr(j+1)-1
8. sum = sum + a(k)*p(colidx(k))
9. enddo
10. q(j) = sum
11. 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;
compute residual norm

OpenMp implementation
1. sum = 0.0d0
2. !Smp parallel default(shared) private(j,d)
3. !Smp& shared(sum)
4. !Smp do
5. do j=1,lastrow-firstrow+1
6. d = 0.d0
7. do k=rows[j],rows[j+1]-1
8. d = d + a(k)^2*colidx(k)
9. enddo
10. r(j) = d
11. enddo
12. !Smp do reduction(+;sum)
13. do j=1,lastcol-firstcol+1
14. d = x(j) - r(j)
15. sum = sum + d*d
16. enddo
17. !Smp end parallel

Parallel Java Implementation
1. try{
2. team execute(new ParallelRegion[]){
3. public void run() throws Exception{
4. for(int i=first; i<last; i++)
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 = rows[j]; k<rows[j+1]-1; k++)
10. d2 += a[k]^2*colidx(k);
11. }
12. r[i] = d2;
13. }
14. }
15. }
16. }
17. }
18. }
19.catch(Exception e){
20. System.out.println(e);
21. }}
22. try{
23. team.execute(new ParallelRegion[]){
24. public void run() throws Exception{
25. for(int i=first; i<last; i++)
26. double d2 = 0.0;
27. public void run(int first, int last){
28. for(int j=first; j<last; j++)
29. d2 += a[j]^2*colidx(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. System.out.println(e);
42. }
43.
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.   }
33.   public void finish(){
34.     // adding up all our collapsed d2 sums
35.     sum2.addAndGet(d2);
36.   }
37. }
38. }
39.}
40. catch(Exception e){
41.   e.printStackTrace();
42.}
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 $\lambda$ (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:

$$zeta = \text{shift} + 1/(x \cdot z)$$
Performance

Running Time

Speedup

EDSF

Efficiency
Running Time

Running Time vs. Processors

- N = Class B
- N = Class A

Processors, K

1E0
1E-1
1E-2
1E-3

1
10
100

1E0
1E-1
1E-2
1E-3

1
10
100

N = Class B
N = Class A
Speedup

![Speedup vs. Processors](image)

Speedup (N,K) vs. Processors, K

- N = Class B
- N = Class A

Efficiency
Efficiency

Efficiency vs. Processors

Efficiency vs. Processors
The BT Benchmark

Implementation

Performance
Implementation

The BT benchmark solves a system of 5 non-linear partial differential equations (PDEs). These PDEs are modified from the original system of equations in order to make them suitable for benchmarking.

Changes made to the array structure of the CR5 benchmarks

Problem Size

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 situations performed in computing the x, y and z directional sweeps as well as the time step. Table 12 gives the problem sizes defined for 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.
\[
\frac{\partial U}{\partial \tau} = \frac{\partial E(U)}{\partial \xi} + \frac{\partial F(U)}{\partial \eta} + \frac{\partial G(U)}{\partial \zeta}
+ \frac{\partial T(U, U_\xi)}{\partial \xi} + \frac{\partial V(U, U_\eta)}{\partial \eta} + \frac{\partial W(U, U_\zeta)}{\partial \zeta}
+ H(U, U_\xi, U_\eta, U_\zeta), \quad (\tau, \xi, \eta, \zeta) \in D_\tau \times D
\]
From i to maxSteps
I. Initialize RHS
II. Solve the system of linear equations for U1 along the x-axis
\[
(I - \Delta \tau [D_x(A)^n + D_x^2(N)^n])\Delta U_1 = \text{RHS}
\]
III. Solve the system of linear equations for U2 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 U3 along the z-axis
\[
(I - \Delta \tau [D_z(C)^n + D_z^2(S)^n])\Delta U^n = \Delta U_2
\]
V. Update the solution for the current time-step
\[
[U^{n+1}]_{i,j,k} = [U^n]_{i,j,k} + [\Delta U^n]_{i,j,k}, \quad \text{for } (i, j, k) \in D_h
\]
Problem Sizes

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>
<thead>
<tr>
<th></th>
<th>Problem size</th>
<th>Time-step</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>12 x 12 x 12</td>
<td>.01</td>
</tr>
<tr>
<td>W</td>
<td>24 x 24 x 24</td>
<td>.0008</td>
</tr>
<tr>
<td>A</td>
<td>64 x 64 x 64</td>
<td>.0008</td>
</tr>
<tr>
<td>B</td>
<td>102 x 102 x 102</td>
<td>.0003</td>
</tr>
</tbody>
</table>
Changes made to the array structure of the CFG benchmarks

OpenMP
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:
forcing = new double[5*(IMAX/2*2+1)*(JMAX/2*2+1)*KMAX];
u = new double[5*(IMAX/2*2+1)*(JMAX/2*2+1)*KMAX];
rhs = new double[5*(IMAX/2*2+1)*(JMAX/2*2+1)*KMAX];
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
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;
                }
            }
        }
    }
});//

... ... ...

for (i = isize - 1; i >= 0; i--) {
    for (m = 0; m <= BLOCK_SIZE - 1; m++) {
        for (n = 0; n <= BLOCK_SIZE - 1; n++) {
            rhs[m + i * isize2 + j * jsize2 + k * ksize2] -= lhs[m + n * isize4 + cc * jsize4 + i * ksize4] * rhs[n + (i + 1) * isize2 + j * jsize2 + k * ksize2];
        }
    }
}
Performance
The SP Benchmark

Implementation

Performance

ARRAY ACCESS OPTIMIZATION

- In Fortran, many of the arrays are optimized and transformed by default by the Intel Fortran compiler
- Parallel Java implementation of SP uses the Fortran arrays. This implementation however, bears a tradeoff in the computation done in order to get the value of the index.
Implementation

The SP problem is solved: the solution of multiple, independent systems of nonlinear elliptic scalar partial differential equations. This is a computationally intensive CPU application much like FT. The threads are mapped appropriately. Data is sent to threads, a system of equations is sent to the 


dimension 

The SP problem is solved: the solution of multiple, independent systems of nonlinear elliptic scalar partial differential equations. This is a computationally intensive GPU application much like FT. The threads are mapped appropriately. Data is sent to threads, a system of equations is sent to the CPU, and coordinates, and a solution is computed along each of the coordinates.

Compute RHS

- compute the reciprocal of density and speed
- No dependencies on previously computed values
- Parallelization is done at the highest loop level.

Solving the equation in the x-direction

The Add function

- Adding all three equations
- All computed values in the rhs array are added to the "U" array
- Straightforward parallelism at the outermost loop
- Contains no reduction or final clauses
SP defines a solution of multiple, independent systems of nondiagonally dominant scalar pentadiagonal equations. 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.
### 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>
<thead>
<tr>
<th>Class Size</th>
<th>grid_points1</th>
<th>grid_points2</th>
<th>grid_points3</th>
<th>niter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>400</td>
<td>400</td>
<td>400</td>
<td>400</td>
</tr>
<tr>
<td>2</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>400</td>
</tr>
<tr>
<td>3</td>
<td>600</td>
<td>600</td>
<td>600</td>
<td>400</td>
</tr>
<tr>
<td>4</td>
<td>700</td>
<td>700</td>
<td>700</td>
<td>400</td>
</tr>
<tr>
<td>5</td>
<td>800</td>
<td>800</td>
<td>800</td>
<td>400</td>
</tr>
</tbody>
</table>
the values of the variables for the 5 different

gest

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
| S | problem_size=12  
  grid_points[0]=12  
  grid_points[1]=12  
  grid_points[2]=12  
  dt_default = .015  
  niter_default = 400 | W | problem_size=36  
  grid_points[0]=36  
  grid_points[1]=36  
  grid_points[2]=36  
  dt_default = .0015  
  niter_default = 400 |
| A | problem_size=64  
  grid_points[0]=64  
  grid_points[1]=64  
  grid_points[2]=64  
  dt_default = .0015  
  niter_default = 400 | B | problem_size=102  
  grid_points[0]=102  
  grid_points[1]=102  
  grid_points[2]=102  
  dt_default = .001  
  niter_default = 400 |
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.
Compute RHS

- compute the reciprocal of density and speed
- No dependencies on previously computed values
- Parallelization is done at the highest loop level.

Solving the problem...

```java
public void x = 1;
if (timeron) {
    try {
        team.execute();
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
```
Solving the equation in the x-direction

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

    try{
        team.execute(new ParallelRegion(){
            public void run() throws Exception {
                execute(1, nz2, new IntegerForLoop(){
                    int i, j, k, n, i1, i2, m;
                    double ru1, fac1, fac2;
                    public void run(int first, int last)
                        for(k=first; k<=last; k++){
                            for(j=1;j<=ny2;j++){
                                ...
                                // first fill the lhs for the u-eigenvalue
                                for(i=0; i<=grid_points[0]-1; i++){
                                    ru1 = c3c4*rho_i[i+j*size2+k*size2];
                                    cv[i] = us[i+j*size2+k*size2];
                                    rhon[i] = dmax1(dx2+c43*ru1, dx5+c1c5*ru1,dxmax+ru1,dx1)
                                }
                            }
                        }
                }
            }
        });
    }
```
public void x_solve()
{
    if (timeron) timer.timerStart(t_xsolve);

    try{
        team.execute(new ParallelRegion());
        public void run() throws Exception {
            execute(1, nz2, new IntegerForLoop())
            int i, j, k, n, i1, i2, m;
            double ru1, fac1, fac2;
            public void run(int first, int last)
            for(k=first; k<=last; k++)
                for(j=1;j<=ny2;j++)
                ...

                //first fill the lhs for the u-eigenvalue
                for(i=0; i<=grid_points[0]-1; i++)
                    ru1=c3c4*rho_i[i+j*size2+k*size2];
                    cv[i] = us[i+j*size2+k*size2];
                    rhon[i] = dmax1(dx2+con43*ru1,
                    dx5+c1c5*ru1,dxmax+ru1,dx1)
    }
The Add function

- Adding all three equations
- All computed values in the rhs array are added to the “U” array
- Straightforward parallelism at the outermost loop
- Contains no reduction or final clauses
Performance

ARRAY ACCESS OPTIMIZATION

- In Fortran, many of the arrays are optimized and transformed by default by the Intel Fortran optimizer
- Parallel Java implementation of SP uses the flattened arrays. This implementation however, bears a tradeoff in the computation done in order to get the value of the index
Performance

ARRAY ACCESS OPTIMIZATION

- In Fortran, many of the arrays are optimized and transformed by default by the Intel Fortran optimizer
- Parallel Java implementation of SP uses the flattened arrays. This implementation however, bears a tradeoff in the computation done in order to get the value of the index
The LU Benchmark

Implementation

Performance
Implementation

LU solves 5 x 5 sparse matrix blocks of lower and upper triangular systems. This is another CPU 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.

FORMING THE LOWER AND UPPER TRIANGULAR MATRICES (STEPS 1 AND 3)
- Form the block diagonal d
- Form the first second and third sub-diagonals, a, b and c
  - Three level deep loops involved in forming the corresponding d, a, b and c matrices
  - Forming these matrices does not require previously computed values from either of them

```plaintext
... implementation code ...
```

Breakdown of the SSOR method
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

Problem Size
- `isize`: defines the number of grid points in each dimension.
- `dt default`: defines the time-step
- `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.

```plaintext
... implementation code ...
```
LU solves 5 X 5 sparse matrix blocks of lower and upper triangular systems. This is another CFD application that solves the Navier-Strokes equation using the symmetric successive over-relaxation (SSOR) method. The Navier-Strokes 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.
Breakdown of the SSOR method

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
Problem Size

- isize: - defines the number of grid points in each dimension.
- dt_default:- defines the time-step
- 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.
Console during iterations.

\texttt{it:-} defines the quantity of time steps.
FORMING THE LOWER AND UPPER TRIANGULAR MATRICES (STEPS 1 AND 3)

- Form the block diagonal d
- Form the first second and third sub-diagonals, a, b and c
  - Three level deep loops involved in forming the corresponding d, a, b and c matrices
- Forming these matrices does not require previously computed values from either of them

```java
    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(i=first; j<=last; j++){
                for(i=ist-1; i<=iend-1; i++){
                    tmp1 = rho_i[i+i]*size3+k*ksize3;
                    tmp2 = tmp1 * tmp1;
                    tmp3 = tmp1 * tmp2;
                    d[0+0*size+i*size+i*size+j*size] = 1.0
                    + dt * 2.0 * (tx1 * dx0 + ty1 * dy0 + tz1 * dz0);
                    d[0+1*size+i*size+i*size+j*size] = 0.0;
                    ...
                    ...
                    ...
    ```
```
```java
    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*size3+k*size3];
                        tmp2 = tmp1 * tmp1;
                        tmp3 = tmp1 * tmp2;
                        d[0+0*isize4+i*size4+j*size4+k*size4] = 1.0
                        + dt * 2.0 * ( tx1 * dx1 + ty1 * dy1 + tz1 * dz1 );
                        d[0+1*isize4+i*size4+j*size4+k*size4] = 0.0;
                    }
                }
            }
        }
    }
```
Future Work

• Restructuring array access to take advantage of further improvements of the Java JIT compiler
• 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.
• 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.
Benchmarking Parallel Java
Conclusion

Parallel Java is by no doubt miles ahead of using the native Java code to do parallelism.

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.

Future Work

• Restructuring array access to take advantage of further improvements of the Java JIT compiler
• 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.
• 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.