EECS 587 Parallel Computing

Term Project Report

Jacobi Algorithm for diagonalization of symmetric matrices: from serial to parallel

Guangsha Shi

67334792

Dec 15, 2013

1. Introduction

Lots of scientific problems, such as eigenvalues for Hamiltonian matrix in quantum mechanics and vibrational analysis of structures with many degrees of freedom in solid mechanics, involve the diagonalization of symmetric matrices for eigenvalues. In real practice, this problem is usually solved by using numerical methods such as Jacobi method and QR method instead of directly from definition, and Jacobi method has been proved to be a perfect choice for symmetric matrices.

If matrix A is an nth-order real symmetric matrix, these must exist an orthogonal matrix so that

$$U^T AU = D$$

in which D is a diagonal matrix,

$$D = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix}$$

where λ_i (i = 1, 2, …, n) are the eigenvalues of matrix A. Jacobi eigenvalue algorithm is an iterative method to calculate the eigenvalues of a real symmetric matrix by a applying a sequence of orthogonal transformations which literately turn the matrix into diagonal form.

Jacobi rotation is an orthogonal transformation that zeros a pair of off-diagonal elements of a real symmetric matrix A,

$$A' = J(p,q)^T AJ(p,q)$$

$$A'_{pq} = A'_{qp} = 0 \ .$$

The orthogonal matrix J(p, q) which eliminates the element A_{pq} and A_{qp} is called the Jacobi rotation matrix. It is equal to an identity matrix except for the four elements with indices pp, pq, qp and qq,

$$J(p,q) = \left[\begin{pmatrix} 1 & & & \cdots & & & 0 \\ & \ddots & & & & & \\ & & cos\phi & \cdots & -sin\phi & & \\ & & \vdots & \ddots & \vdots & \vdots & \\ & & sin\phi & \cdots & cos\phi & & \\ & & & & & \ddots & \\ 0 & & & \cdots & & & 1 \end{pmatrix}\right]$$

Or more explicitly,

$$J(p,q)_{ij} = \delta_{ij} \; \forall \; ij \notin \{pq, qp, pp, qq\}$$

$$J(p,q)_{pp} = J(p,q)_{qq} = cos\phi$$

$$J(p,q)_{pq} = -J(p,q)_{qp} = -sin\phi$$

After a Jacobi rotation, the matrix elements of A' become

$$A'_{ij} = A_{ij} \; \forall \; i \neq p, q \wedge j \; \neq p, q$$

$$A'_{pi} = A'_{ip} = cA_{pi} + s\, A_{qi} \; \forall \; i \neq p, q$$

$$A'_{qi} = A'_{iq} = c\, A_{qi} - sA_{pi} \; \forall \; i \neq p, q$$

$$A'_{pp} = c^2 A_{pp} + 2sc\, A_{pq} + s^2 A_{qq}$$

$$A'_{qq} = s^2 A_{pp} - 2sc\, A_{pq} + c^2 A_{qq}$$

$$A'_{pq} = A'_{qp} = sc\,(A_{qq} - A_{pp}) + (c^2 - s^2)A_{pq}$$

where $c = cos\phi$ and $s = sin\phi$. The angle ϕ is chosen such that the matrix element A'_{pq} and A'_{qp} are zeroed after rotation.

$$A'_{pq} = 0 \;\Rightarrow\; tan(2\phi) = \frac{2A_{pq}}{A_{pp} - A_{qq}} \;\Rightarrow\; \phi = \frac{1}{2} tan^{-1}\left(\frac{2A_{pq}}{A_{pp} - A_{qq}}\right)$$

While the off-diagonal matrix elements with indices pq and qp are annihilated, the other off-diagonal elements may also get changed, as shown in Figure 1. However, after the Jacobi rotation the sum of squares of all off-diagonal elements is reduced by $2A_{pq}^2$ while the sum of squares of all diagonal elements is increased by $2A_{pq}^2$. The algorithm repeatedly applies the Jacobi rotations until all the off-diagonal elements become sufficiently small.

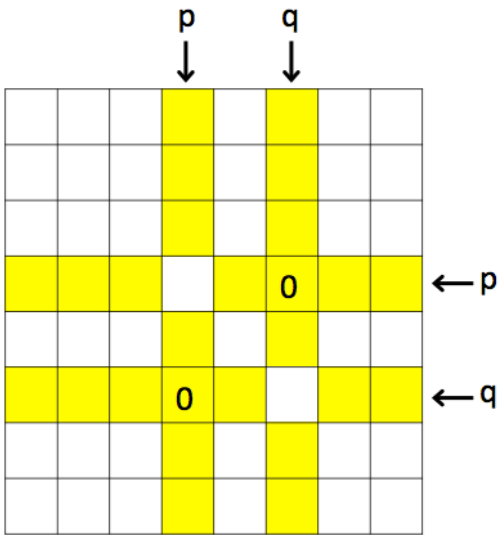

Figure 1. The off-diagonal matrix elements that are changed after Jacobi rotation $J(p,q)$.

2. Serial algorithm

In a symmetric matrix there are $\frac{N^2-N}{2}$ off-diagonal elements to be zeroed, and the annihilation of one off-diagonal element involves the change to 2N-1 elements. Therefore, the complexity of the serial algorithm is $O(n^3)$.

Before we move on to design the parallel algorithm, it would be better to choose the appropriate strategy in serial. So far there are two typical strategies to zero all the off-diagonal elements: classical method and cyclic method. The algorithms are listed as follows and are both implemented in this work (Appendix A and B).

```
// Classical method
while (max > ε)
        Find maximal off-diagonal elements A_pq
        Apply Jacobi rotation J(p,q) to A
end while
```

```
// Cyclic method
finished = false
while (finished == false)
        finished = true
        for each off-diagonal element A_pq
                if (A_pq > ε) then
                        finished = false
                        Apply Jacobi rotation J(p,q)
                end if
        end for
end while
```

The convergence criteria are the same in both the methods, which require that all the off-diagonal elements are smaller than ε. The classical method only applies the Jacobi rotation to the largest of the remaining off-diagonal elements while the cyclic method applies the rotations to each off-diagonal element at each step. Since the classical method always minimizes the sum of squares of all off-diagonal elements at each rotation, it takes fewer rotations to reach convergence. However, the classical method is actually slower than the cyclic method since searching for the largest element becomes dominant and takes $O(n^2)$ before each rotation.

N	50	100	150	200
Classical	4396	18371	42157	75889
Cyclic	6909	33320	79377	154414

Table 1. Comparison of number of rotations with different algorithms for symmetric matrix with size N×N.

N	50	100	150	200
Classical	0.053	0.660	3.223	9.953
Cyclic	0.017	0.110	0.321	0.811

Table 2. Comparison of time consumed (in second) with different algorithms for symmetric matrix with size N×N.

A couple of tests were done to compare the two serial algorithms. The matrix was generated with random numbers from 0 to 10 in uniform distribution for diagonal elements and random numbers from 0 to 1 in uniform distribution for off-diagonal elements. The code for matrix generation is attached in Appendix E. As shown in Table 1 and 2, classical method takes fewer rotations to reach convergence than cyclic method, but it consumes much more time for various inputs. Therefore, cyclic method is usually preferred over classical method and is widely used in numerical computation packages. Recently some semi-classical methods are also being developed to take over from the two traditional methods, which requires fewer rotations than cyclic method by searching for maximal off-diagonal values in small subsets of the original matrix and is claimed to be more efficient than both the classical and cyclic methods. These semi-classical methods will not be covered in this work, and our parallel algorithm would originate only from the two traditional serial algorithms.

3. Parallel algorithm

To parallelize either of the two serial algorithms, it would be desirable if we could divide all the M rotations evenly to P processors. However, the problem is, we have forward dependency in both of the two serial algorithms, which means the rotation J(p, q) to apply at each step always depends on the matrix got from the previous rotation. There is really little we can do for the classical method, since the search for the maximal off-diagonal matrix element always requires the information of the latest updated matrix. For cyclic method, which zeroed each off-diagonal matrix element in strict order (e.g. row after row), it is actually possible to do some parallelization.

It has been shown that a Jacobi rotation J(p, q) is only determined by the matrix elements with indices pp, qq and pq, and affects only the elements in the columns p, q and rows p, q, as shown in Figure 2. Therefore, two rotations J(p, q) and J(m, n) could be performed in parallel as long as neither m nor n is equal to p or q, as shown in Figure 3, since the input of rotation J(p, q) does not depend on the output of rotation J(m, n) and the input of J(m, n) does not depend on the output of J(p, q).

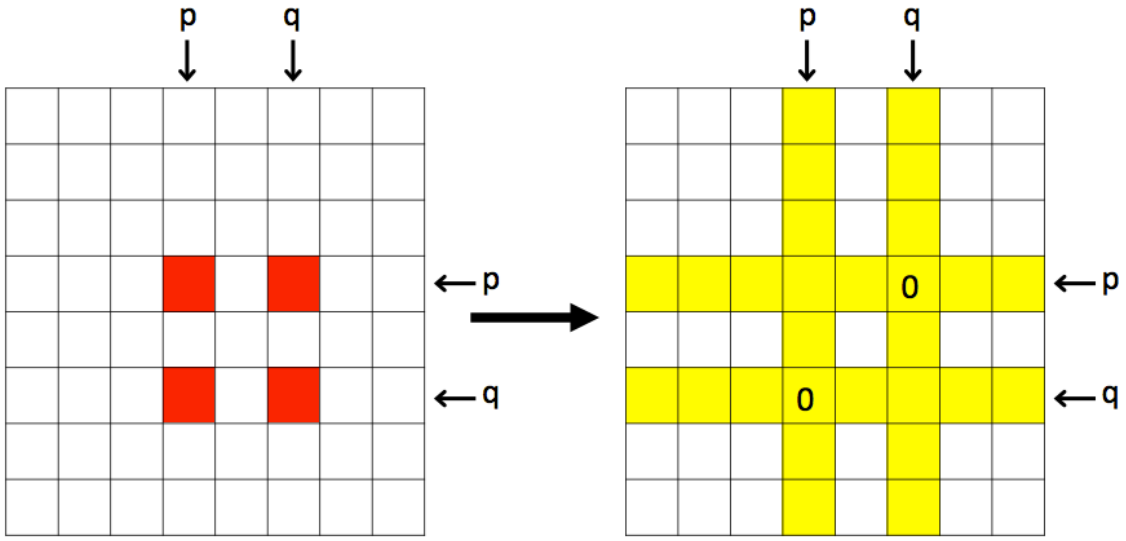

Figure 2. When a Jacobi rotation is applied to matrix, only matrix elements with indices pp, qq, pq and qp (red) are used as input, and only the elements in the columns p, q and row p, q (yellow) will be updated.

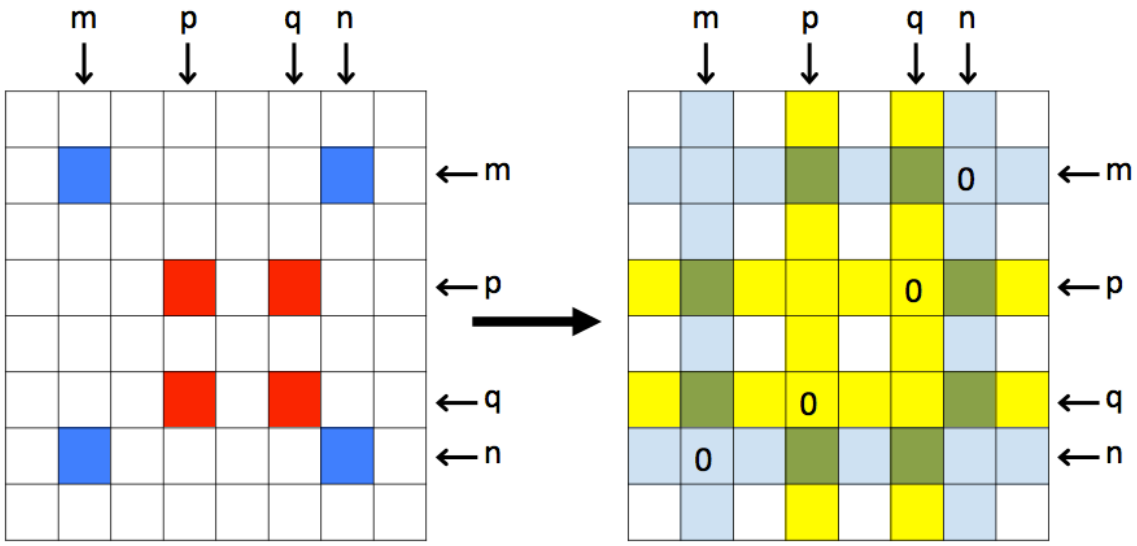

Figure 3. The two Jacobi rotations J(p, q) and J(m, n) could be applied in parallel because the input for J(p, q) (red, left figure) has no overlap with the output of J(m, n) (light blue and green, right figure) and similarly the input for J(m, n) (blue, left figure) has no overlap with the output of J(p, q) (yellow and green, right figure).

However, there are some overlapping elements in the output of rotation J(p, q) and J(m, n), shown in green in Figure 3. These elements, different from those in light blue or yellow, should be updated twice, with both J(p, q) and J(m, n). It could be shown that it is equivalent to update the overlapping elements with J(p, q) first or with J(m, n) first. For example, for the element with index pn (row p and column n), if we apply rotation J(p, q) first and then J(m, n),

$$A'_{np} = c_1 A_{pn} + s_1 A_{qn}$$

$$A'_{mp} = c_1 A_{pm} + s_1 A_{qm}$$

$$(1)\ A''_{pn} = c_2 A'_{np} + s_2 A'_{mp} = c_2 c_1 A_{pn} + c_2 s_1 A_{qn} + s_2 c_1 A_{pm} + s_2 s_1 A_{qm}$$

and if we apply rotation J(m, n) first and then J(p, q),

$$A'_{pn} = c_2 A_{pn} + s_2 A_{pm}$$

$$A'_{qn} = c_2 A_{qn} + s_2 A_{qm}$$

$$(2)\ A''_{pn} = c_1 A'_{pn} + s_1 A'_{qn} = c_1 c_2 A_{pn} + c_1 s_2 A_{pm} + s_1 c_2 A_{qn} + s_1 s_2 A_{qm}$$

and equation (1) and (2) are equivalent to each other. For this reason, for any two rotations J(p, q) and J(m, n) with p, q, m, n all distinct, the work could be distributed to two processors as shown in Figure 4. Each processor updates the corresponding rows first, and then exchanges the information of rotation parameters to update the corresponding elements in each row. This procedure also holds for more than two Jacobi rotations.

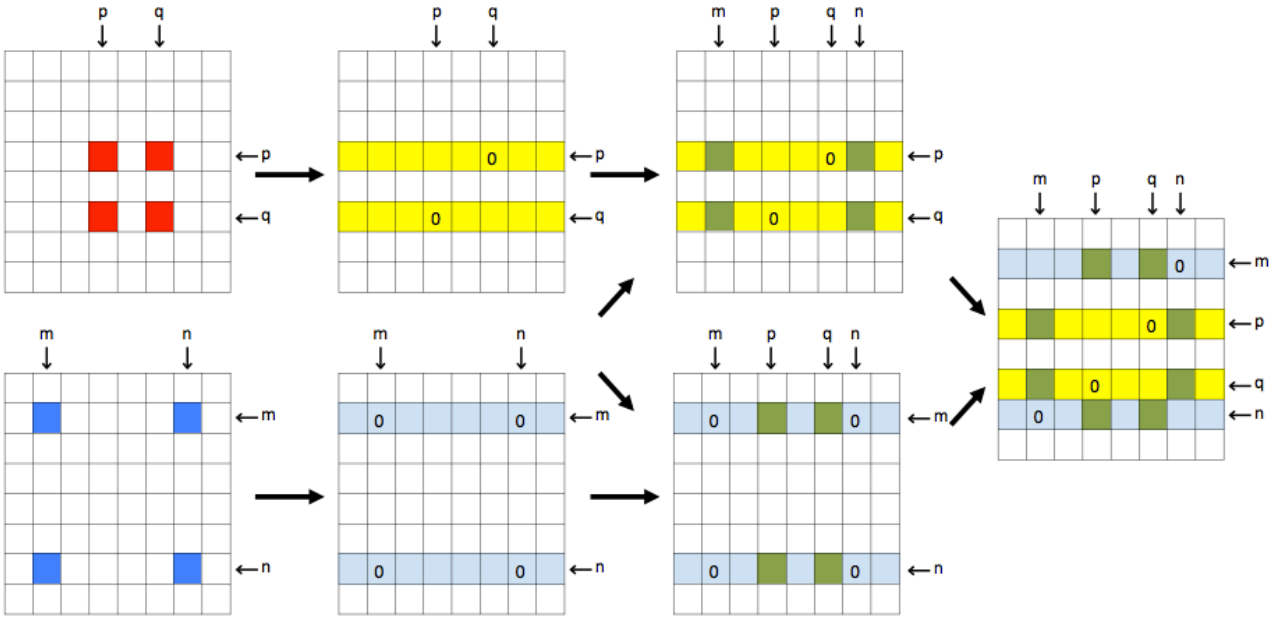

Figure 4. Example of two Jacobi rotations being done in parallel. Each processor updates the corresponding rows first, and then exchanges the information of rotation parameters to update the corresponding elements in each row.

Cyclic method zeros all the off-diagonal elements in order in each turn, which takes $\frac{N^2-N}{2}$ rotations. However, only $\frac{N}{2}$ rotations could be done in parallel at a time to avoid conflicting rotations. For example, rotations J(1, 2) and J(3, 4) could be done in parallel but J(1, 2) and J(1, 3) could not. Therefore, the best we could do is to divide one turn of $\frac{N^2-N}{2}$ rotations into N-1 steps with $\frac{N}{2}$ rotations done in parallel in each step. To maximize concurrency, the rotation ordering shown in Figure 5 is used to update $\frac{N}{2}$ rotation pairs in each of the N-1 steps. Parallel cyclic method was implemented with OpenMP in this work.

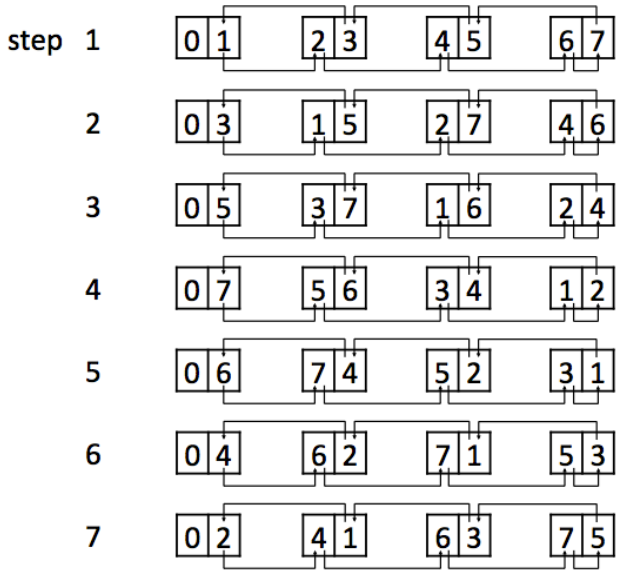

Figure 5. The rotation ordering used to update 4 rotation pairs in each of the 7 steps for an 8×8 symmetric matrix. For a 7×7 matrix (with odd number of elements), we still have 7 steps but replace index 7 with -1 in each step. Rotation pairs including -1 will be skipped later.

While the convergence criterion is not satisfied, one more turn of $\frac{N^2-N}{2}$ rotations is applied to the matrix. The $\frac{N^2-N}{2}$ rotations are divided into N-1 steps with $\frac{N}{2}$ rotations done in parallel in each step. The $\frac{N}{2}$ rotations are distributed to P cores and each core updates the rows p, q corresponding to the rotations J(p, q) distributed to it. After all the $\frac{N}{2}$ rotations are applied to the corresponding rows, the $\frac{N}{2}$ rotations are distributed again to P cores and for each rotation J(p, q), the elements A_{ip} and A_{iq} are updated for each row $i \neq p, q$. The parallel algorithm is shown in the following block and the code is attached in Appendix C. The code is tested for matrices with size 500×500, 1000×1000 and 1500×1500. The elapsed time, speed up and efficiency with various numbers of processors are shown in Table 3, 4 and 5. Though the increase of efficiency is observed with increasing matrix size, the efficiency drops dramatically as the number of cores increases. One cause is, although $\frac{N}{2}$ rotations are distributed to P cores in each step, different steps still have forward dependency since step i always has to be done based on the result from step i-1. Implicit barriers at the end of the second ‘omp for’ section were

used to separate different steps completely to prevent multiple threads from writing to the same memory unit at the same time. Another cause is, all the elements in row p and row q are (i) first updated by applying rotation J(p, q) and

(ii) then updated by all the other rotations.

Though both (i) amd (ii) can be done individually in parallel, implicit barriers at the end of the first 'omp for' section were used to separate operation (i) and (ii) to avoid any data race.

```
// Original cyclic method with OpenMP
finished = false
while (finished == false)
        finished = true
        for N - 1 steps
                omp for
                for N/2 rotations
                        if (A_pq > ε) then
                                finished = false
                                Apply Jacobi rotation J(p,q) to row p, q
                        end if
                end for
                omp for
                for N/2 rotations
                        for i = row 0 to row N-1
                                Apply Jacobi rotation J(p,q) to A_ip and A_iq
                        end for
                end for
        end for
end while
```

NP	1	2	4	6	8	10	12
Time (s)	15.6	8.78	5.30	6.27	5.19	4.62	4.20
Speedup	1	1.77	2.94	2.48	3.00	3.38	3.71
Efficiency	1	0.887	0.735	0.414	0.376	0.338	0.309

Table 3. Elapsed time, speed up and efficiency with various numbers of processors for a 500×500 symmetric matrix.

NP	1	2	4	6	8	10	12
Time (s)	164	85.7	68.9	50.6	40.8	33.2	29.0
Speedup	1	1.91	2.38	3.24	4.02	4.94	5.66
Efficiency	1	0.957	0.595	0.540	0.502	0.494	0.471

Table 4. Elapsed time, speed up and efficiency with various numbers of processors for a 1000×1000 symmetric matrix.

NP	1	2	4	6	8	10	12
Time (s)	681	330	234	176	141	112	100
Speedup	1	2.06	2.91	3.88	4.84	6.08	6.81
Efficiency	1	1.03	0.727	0.646	0.606	0.608	0.568

Table 5. Elapsed time, speed up and efficiency with various numbers of processors for a 1500×1500 symmetric matrix.

It is found necessary to have barriers at the end of each step since the forward dependency between different steps cannot be avoided. However, for the barriers between operation (i) and (ii), I noticed that there was actually something I could do to avoid using them.

When N/2 rotations are evenly distributed to P cores, some cores may finish operation (i) faster than the others. The cores that finish their work earlier have to stay idle until all the cores finish operation (i) due to the existence of barrier at the end of for loop. The improvement of load balance is very little when it is changed from static to dynamic schedule, which may be due to the significant overhead. However, it is noticed that we could actually create some partial overlapping of operation (i) and (ii) without any data race. When a processor finishes applying the Jacobi rotations J(p,q) to all the row pairs (p, q), instead of waiting for the other processors to finish their work, it could go directly to do operation (ii). More explicitly, this processor looks around to see if any other processor has finished any rotation in operation (i). Whenever it detects that another processors finishes rotation J(p, q), it applies J (p, q) to all the rows this processor is responsible for. The rotations that have been applied in operation (i) are pushed into a

container which could be accessed by all the processors. Each core finishes operation (i) first, pushes the finished Jacobi rotations into container, and then keeps applying all the other rotations pushed into container until the container size gets equal to N/2. The improved parallel algorithm is shown in the following block and the code is attached in Appendix D.

```
// Improved parallel cyclic method with OpenMP
finished = false
start = ⌈ tid * N/2 / P ⌉;
end = ⌈ (tid + 1) * N/2 / P ⌉ - 1;
while (finished == false)
        finished = true
        for N - 1 steps
                Clear container C
                for i = rotation start to end
                        if (A_pq > ε) then
                                finished = false
                                Apply Jacobi rotation J(p,q) to row p, q
                                Push J(p,q) to container C
                        end if
                end for
                count = 0
                while (count < N/2 and count < C.size)
                        J(p,q) = C[count]
                        for i = rotation start to end
                                m, n = rows corresponding to rotation i
                                Apply Jacobi rotation J(p,q) to A_mp, A_mq, A_np and A_nq
                        end for
                        count = count + 1
                end while
                omp barrier
        end for
end while
```

The code is also tested for matrices with size 500×500, 1000×1000 and 1500×1500. The elapsed time, speed up and efficiency with various numbers of processors are shown in Table 6, 7 and 8. Compared to the original parallel algorithm, my improved algorithm gives much better speedup and efficiency, especially for the 1500×1500 matrix. Figure 6,

7 and 8 show the comparison of elapsed time, speedup and efficiency between the original and improved parallel cyclic algorithm for a 1000×1000 symmetric matrix.

NP	1	2	4	6	8	10	12
Time (s)	19.7	10.0	6.17	4.27	3.57	3.09	2.83
Speedup	1	1.97	3.19	4.61	5.52	6.37	6.97
Efficiency	1	0.983	0.798	0.769	0.690	0.637	0.581

Table 6. Elapsed time, speed up and efficiency with various numbers of processors for a 500×500 symmetric matrix.

NP	1	2	4	6	8	10	12
Time (s)	183	92.5	49.8	33.7	25.7	21.2	18.0
Speedup	1	1.98	3.67	5.42	7.11	8.61	10.1
Efficiency	1	0.988	0.917	0.903	0.889	0.861	0.845

Table 7. Elapsed time, speed up and efficiency with various numbers of processors for a 1000×1000 symmetric matrix.

NP	1	2	4	6	8	10	12
Time (s)	716	364	185	125	92.4	75.7	63.3
Speedup	1	1.97	3.88	5.72	7.75	9.46	11.3
Efficiency	1	0.983	0.970	0.954	0.968	0.946	0.942

Table 8. Elapsed time, speed up and efficiency with various numbers of processors for a 1500×1500 symmetric matrix.

The improved algorithm requires some extra time to write and read the container to keep record of the finished Jacobi rotations, so it takes a little more time than the original one when the number of cores in small. We minimizes the idle time of each core by allowing the partial overlapping of operation (i) and (ii) and maximizing the concurrency. The benefit from this modification becomes significant when the number of cores increases. For example, the efficiency of the original parallel algorithm drops from 1 to 0.6 with

number of cores increased from 1 to 12, while the efficiency of the improved parallel algorithm never goes below 0.94, as shown in Figure 8. As the matrix size grows larger, the efficiency gets better as shown in Table 6, 7 and 8, which is similar to the trend with original parallel algorithm. The little oscillations in the curves and efficiency should be due to the instability of the Flux system.

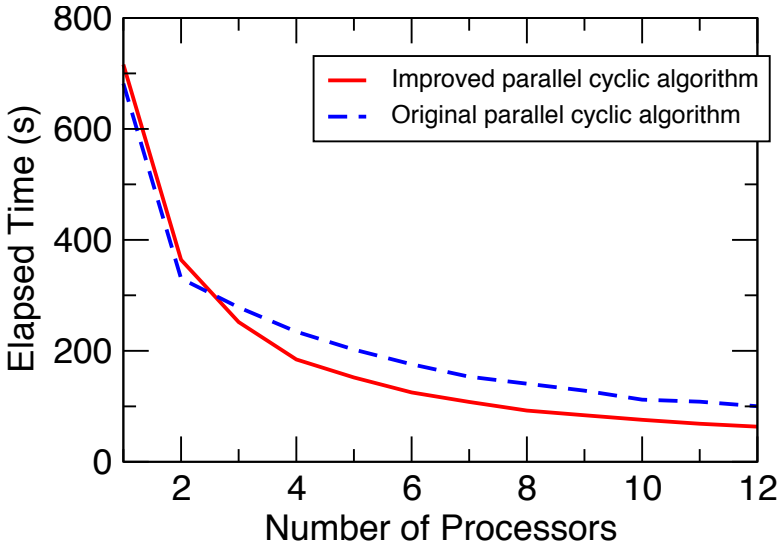

Figure 6. Comparison of elapsed time between original and improved parallel cyclic algorithm for a 1500×1500 symmetric matrix.

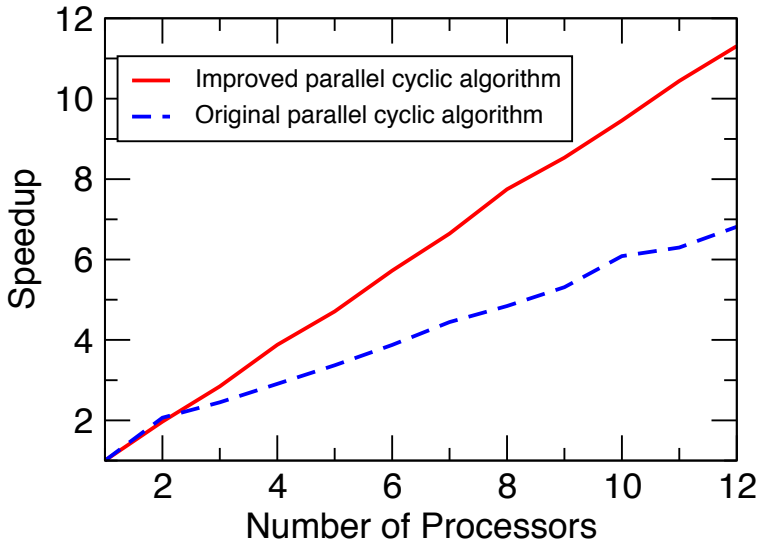

Figure 7. Comparison of speedup between original and improved parallel cyclic algorithm for a 1500×1500 symmetric matrix.

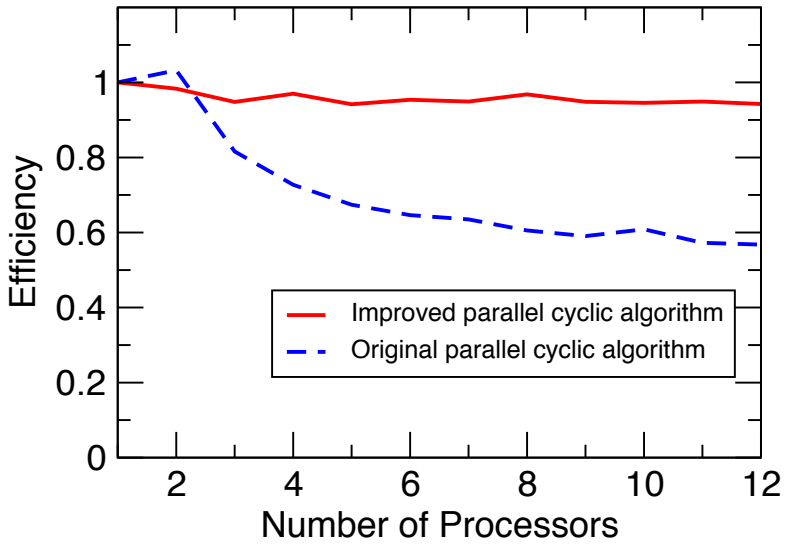

Figure 8. Comparison of efficiency between original and improved parallel cyclic algorithm for a 1500×1500 symmetric matrix.

4. Conclusion

I implemented two parallel Jacobi algorithms in this work for diagonalization of symmetric matrices with OpenMP. While the original parallel algorithm loses efficiency with more processors, the improved parallel algorithm keeps almost a linear speed with up to 12 cores by increasing the concurrency.

For future work, I plan to implement a hybrid MPI + OpenMP code so that the code would be well parallelized on more than 12 cores. It is reported by the previous investigators that the parallel cyclic Jacobi algorithm implemented with MPI typically has low efficiency due to the large amount of communication. Thus, I suppose a better performance could be achieved with a hybrid method than pure MPI.

5. Reference

1. Singer, S., Novaković, V., Uščumlić, A., & Dunjko, V. (2012). Novel modifications of parallel Jacobi algorithms. Numerical algorithms, 59(1), 1-27.
2. Gao, G. R., & Thomas, S. J. (1988, August). An optimal parallel Jacobi-like solution method for the singular value decomposition. In Proc. Internat. Conf. Parallel Proc (pp. 47-53).
3. Gotze, J., Paul, S., & Sauer, M. (1993). An efficient Jacobi-like algorithm for parallel eigenvalue computation. Computers, IEEE Transactions on, 42(9), 1058-1065.
4. Kempen, H. P. (1966). On the quadratic convergence of the special cyclic Jacobi method. Numerische Mathematik, 9(1), 19-22.
5. Hansen, E. R. (1963). On cyclic Jacobi methods. Journal of the Society for Industrial & Applied Mathematics, 11(2), 448-459.
6. Henrici, P. (1958). On the speed of convergence of cyclic and quasicyclic Jacobi methods for computing eigenvalues of Hermitian matrices. Journal of the Society for Industrial & Applied Mathematics, 6(2), 144-162.

Appendix A

```
////////////////////////////////////////////////////
// Classical Jacobi Algorithm for diagonalization
// of symmetric matrices (serial)
////////////////////////////////////////////////////

#include <iostream>
#include <cmath>

using namespace std;

void rotate( int p, int q, double **matrix, int size, double epsilon )
{
    double theta, SinTheta, CosTheta;
    double *rowp = new double[size];

    // Compute the orthogonal matrix for Jacobi rotation
    theta = atan( 2 * matrix[p][q] / ( matrix[p][p] - matrix[q][q] ) )
/ 2;
    SinTheta = sin(theta);
    CosTheta = cos(theta);

    // Compute the off-diagonal element in the new matrix
    rowp[p] = matrix[p][p] * CosTheta * CosTheta
    + matrix[q][q] * SinTheta * SinTheta
    + 2 * SinTheta * CosTheta * matrix[p][q];
    matrix[q][q] = matrix[q][q] * CosTheta * CosTheta
    + matrix[p][p] * SinTheta * SinTheta
    - 2 * SinTheta * CosTheta * matrix[p][q];
    rowp[q] = matrix[q][p] = 0;

    for ( int i = 0; i < size; i++ )
    {
        if ( ( i != p ) && ( i != q ) )
        {
            rowp[i] = CosTheta * matrix[p][i] + SinTheta *
matrix[q][i];
            matrix[q][i] = CosTheta * matrix[q][i] - SinTheta *
matrix[p][i];
        }
    }

    // Update the matrix
    for ( int i = 0; i < size; i++ )
    {
        matrix[i][p] = matrix[p][i] = rowp[i];
        matrix[i][q] = matrix[q][i];
    }

    delete [] rowp;
}

void jacobi( double **matrix, int size, double epsilon )
{
    double** NewMatrix = new double*[size];
```

```
    double max, theta, SinTheta, CosTheta;
    int p, q;
    // First find the largest off-diagonal element
    max = matrix[0][1]; // row = 0, column = 1
    p = 0;
    q = 1;
    for ( int i = 0; i < size; i++ )
    {
        for ( int j = i + 1; j < size; j++ )
        {
            if ( abs( matrix[i][j] ) > max )
            {
                max = abs( matrix[i][j] );
                p = i;
                q = j;
            }
        }
    }

    while ( max > epsilon )
    {
        rotate( p, q, matrix, size, epsilon );

        // Find the largest off-diagonal element
        max = matrix[0][1]; // row = 0, column = 1
        p = 0;
        q = 1;
        for ( int i = 0; i < size; i++ )
        {
            for ( int j = i + 1; j < size; j++ )
            {
                if ( abs( matrix[i][j] ) > max )
                {
                    max = abs( matrix[i][j] );
                    p = i;
                    q = j;
                }
            }
        }
    }
    cout << sum << endl;
}


int main(int argc,char** argv)
{
    int size;
    double epsilon;

    cin >> size >> epsilon;

    double** matrix = new double*[size];

    for ( int i = 0; i < size; i++ )
    {
        matrix[i] = new double[size];
        for ( int j = 0; j < size; j++ )
```

```
            {
                cin >> matrix[i][j];
            }
        }

        jacobi( matrix, size, epsilon );

        // Print the matrix
        for ( int i = 0; i < size; i++ )
        {
            for ( int j = 0; j < size; j++ )
            {
                cout << matrix[i][j] << "    ";
            }
            cout << endl;
        }

        // Free memory
        for ( int i = 0; i < size; i++ )
        {
            delete [] matrix[i];
        }
        delete [] matrix;

        return 0;
    }
```

Appendix B

```
//////////////////////////////////////////////////
// Cyclic Jacobi Algorithm for diagonalization
// of symmetric matrices (serial)
//////////////////////////////////////////////////

#include <iostream>
#include <cmath>

using namespace std;

void rotate( int p, int q, double **matrix, int size, double epsilon )
{
    double theta, SinTheta, CosTheta;
    double *rowp = new double[size];

    // Compute the orthogonal matrix for Jacobi rotation
    theta = atan( 2 * matrix[p][q] / ( matrix[p][p] - matrix[q][q] ) )
/ 2;
    SinTheta = sin(theta);
    CosTheta = cos(theta);

    // Compute the off-diagonal element in the new matrix
    rowp[p] = matrix[p][p] * CosTheta * CosTheta
    + matrix[q][q] * SinTheta * SinTheta
    + 2 * SinTheta * CosTheta * matrix[p][q];
    rowp[q] = matrix[q][p] = 0;
    matrix[q][q] = matrix[q][q] * CosTheta * CosTheta
    + matrix[p][p] * SinTheta * SinTheta
    - 2 * SinTheta * CosTheta * matrix[p][q];

    for ( int i = 0; i < size; i++ )
    {
        if ( ( i != p ) && ( i != q ) )
        {
            rowp[i] = CosTheta * matrix[p][i] + SinTheta *
matrix[q][i];
            matrix[i][q] = matrix[q][i] = CosTheta * matrix[q][i] -
SinTheta * matrix[p][i];
        }
    }
    // Update the matrix
    for ( int i = 0; i < size; i++ )
    {
        matrix[i][p] = matrix[p][i] = rowp[i];
    }

    delete [] rowp;
}

void jacobi( double **matrix, int size, double epsilon )
{
    double max, theta, SinTheta, CosTheta;
    int p, q;
```

```
    bool finished = false;
    int sum = 0;
    while ( !finished )
    {
        finished = true;
        for ( int p = 0; p < size; p++ )
            for ( int q = p + 1; q < size; q++ )
            {
                if ( abs( matrix[p][q] ) > epsilon )
                {
                    finished = false;
                    rotate( p, q, matrix, size, epsilon );
                }
            }
    }
}


int main(int argc,char** argv)
{
    int size;
    double epsilon;

    cin >> size >> epsilon;

    double** matrix = new double*[size];

    for ( int i = 0; i < size; i++ )
    {
        matrix[i] = new double[size];
        for ( int j = 0; j < size; j++ )
        {
            cin >> matrix[i][j];
        }
    }

    jacobi( matrix, size, epsilon );

    // Print the matrix
    for ( int i = 0; i < size; i++ )
    {
        for ( int j = 0; j < size; j++ )
        {
            cout << matrix[i][j] << "   ";
        }
        cout << endl;
    }

    // Free memory
    for ( int i = 0; i < size; i++ )
    {
        delete [] matrix[i];
    }
    delete [] matrix;

    return 0;
}
```

Appendix C

```
////////////////////////////////////////////////////
// Cyclic Jacobi Algorithm for diagonalization
// of symmetric matrices (parallel)
////////////////////////////////////////////////////

#include <iostream>
#include <cmath>
#include "omp.h"

using namespace std;

struct RotationParameter
{
    bool applied;
    int p;
    int q;
    double SinTheta;
    double CosTheta;
};

void RotateMore( double **matrix, int size, int index,
RotationParameter *RotationParameterList )
{
    int p, q;
    double SinTheta, CosTheta;
    double *ColumnP = new double[size];

    p = RotationParameterList[index].p;
    q = RotationParameterList[index].q;
    SinTheta = RotationParameterList[index].SinTheta;
    CosTheta = RotationParameterList[index].CosTheta;

    for ( int i = 0; i < size; i++ )
    {
        if ( ( i != p ) && ( i != q ) )
        {
            ColumnP[i] = CosTheta * matrix[i][p] + SinTheta *
matrix[i][q];
            matrix[i][q] = CosTheta * matrix[i][q] - SinTheta *
matrix[i][p];
        }
    }

    for ( int i = 0; i < size; i++ )
    {
        if ( ( i != p ) && ( i != q ) )
        {
            matrix[i][p] = ColumnP[i];
        }
    }

    delete [] ColumnP;
}
```

```
void rotate( int p, int q, double **matrix, int size, double epsilon,
int index,
            RotationParameter *RotationParameterList )
{
    double theta, SinTheta, CosTheta;
    double *rowp = new double[size];
    double *rowq = new double[size];

    // Compute the orthogonal matrix for Jacobi rotation
    theta = atan( 2 * matrix[p][q] / ( matrix[p][p] - matrix[q][q] ) )
/ 2;
    RotationParameterList[index].SinTheta = SinTheta = sin(theta);
    RotationParameterList[index].CosTheta = CosTheta = cos(theta);
    RotationParameterList[index].p = p;
    RotationParameterList[index].q = q;
    RotationParameterList[index].applied = true;

    // Compute the off-diagonal element in the new matrix
    rowp[p] = matrix[p][p] * CosTheta * CosTheta
    + matrix[q][q] * SinTheta * SinTheta
    + 2 * SinTheta * CosTheta * matrix[p][q];
    rowq[q] = matrix[q][q] * CosTheta * CosTheta
    + matrix[p][p] * SinTheta * SinTheta
    - 2 * SinTheta * CosTheta * matrix[p][q];
    rowp[q] = rowq[p] = 0;

    for ( int i = 0; i < size; i++ )
    {
        if ( ( i != p ) && ( i != q ) )
        {
            rowp[i] = CosTheta * matrix[p][i] + SinTheta *
matrix[q][i];
            rowq[i] = CosTheta * matrix[q][i] - SinTheta *
matrix[p][i];
        }
    }

    // Update the matrix
    for ( int i = 0; i < size; i++ )
    {
        matrix[p][i] = rowp[i];
        matrix[q][i] = rowq[i];
    }

    delete [] rowp;
    delete [] rowq;
}

int main(int argc,char** argv)
{
    int size, ListSize, NumberOfStep, NumberOfCombination,
NumberOfPair;
    double epsilon;
    bool finished = false;
    double t_begin, t_end;

    cin >> size >> epsilon;
```

```
    double** matrix = new double*[size];

    for ( int i = 0; i < size; i++ )
    {
        matrix[i] = new double[size];
        for ( int j = 0; j < size; j++ )
        {
            cin >> matrix[i][j];
        }
    }

    if ( size % 2 == 0 )
    {
        ListSize = size;
    }
    else
    {
        ListSize = size + 1;
    }
    NumberOfPair = ListSize / 2;

    NumberOfCombination = 1;
    for ( int i = ListSize - 1; i <= ListSize; i++ )
    {
        NumberOfCombination *= i;
    }
    NumberOfCombination /= 2;
    NumberOfStep = ListSize - 1;

    int** ListOfIndices = new int*[NumberOfStep];

    ListOfIndices[0] = new int[ListSize];
    for ( int j = 0; j < size; j++ )
    {
        ListOfIndices[0][j] = j;
    }
    if ( size % 2 == 1 )
    {
        ListOfIndices[0][ListSize - 1] = -1;
    }

    for ( int i = 1; i < NumberOfStep; i++ )
    {
        ListOfIndices[i] = new int[ListSize];
        ListOfIndices[i][0] = 0;
        for ( int j = 1; j < ListSize - 2; j = j + 2 )
        {
            ListOfIndices[i][j] = ListOfIndices[i - 1][j + 2];
        }
        ListOfIndices[i][ListSize - 1] = ListOfIndices[i - 1][ListSize
- 2];

        for ( int j = ListSize - 2; j > 2; j = j - 2 )
        {
            ListOfIndices[i][j] = ListOfIndices[i - 1][j - 2];
        }
```

```
        ListOfIndices[i][2] = ListOfIndices[i - 1][1];
    }

    RotationParameter* RotationParameterList = new
RotationParameter[NumberOfPair];

    omp_lock_t mylock;
    omp_init_lock(&mylock);

#pragma omp parallel default(shared)
    {
#pragma omp single
        {
            t_begin = omp_get_wtime();
        }
        while ( !finished )
        {
#pragma omp barrier
            finished = true;
            for ( int i = 0; i < NumberOfStep; i++ )
            {

#pragma omp for schedule(guided)
                for ( int j = 0; j < NumberOfPair; j++ )
                {
                    int p = ListOfIndices[i][j * 2];
                    int q = ListOfIndices[i][j * 2 + 1];
                    if ( ( p != -1 ) && ( q != -1 ) )
                    {
                        if ( abs( matrix[p][q] ) > epsilon )
                        {
                            finished = false;
                            // This rotate fxn only applies to row p
and row q
                            // but not to column p and column q
                            rotate( p, q, matrix, size, epsilon, j,
RotationParameterList );
                        }
                        else
                        {
                            RotationParameterList[j].applied = false;
                        }
                    }
                    else
                    {
                        RotationParameterList[j].applied = false;
                    }
                }

#pragma omp for schedule(guided)
                for ( int k = 0; k < NumberOfPair; k++ )
                {
                    if ( RotationParameterList[k].applied )
                    {
                        RotateMore( matrix, size, k,
RotationParameterList );
                    }
```

```
                }

            }
        }
#pragma omp single
        {
            t_end = omp_get_wtime();
            printf("Elapsed wall time: %f\n", t_end - t_begin);
        }
    }

    // Print the matrix
    for ( int i = 0; i < size; i++ )
    {
        for ( int j = 0; j < size; j++ )
        {
            cout << matrix[i][j] << "   ";
        }
        cout << endl;
    }

    // Free memory
    omp_destroy_lock(&mylock);

    for ( int i = 0; i < size; i++ )
    {
        delete [] matrix[i];
    }
    delete [] matrix;

    for ( int i = 0; i < NumberOfStep; i++ )
    {
        delete [] ListOfIndices[i];
    }
    delete [] ListOfIndices;

    delete [] RotationParameterList;

    return 0;
}
```

Appendix D

```
////////////////////////////////////////////////////
// Improved cyclic Jacobi Algorithm for
// diagonalization of symmetric matrices (parallel)
////////////////////////////////////////////////////

#include <iostream>
#include <cmath>
#include "omp.h"
#include <vector>

using namespace std;

struct RotationParameter
{
    bool applied;
    int p;
    int q;
    double SinTheta;
    double CosTheta;
};

void RotateMore( double **matrix, int size, int m, RotationParameter
*rp )
{
    int p, q;
    double SinTheta, CosTheta;
    double newmp;

    p = rp->p;
    q = rp->q;
    SinTheta = rp->SinTheta;
    CosTheta = rp->CosTheta;

    newmp = CosTheta * matrix[m][p] + SinTheta * matrix[m][q];
    matrix[m][q] = CosTheta * matrix[m][q] - SinTheta * matrix[m][p];
    matrix[m][p] = newmp;
}

void rotate( int p, int q, double **matrix, int size, RotationParameter
*rp )
{
    double theta, SinTheta, CosTheta;
    double *rowp = new double[size];

    // Compute the orthogonal matrix for Jacobi rotation
    theta = atan( 2 * matrix[p][q] / ( matrix[p][p] - matrix[q][q] ) )
/ 2;
    rp->SinTheta = SinTheta = sin(theta);
    rp->CosTheta = CosTheta = cos(theta);
    rp->p = p;
    rp->q = q;
    rp->applied = true;

    // Compute the off-diagonal element in the new matrix
```

```
    rowp[p] = matrix[p][p] * CosTheta * CosTheta
    + matrix[q][q] * SinTheta * SinTheta
    + 2 * SinTheta * CosTheta * matrix[p][q];
    matrix[q][q] = matrix[q][q] * CosTheta * CosTheta
    + matrix[p][p] * SinTheta * SinTheta
    - 2 * SinTheta * CosTheta * matrix[p][q];
    rowp[q] = matrix[q][p] = 0;

    for ( int i = 0; i < size; i++ )
    {
        if ( ( i != p ) && ( i != q ) )
        {
            rowp[i] = CosTheta * matrix[p][i] + SinTheta *
matrix[q][i];
            matrix[q][i] = CosTheta * matrix[q][i] - SinTheta *
matrix[p][i];
        }
    }

    // Update the matrix
    for ( int i = 0; i < size; i++ )
    {
        matrix[p][i] = rowp[i];
    }

    delete [] rowp;
}

int main(int argc,char** argv)
{
    int size, ListSize, NumberOfStep, NumberOfCombination,
NumberOfPair;
    double epsilon;
    bool finished = false;
    double t_begin, t_end;

    cin >> size >> epsilon;

    double** matrix = new double*[size];

    for ( int i = 0; i < size; i++ )
    {
        matrix[i] = new double[size];
        for ( int j = 0; j < size; j++ )
        {
            cin >> matrix[i][j];
        }
    }

    if ( size % 2 == 0 )
    {
        ListSize = size;
    }
    else
    {
        ListSize = size + 1;
    }
```

```
    NumberOfPair = ListSize / 2;

    NumberOfCombination = 1;
    for ( int i = ListSize - 1; i <= ListSize; i++ )
    {
        NumberOfCombination *= i;
    }
    NumberOfCombination /= 2;
    NumberOfStep = ListSize - 1;

    int** ListOfIndices = new int*[NumberOfStep];

    ListOfIndices[0] = new int[ListSize];
    for ( int j = 0; j < size; j++ )
    {
        ListOfIndices[0][j] = j;
    }
    if ( size % 2 == 1 )
    {
        ListOfIndices[0][ListSize - 1] = -1;
    }

    for ( int i = 1; i < NumberOfStep; i++ )
    {
        ListOfIndices[i] = new int[ListSize];
        ListOfIndices[i][0] = 0;
        for ( int j = 1; j < ListSize - 2; j = j + 2 )
        {
            ListOfIndices[i][j] = ListOfIndices[i - 1][j + 2];
        }
        ListOfIndices[i][ListSize - 1] = ListOfIndices[i - 1][ListSize
- 2];

        for ( int j = ListSize - 2; j > 2; j = j - 2 )
        {
            ListOfIndices[i][j] = ListOfIndices[i - 1][j - 2];
        }
        ListOfIndices[i][2] = ListOfIndices[i - 1][1];

    }

    vector <RotationParameter> RotationParameterList;

    omp_lock_t mylock;
    omp_init_lock(&mylock);

#pragma omp parallel default(shared)
    {
        int NumberOfThread = omp_get_num_threads();
        int tid = omp_get_thread_num();

        int StartPairId = ceil( tid * double(NumberOfPair) /
NumberOfThread );
        int EndPairId = ceil( ( tid + 1 ) * double(NumberOfPair) /
NumberOfThread ) - 1;

#pragma omp single
```

```
        {
            t_begin = omp_get_wtime();
        }
        while ( !finished )
        {
#pragma omp barrier
            finished = true;
            for ( int i = 0; i < NumberOfStep; i++ )
            {
#pragma omp single
                {
                    RotationParameterList.clear();
                }
                for ( int j = StartPairId; j <= EndPairId; j++ )
                {
                    int p = ListOfIndices[i][j * 2];
                    int q = ListOfIndices[i][j * 2 + 1];
                    if ( ( p != -1 ) && ( q != -1 ) )
                    {
                        if ( abs( matrix[p][q] ) > epsilon )
                        {
                            finished = false;
                            // This rotate fxn only applies to row p
and row q
                            // but not to column p and column q
                            RotationParameter rp;
                            rotate( p, q, matrix, size, &rp );
                            omp_set_lock(&mylock);
                            RotationParameterList.push_back(rp);
                            omp_unset_lock(&mylock);
                        }
                        else
                        {
                            RotationParameter rp;
                            rp.applied = false;
                            omp_set_lock(&mylock);
                            RotationParameterList.push_back(rp);
                            omp_unset_lock(&mylock);
                        }
                    }
                    else
                    {
                        RotationParameter rp;
                        rp.applied = false;
                        omp_set_lock(&mylock);
                        RotationParameterList.push_back(rp);
                        omp_unset_lock(&mylock);
                    }
                }
                int count = 0;
                int parametersize = 0;
                while ( count < NumberOfPair )
                {
                    parametersize = RotationParameterList.size();
                    if ( count < parametersize )
                    {
                        if ( RotationParameterList[count].applied )
```

```
                        {
                            for ( int j = StartPairId; j <= EndPairId;
j++ )
                            {
                                int p = ListOfIndices[i][j * 2];
                                int q = ListOfIndices[i][j * 2 + 1];
                                if ( ( p == -1 ) )
                                {
                                    RotateMore( matrix, size, q,
&RotationParameterList[count] );
                                }
                                else if ( ( q == -1 ) )
                                {
                                    RotateMore( matrix, size, p,
&RotationParameterList[count] );
                                }
                                else
                                {
                                    if ( ( q !=
RotationParameterList[count].p )
                                        && ( q !=
RotationParameterList[count].q ) )
                                    {
                                        RotateMore( matrix, size, p,
&RotationParameterList[count] );
                                        RotateMore( matrix, size, q,
&RotationParameterList[count] );
                                    }
                                }
                            }
                        }
                        count++;
                    }
                }
#pragma omp barrier
            }
        }
#pragma omp single
        {
            t_end = omp_get_wtime();
            printf("Elapsed wall time: %f\n", t_end - t_begin);
        }
    }

    // Print the matrix
    for ( int i = 0; i < size; i++ )
    {
        for ( int j = 0; j < size; j++ )
        {
            cout << matrix[i][j] << "   ";
        }
        cout << endl;
    }

    // Free memory
    omp_destroy_lock(&mylock);
```

```
    for ( int i = 0; i < size; i++ )
    {
        delete [] matrix[i];
    }
    delete [] matrix;

    for ( int i = 0; i < NumberOfStep; i++ )
    {
        delete [] ListOfIndices[i];
    }
    delete [] ListOfIndices;

    return 0;
}
```

Appendix E

```
//////////////////////////////////////////////////
// Generate N*N matrix with random numbers from
// 0 to 10 for diagonal elements and random numbers
// from 0 to 1 for off-diagonal elements
//////////////////////////////////////////////////

#include <iostream>
#include <random>

using namespace std;

int main(int argc,char** argv)
{
    int size = atoi(argv[1]);
    int seed = atoi(argv[2]);

    std::mt19937 gen(seed);
    std::uniform_real_distribution<> dis(0, 10);

    double** matrix = new double*[size];
    for ( int i = 0; i < size; i++ )
    {
        matrix[i] = new double[size];
    }

    for ( int i = 0; i < size; i++ )
    {
        matrix[i][i] = dis(gen);
        for ( int j = i + 1; j < size; j++ )
        {
            matrix[j][i] = matrix[i][j] = dis(gen) / 10;
        }
    }

    printf("%d\t%20.15f\n", size, 0.0000000001);
    for ( int i = 0; i < size; i++ )
    {
        for ( int j = 0; j < size; j++ )
        {
            printf("%4.2f\t", matrix[i][j]);
        }
        printf("\n");
    }

    for ( int i = 0; i < size; i++ )
    {
        delete [] matrix[i];
    }
    delete [] matrix;

    return 0;
}
```