CAAM520 Computational Science

Homework 3

AO CAI
S01255664
Earth Science Department
Rice University
March 5, 2019

Contents

1 Introduction

The Goal is to solve the matrix system $Au = b$ resulting from an $(N+2) \times (N+2)$ 2D finite difference method for Laplace's equation $-\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) = f(x, y)$ on $[-1, 1]^2$. At each point (x_i, y_i), derivatives are approximated by:

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}, \frac{\partial^2 u}{\partial y^2} \approx \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h^2}$$

where $h = 2/(N+1)$. Assuming zero boundary conditions $u(x, y) = 0$ reduce this to an $N \times N$ system for the interior nodes.

2 Weighted Jacobi method with OPENMP

To implement the parallel computing for the Poisson's equaion, I used the parallel for function in the openmp. First I compute the off-diagonal FD operators by using:

$$-Ru \approx \frac{-u_{i,j+1} - u_{i+1,j} - u_{i-1,j} - u_{i,j-1}}{h^2}$$

Then after this computing, the wavefield on each node is updated with

$$u^{(k+1)} = \omega D^{-1}(b - Ru^{(k)}) + (1 - \omega)u^{(k)}$$

The code snippets for updating the wavefield is:

```
double compute_xk(double *u, double *b, int N, double weight, int size)
/* Computing the new xk and send it to u */
{
    int ix, iz, ii;
    double invD, Ru, rhs, oldu, newu, tmp, obj=0.0;
    double *temp;

    temp = (double*)calloc(N*N, sizeof(double));
    invD = 1/4.0;

#pragma omp parallel for num_threads(size) reduction(+:obj) \
    private(ix, iz, ii, Ru, tmp) \
    shared(N, u, b)
    for(iz = 0; iz < N; iz++){
        for(ix = 0; ix < N; ix++){
            ii = ix+iz*N;
            Ru = 0.0;
            Ru -= (iz-1>=0)?u[ii-N]:0.0;
            Ru -= (iz+1<N)?u[ii+N]:0.0;
            Ru -= (ix-1>=0)?u[ii-1]:0.0;
            Ru -= (ix+1<N)?u[ii+1]:0.0;
            tmp = b[ii] - Ru;
            temp[ii] = tmp;
            tmp = tmp - 4.0*u[ii];
            obj += tmp*tmp;
        }
    }

#pragma omp parallel for num_threads(size) \
    private(ii, rhs, oldu, newu) \
    shared(N, u, temp, invD, weight)
    for(ii = 0; ii < N*N; ii++){
            rhs = invD*temp[ii];
            oldu = u[ii];
            newu = weight*rhs+ (1.0-weight)*oldu;
            u[ii] = newu;
    }

    free(temp);
    return sqrtf(obj);
}
```

Figure 1: Weighted Jacobi method code snippets

Figure 1 shows the snippets of the implementation of openmp based weighted Jacobi method, where u is the wavefield, b is the source function $f(x, y)$. The Ru is first computed for the off-diagonal modeling operator, then the objective function is calculated and stored in obj. However, there is a race condition when computing the objective fuction. To solve the problem, I use the $reduction(+ : obj)$ module to let the threads caches the sum results and put it back to a total sum after all the tasks on threads are finished.

In the numerical implementation, I show the examples by using different number of nodes on NOTS from 1 to 16. The solution is on the grid size of $N = 200$

```
The objective at iter 0 is: 0.00995024852
The objective at iter 1000 is: 0.00779355038
The objective at iter 2000 is: 0.00610431237
The objective at iter 3000 is: 0.00478121405
The final objective at iter 37681 is: 0.00000099985
The Total computing time is: 43.529999(s)
```

Figure 2: Weigthted Jacobi method for N=200 Serial code

```
N=200
Threads=1
The objective at iter 0 is: 0.00995024852
The objective at iter 1000 is: 0.00779355038
The objective at iter 2000 is: 0.00610431237
The objective at iter 3000 is: 0.00478121405
The final objective at iter 37681 is: 0.00000099985
The Total computing time is: 6.82655(s)
```

Figure 3: Weigthted Jacobi method for N=200 with 1 Thread

```
N=200
Threads=2
The objective at iter 0 is: 0.00995024852
The objective at iter 1000 is: 0.00779355038
The objective at iter 2000 is: 0.00610431237
The objective at iter 3000 is: 0.00478121405
The final objective at iter 37681 is: 0.00000099985
The Total computing time is: 4.34284(s)
```

Figure 4: Weigthted Jacobi method for N=200 with 2 Threads

```
N=200
Threads=4
The objective at iter 0 is: 0.00995024852
The objective at iter 1000 is: 0.00779355038
The objective at iter 2000 is: 0.00610431237
The objective at iter 3000 is: 0.00478121405
The final objective at iter 37681 is: 0.00000099985
The Total computing time is: 2.93351(s)
```

Figure 5: Weigthted Jacobi method for N=200 with 4 Threads

```
N=200
Threads=8
The objective at iter 0 is: 0.00995024852
The objective at iter 1000 is: 0.00779355038
The objective at iter 2000 is: 0.00610431237
The objective at iter 3000 is: 0.00478121405
The final objective at iter 37681 is: 0.00000099985
The Total computing time is: 2.45709(s)
```

Figure 6: Weigthted Jacobi method for N=200 with 8 Threads

```
N=200
Threads=16
The objective at iter 0 is: 0.00995024852
The objective at iter 1000 is: 0.00779355038
The objective at iter 2000 is: 0.00610431237
The objective at iter 3000 is: 0.00478121405
The final objective at iter 37681 is: 0.00000099985
The Total computing time is: 2.70607(s)
```

Figure 7: Weigthted Jacobi method for N=200 with 16 Threads

N threads	Computing time (s)
Serial	43.530
1	6.827
2	4.343
4	2.933
8	2.457
16	2.706

Table 1: Table for computing time using Weighted Jacobi method N=200

Figure 2 to 7 shows the computing time by serial code and by using OPENMP with different number of threads. For relatively small threads value N¡=4, we can see obvious improvements in the total computing time. The time is listed in the table 1, where we can see for large value of threads ($N = 8, 16$), the computing time stops decreasing and even raise a little bit.

For the OpenMP directives, I am using the pragma omp parallel for with the input number of threads. When compiling the program, I am using gcc -fopenmp -O3 myprog.c -o myprog. By using -fopenmp, I am using as may threads as available cores. The overall performance of the parallel program is good.

3 GaussSeidel method with OPENMP

The Gauss-Seidel method uses the following equations:

$$x_i{}^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j{}^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j{}^{(k)} \right), i = 1, 2, ..., N$$

We first compute the product of upper triangular matrix U and wavefield u, and then compute the b-Ux. Finally, we update the solution u by computing the inverse of lower triangular matrix using forward substitution. The code snippets are provided below:

```
16 double compute_xk(double *u, double *b, int N, double weight, int size)
17 /* Computing the new xk and send it to u */
18 {
19     int ix, iz, ii;
20     double invD, Uu, Lu, diff, tmp, obj=0.0;
21     invD = 1/4.0;
22
23 #pragma omp parallel for num_threads(size) reduction(+:obj) \
24     private(ix, iz, ii, diff, tmp) \
25     shared(N, u, b)
26     for(iz = 0; iz < N; iz++){
27         for(ix = 0; ix < N; ix++){
28             ii = ix+iz*N;
29             diff = 4.0*u[ii];
30             diff -= (iz+1<N)?u[ii+N]:0.0;
31             diff -= (ix+1<N)?u[ii+1]:0.0;
32             diff -= (iz-1>=0)?u[ii-N]:0.0;
33             diff -= (ix-1>=0)?u[ii-1]:0.0;
34             tmp = b[ii] - diff;
35             obj += tmp*tmp;
36         }
37     }
38 #pragma omp parallel for num_threads(size) \
39     private(ix, iz, ii, Uu, Lu, tmp) \
40     shared(N, u, b, invD)
41     for(iz = 0; iz < N; iz++){
42         for(ix = 0; ix < N; ix++){
43             ii = ix+iz*N;
44             Uu = 0.0;
45             Uu -= (iz+1<N)?u[ii+N]:0.0;
46             Uu -= (ix+1<N)?u[ii+1]:0.0;
47             tmp = b[ii] - Uu;
48             Lu = 0.0;
49             Lu -= (iz-1>=0)?u[ii-N]:0.0;
50             Lu -= (ix-1>=0)?u[ii-1]:0.0;
51             tmp = tmp - Lu;
52             u[ii] = invD*tmp;
53         }
54     }
55     return sqrtf(obj);
56 }
```

Figure 8: GaussSeidel method code snippets

Figure 8 shows the snippets of the implementation of openmp based Gauss-Seidel method, where u is the wavefield, b is the source function $f(x, y)$. The Uu is first computed for the Upper-triangle modeling operator, then wavefield u is updated. However, there is a race condition when computing the objective fuction. To solve the problem, I use the $reduction(+ : obj)$ module to let the threads caches the sum results and put it back to a total sum after all the tasks on threads are finished.

In the numerical implementation, I show the examples by using different number of nodes on NOTS from 1 to 16. The solution is on the grid size of $N = 200$

```
The objective at iter 0 is: 0.00995024852
The objective at iter 1000 is: 0.00374705927
The objective at iter 2000 is: 0.00141071563
The objective at iter 3000 is: 0.00053106103
The final objective at iter 9524 is: 0.00000099989
The Total computing time is: 12.180000(s)
```

Figure 9: Gauss Seidel method on N=200, Serial code

```
N=200
Threads=1
The objective at iter 0 is: 0.00995024852
The objective at iter 1000 is: 0.00374705927
The objective at iter 2000 is: 0.00141071563
The objective at iter 3000 is: 0.00053106103
The final objective at iter 9524 is: 0.00000099989
The Total computing time is: 2.73528(s)
```

Figure 10: Gauss Seidel method on N=200 with 1 Thread

```
N=200
Threads=2
The objective at iter 0 is: 0.00995024852
The objective at iter 1000 is: 0.00374704390
The objective at iter 2000 is: 0.00141070457
The objective at iter 3000 is: 0.00053105532
The final objective at iter 9522 is: 0.00000099983
The Total computing time is: 1.42131(s)
```

Figure 11: Gauss Seidel method on N=200 with 2 Threads

```
N=200
Threads=4
The objective at iter 0 is: 0.00995024852
The objective at iter 1000 is: 0.00376562309
The objective at iter 2000 is: 0.00141767820
The objective at iter 3000 is: 0.00053367455
The final objective at iter 9526 is: 0.00000099937
The Total computing time is: 0.778582(s)
```

Figure 12: Gauss Seidel method on N=200 with 4 Threads

```
N=200
Threads=8
The objective at iter 0 is: 0.00995024852
The objective at iter 1000 is: 0.00378422975
The objective at iter 2000 is: 0.00142466056
The objective at iter 3000 is: 0.00053629623
The final objective at iter 9527 is: 0.00000099990
The Total computing time is: 0.492171(s)
```

Figure 13: Gauss Seidel method on N=200 with 8 Threads

```
N=200
Threads=16
The objective at iter 0 is: 0.00995024852
The objective at iter 1000 is: 0.00381826679
The objective at iter 2000 is: 0.00143743260
The objective at iter 3000 is: 0.00054109184
The final objective at iter 9530 is: 0.00000099996
The Total computing time is: 0.395271(s)
```

Figure 14: Gauss Seidel method on N=200 with 16 Threads

N threads	Computing time (s)
Serial	12.180
1	2.735
2	1.421
4	0.779
8	0.492
16	0.395

Table 2: Table for computing time using Gauss Seidel method N=200

Figure 9 to 14 shows the computing time by serial code and by using OPENMP with different number of threads. For improving the number of threads, we can see obvious improvements in the total computing time. The time is listed in the table 2, where we can see the improvements from 8 threads to 16 threads is relatively smaller than from 1 thread to 2 threads.

For the OpenMP directives, I am using the pragma omp parallel for with the input number of threads. When compiling the program, I am using gcc -fopenmp -O3 myprog.c -o myprog. By using -fopenmp, I am using as may threads as available cores. The overall performance of the parallel program is good.

4 Successive over-relaxation method with OPENMP

The SOR method uses the following equations:

$$x_i{}^{(k+1)} = (1-\omega)x_i{}^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j{}^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j{}^{(k)}\right), i = 1, 2, ..., N$$

Where the *omega* is called the relaxation factor. The method goes back to Gauss-Seidel method when $\omega = 0$. We first compute the product of upper triangular matrix U and wavefield u, and then compute the b-Ux. Finally, we update the solution u by computing the linear combination of updates from Gauss-Seidel method and solution u at last time step. The relaxation factor is $\omega = 0.9$. The code snippets are provided below:

```
16 double compute_xk(double *u, double *b, int N, double weight, int size)
17 /* Computing the new xk and send it to u */
18 {
19     int ix, iz, ii;
20     double invD, Uu, diff, tmp, obj=0.0;
21     double *temp;
22     invD = 1/4.0;
23
24     temp = (double*)calloc(N*N,sizeof(double));
25
26 #pragma omp parallel for num_threads(size) reduction(+:obj) \
27     private(ix, iz, ii, diff, tmp) \
28     shared(N, u, b)
29     for(iz = 0; iz < N; iz++){
30         for(ix = 0; ix < N; ix++){
31             ii = ix+iz*N;
32             diff = 4.0*u[ii];
33             diff -= (iz+1<N)?u[ii+N]:0.0;
34             diff -= (ix+1<N)?u[ii+1]:0.0;
35             diff -= (iz-1>=0)?u[ii-N]:0.0;
36             diff -= (ix-1>=0)?u[ii-1]:0.0;
37             tmp = b[ii] - diff;
38             obj += tmp*tmp;
39         }
40     }
41 #pragma omp parallel for num_threads(size) \
42     default(none) private(ix, iz, ii, Uu, tmp) \
43     shared(N, u, temp, b)
44     for(iz = 0; iz < N; iz++){
45         for(ix = 0; ix < N; ix++){
46             ii = ix+iz*N;
47             Uu = 0.0;
48             Uu -= (iz+1<N)?u[ii+N]:0.0;
49             Uu -= (ix+1<N)?u[ii+1]:0.0;
50             tmp = b[ii] - Uu;
51             temp[ii] = tmp;
52         }
53     }
54 #pragma omp parallel for num_threads(size) \
55     private(ix, iz, ii, Uu, tmp) \
56     shared(N, u, temp, weight, invD)
57     for(iz = 0; iz < N; iz++){
58         for(ix = 0; ix < N; ix++){
59             ii = ix+iz*N;
60             Uu = 0.0;
61             Uu -= (iz-1>=0)?u[ii-N]:0.0;
62             Uu -= (ix-1>=0)?u[ii-1]:0.0;
63             tmp = temp[ii] - Uu;
64             //u[ii] = invD*tmp;
65             u[ii] = weight*invD*tmp + (1.0-weight)*u[ii];
66         }
67     }
68
69     free(temp);
70     return sqrtf(obj);
71 }
```

Figure 15: SOR method code snippets

Figure 15 shows the snippets of the implementation of openmp based SOR method, where u is the wavefield, b is the source function $f(x, y)$. The Uu is first computed for the Upper-triangle modeling operator, then wavefield u is updated based on the weight factor. However, there is a race condition when computing the objective fuction. To solve the problem, I use the $reduction(+ : obj)$ module to let the threads caches the sum results and put it back to a total sum after all the tasks on threads are finished.

In the numerical implementation, I show the examples by using different number of nodes on NOTS from 1 to 16. The solution is on the grid size of $N = 200$

```
The objective at iter 0 is: 0.00995024852
The objective at iter 1000 is: 0.00447465898
The objective at iter 2000 is: 0.00201202766
The objective at iter 3000 is: 0.00090463727
The final objective at iter 9999 is: 0.00000344215
The Total computing time is: 13.350000(s)
```

Figure 16: SOR method on N=200, Serial code

```
N=200
Thread=1
The objective at iter 0 is: 0.00995024852
The objective at iter 1000 is: 0.00447465898
The objective at iter 2000 is: 0.00201202766
The objective at iter 3000 is: 0.00090463727
The final objective at iter 11600 is: 0.00000099953
The Total computing time is: 4.35807(s)
```

Figure 17: SOR method on N=200 with 1 Thread

```
N=200
Thread=2
The objective at iter 0 is: 0.00995024852
The objective at iter 1000 is: 0.00447464874
The objective at iter 2000 is: 0.00201201905
The objective at iter 3000 is: 0.00090463308
The final objective at iter 11601 is: 0.00000099929
The Total computing time is: 2.50927(s)
```

Figure 18: SOR method on N=200 with 2 Threads

```
N=200
Thread=4
The objective at iter 0 is: 0.00995024852
The objective at iter 1000 is: 0.00451094424
The objective at iter 2000 is: 0.00204151939
The objective at iter 3000 is: 0.00092385890
The final objective at iter 11694 is: 0.00000099944
The Total computing time is: 1.53357(s)
```

Figure 19: SOR method on N=200 with 4 Threads

```
N=200
Thread=8
The objective at iter 0 is: 0.00995024852
The objective at iter 1000 is: 0.00454663532
The objective at iter 2000 is: 0.00207084860
The objective at iter 3000 is: 0.00094313896
The final objective at iter 11788 is: 0.00000099953
The Total computing time is: 1.19003(s)
```

Figure 20: SOR method on N=200 with 8 Threads

```
N=200
Thread=16
The objective at iter 0 is: 0.00995024852
The objective at iter 1000 is: 0.00461050682
The objective at iter 2000 is: 0.00212424109
The objective at iter 3000 is: 0.00097865658
The final objective at iter 11961 is: 0.00000099974
The Total computing time is: 1.11655(s)
```

Figure 21: SOR method on N=200 with 16 Threads

N threads	Computing time (s)
Serial	13.350
1	4.358
2	2.509
4	1.534
8	1.190
16	1.117

Table 3: Table for computing time using SOR method N=200

Figure 16 to 21 shows the computing time by serial code and by using OPENMP with different number of threads. For increasing the number of threads, we can see obvious improvements in the total computing time. The time is listed in the table 3, where we can see the improvements from 8 threads to 16 threads is relatively smaller than the improvements from 1 thread to 2 threads.

For the OpenMP directives, I am using the pragma omp parallel for with the input number of threads. When compiling the program, I am using gcc -fopenmp -O3 myprog.c -o myprog. By using -fopenmp, I am using as may threads as available cores. The overall performance of the parallel program is good.

5 Conjugate Gradient method with OPENMP

The algorithm is described below for solving our problem $Au = b$:

$$r_0 = b - Au_0$$

$$p_0 = r_0$$

$$k = 0$$

while $||r_k|| >= tol$

$$\alpha_k = \frac{{r_k}^T r_k}{{p_k}^T A p_k}$$

$$u_{k+1} = u_k + \alpha_k p_k$$

$$r_{k+1} = r_k - \alpha_k A p_k$$

$$\beta_k = \frac{{r_{k+1}}^T r_{k+1}}{{r_k}^T r_k}$$

$$p_{k+1} = r_{k+1} + \beta_k p_k$$

$$k = k + 1$$

From the algorithm of CG method, we can expect that because the algorithm uses the gradient information and has approximate the Hessian matrix, if the equation is linear, it may take 1 iteration for the CG method to find out the solution. The code snippets are provided below:

```c
double compute_alpha(double *temp, double *r, double *p, int N, int size)
/* Computing the parameter alpha, temp is the working matrix */
{
    int ii;
    double alpha, alpha1, alpha2;
    alpha1 = alpha2 = 0.0;
    compute_Au(temp, p, N, size);

#pragma omp parallel for num_threads(size) reduction(+:alpha1) reduction(+:alpha2) \
    private(ii) shared(r, p, temp, N)
    for(ii = 0; ii < N*N; ii++){
        alpha1 += r[ii]*r[ii];
        alpha2 += p[ii]*temp[ii];
    }
    alpha = alpha1/alpha2;
    return alpha;
}
void compute_xk(double *u, double *p, double alpha, int N, int size)
/* Computing the new xk and send it to u using conjugate gradient method */
{
    int ii;

#pragma omp parallel for num_threads(size) \
    private(ii) shared(u, alpha, p, N)
    for(ii = 0; ii < N*N; ii++){
            u[ii] = u[ii] + alpha*p[ii];
    }
}
double compute_rk(double *temp, double *r, double alpha, int N, int size)
/* Updating the residual rk, temp is the working matrix */
{
    int ii;
    double norm, tmp;
    norm = 0.0;

#pragma omp parallel for num_threads(size)  reduction(+:norm) \
    private(ii, tmp) shared(r, alpha, temp, N)
    for(ii = 0; ii < N*N; ii++){
        tmp = r[ii];
        norm += tmp*tmp;
        r[ii] = r[ii] - alpha*temp[ii];
    }
    return norm;
}
```

```c
double compute_beta(double *r, double norm, int N, int size)
/* Computing the new pk for conjugate gradient method */
{
    int ii;
    double beta, beta0, tmp;
    beta0 = 0.0;

#pragma omp parallel for num_threads(size)  reduction(+:beta0) \
    private(ii, tmp) shared(r, N)
    for(ii = 0; ii < N*N; ii++){
        tmp = r[ii];
        beta0 += tmp*tmp;
    }
    beta = beta0/norm;
    return beta;
}
void compute_pk(double *r, double *p, double beta, int N, int size)
/* Computing the new pk for conjugate gradient method */
{
    int ii;

#pragma omp parallel for num_threads(size) \
    private(ii) shared(r, p, beta, N)
    for(ii = 0; ii < N*N; ii++){
            p[ii] = r[ii] + beta*p[ii];
    }
}
```

Figure 22: Conjugate gradient method code snippets

Figure 22 shows the snippets of the implementation of openmp based CG method, where u is the wavefield, b is the source function $f(x, y)$. The α and β values are calculated in the corresponding 'compute' function. However, there is a race condition when computing the objective fuction, alpha and beta. To solve the problem, I use the $reduction(+ : obj)$, $reduction(+ : alpha2)$, $reduction(+ : beta0)$ module to let the threads caches the sum results and put it back to a total sum after all the tasks on threads are finished.

```
The objective at iter 0 is: 0.00995024852
The objective at iter 1 is: 0.00000000000
The final objective at iter 1 is: 0.00000000000
The Total computing time is: 0.0000000000000000000(s)
```

Figure 23: Conjugate Gradient method on N=200, Serial code

The CG method is superfast for N=200, as shown in Figure 23. In the numerical implementation, I show the examples by using different number of nodes on NOTS from 1 to 16. The solution is on the grid size of $N = 5000$

```
The objective at iter 0 is: 0.00039992001
The objective at iter 1 is: 0.00000000000
The final objective at iter 1 is: 0.00000000000
The Total computing time is: 4.88999986648559570 31(s)
```

Figure 24: Conjugate Gradient method on N=5000, Serial code

```
N=5000
Thread=1
The objective at iter 0 is: 0.00039992001
The objective at iter 1 is: 0.00000000000
The final objective at iter 1 is: 0.00000000000
The Total computing time is: 1.90161(s)
```

Figure 25: Conjugate Gradient method on N=5000 with 1 Thread

```
N=5000
Thread=2
The objective at iter 0 is: 0.00039992001
The objective at iter 1 is: 0.00000000000
The final objective at iter 1 is: 0.00000000000
The Total computing time is: 1.20263(s)
```

Figure 26: Conjugate Gradient method on N=5000 with 2 Threads

```
N=5000
Thread=4
The objective at iter 0 is: 0.00039992001
The objective at iter 1 is: 0.00000000000
The final objective at iter 1 is: 0.00000000000
The Total computing time is: 0.832491(s)
```

Figure 27: Conjugate Gradient method on N=5000 with 4 Threads

```
N=5000
Thread=8
The objective at iter 0 is: 0.00039992001
The objective at iter 1 is: 0.00000000000
The final objective at iter 1 is: 0.00000000000
The Total computing time is: 0.61372(s)
```

Figure 28: Conjugate Gradient method on N=5000 with 8 Threads

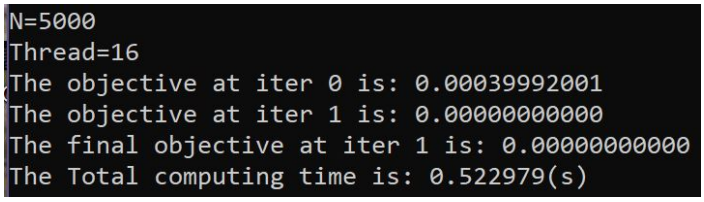

Figure 29: Conjugate Gradient method on N=5000 with 16 Threads

N threads	Computing time (s)
Serial	4.890
1	1.901
2	1.202
4	0.832
8	0.614
16	0.523

Table 4: Table for computing time using Conjugate Gradient method N=5000

Figure 24 to 29 shows the computing time by serial code and by using OPENMP with different number of threads. For increasing the number of threads, we can see obvious improvements in the total computing time. The time is listed in the table 4, where we can see the improvements from 8 threads to 16 threads is relatively smaller than the improvements from 1 thread to 2 threads.

For the OpenMP directives, I am using the pragma omp parallel for with the input number of threads. When compiling the program, I am using gcc -fopenmp -O3 myprog.c -o myprog. By using -fopenmp, I am using as may threads as available cores. The overall performance of the parallel program is good.

In sum, for all the parallel program using openmp, I use the omp parallel for directives to let the threads parallel computing the for loops. When there is a summation in the for loop, there will be the race condition that the different threads might have the make the add function while the add operation in the other threads is not finished or the added sum is not put back to the shared variables. For this situation, the reduction() module is very useful to let different threads caches the sum at current steps.