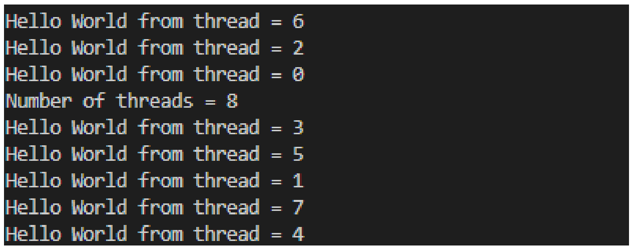
Huy Dinh Tran
htran197
ID: 862325308

PA1 Part 1 Report

1.1:

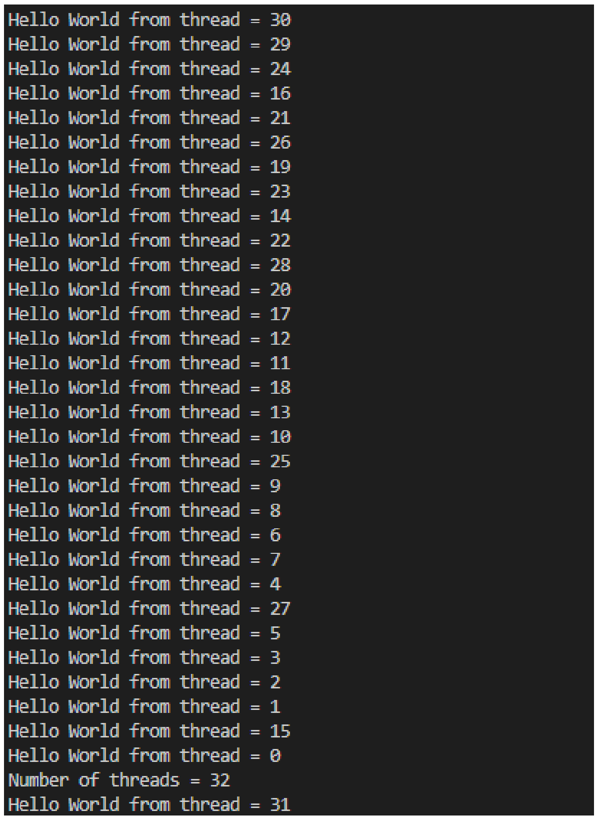
a)



```
Hello World from thread = 6
Hello World from thread = 2
Hello World from thread = 0
Number of threads = 8
Hello World from thread = 3
Hello World from thread = 5
Hello World from thread = 1
Hello World from thread = 7
Hello World from thread = 4
```

OpenMP is an SMP library for parallel programming in which all threads share the same memory. We got different thread orders every time we ran because the threads interleave during execution and the OS schedule decides which thread to run for fairness.

b)

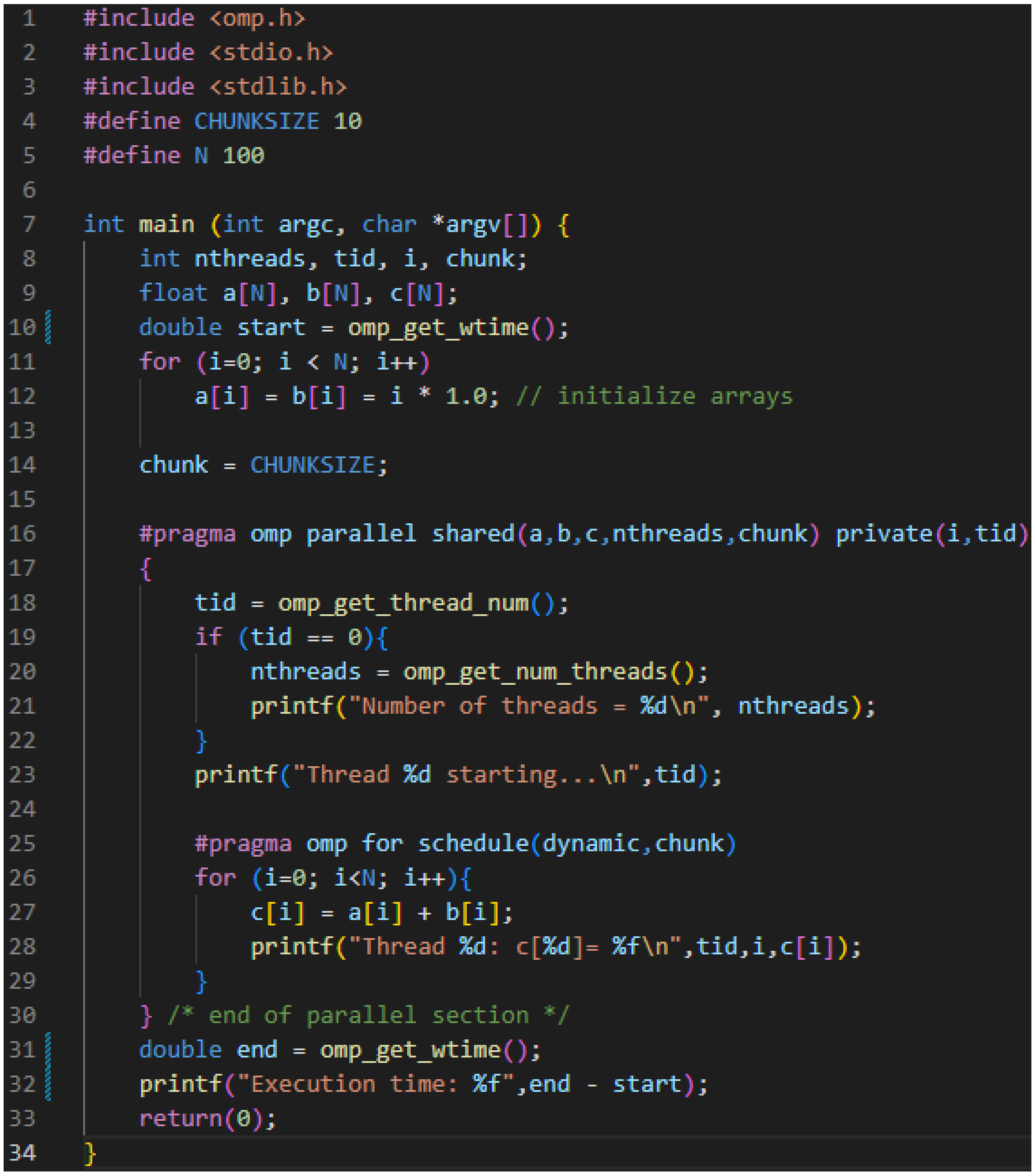


```
Hello World from thread = 30
Hello World from thread = 29
Hello World from thread = 24
Hello World from thread = 16
Hello World from thread = 21
Hello World from thread = 26
Hello World from thread = 19
Hello World from thread = 23
Hello World from thread = 14
Hello World from thread = 22
Hello World from thread = 28
Hello World from thread = 20
Hello World from thread = 17
Hello World from thread = 12
Hello World from thread = 11
Hello World from thread = 18
Hello World from thread = 13
Hello World from thread = 10
Hello World from thread = 25
Hello World from thread = 9
Hello World from thread = 8
Hello World from thread = 6
Hello World from thread = 7
Hello World from thread = 4
Hello World from thread = 27
Hello World from thread = 5
Hello World from thread = 3
Hello World from thread = 2
Hello World from thread = 1
Hello World from thread = 15
Hello World from thread = 0
Number of threads = 32
Hello World from thread = 31
```

Similar to part a but now we set OpenMP to create 32 threads. We can still see the ordering of threads is still random which is explained above.

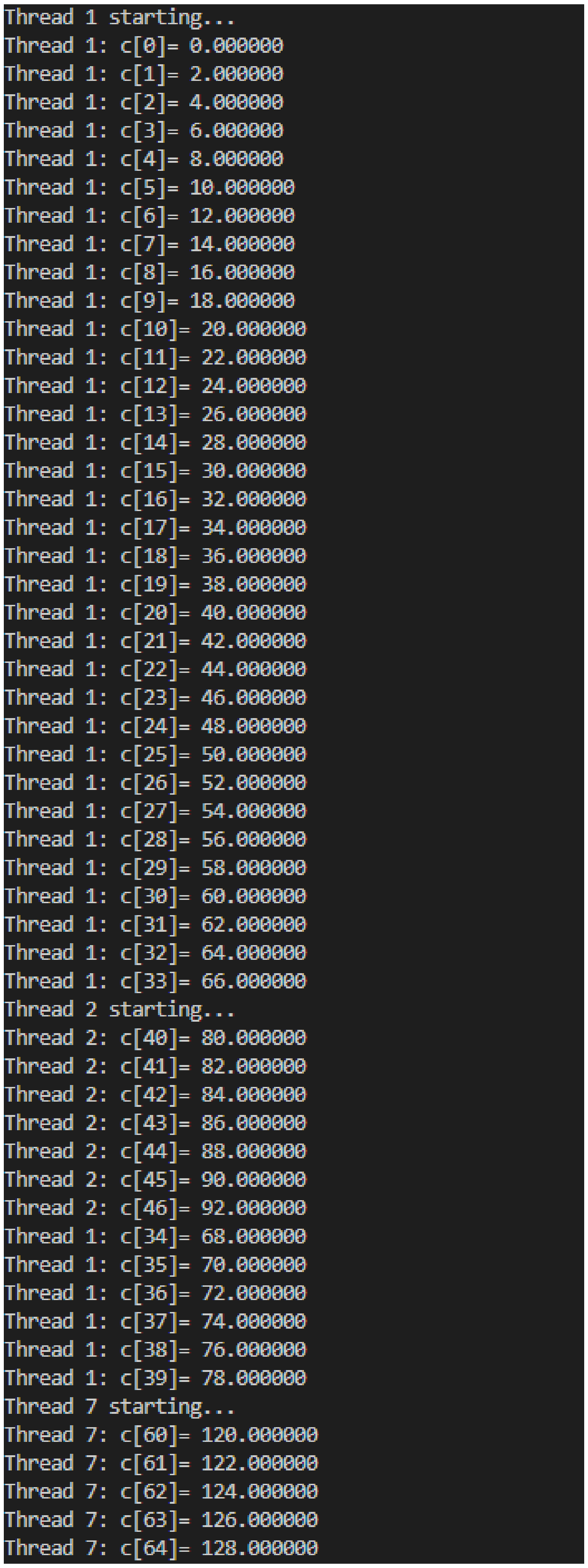
1.2:

a)

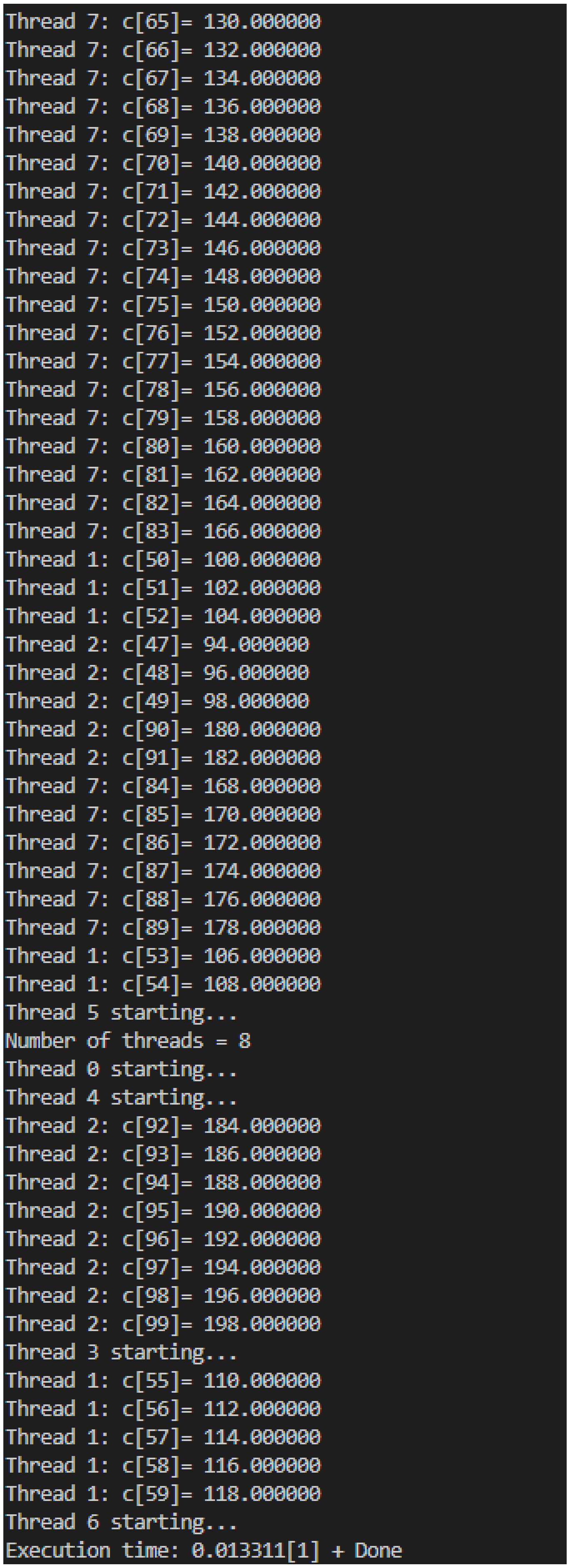


```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define CHUNKSIZE 10
#define N 100

int main (int argc, char *argv[]) {
    int nthreads, tid, i, chunk;
    float a[N], b[N], c[N];
    double start = omp_get_wtime();
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0; // initialize arrays

    chunk = CHUNKSIZE;

    #pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
    {
        tid = omp_get_thread_num();
        if (tid == 0){
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
        printf("Thread %d starting...\n",tid);

        #pragma omp for schedule(dynamic,chunk)
        for (i=0; i<N; i++){
            c[i] = a[i] + b[i];
            printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
        }
    } /* end of parallel section */
    double end = omp_get_wtime();
    printf("Execution time: %f",end - start);
    return(0);
}
```

b)
Dynamic:

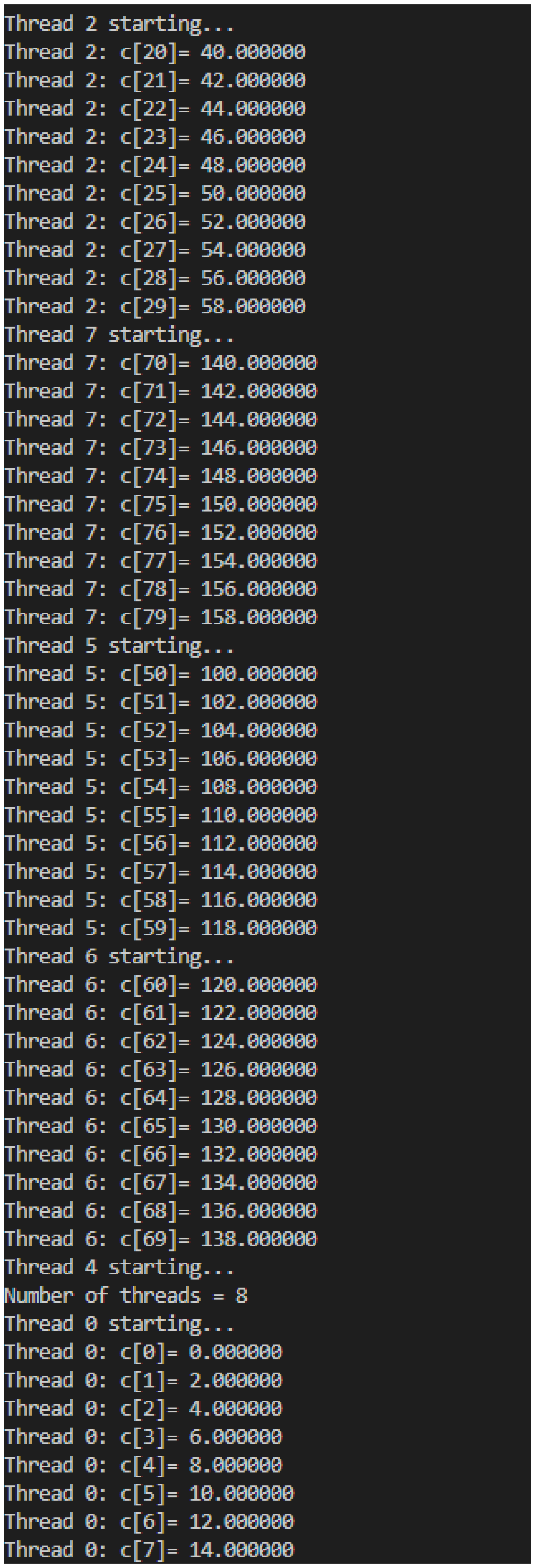


```
Thread 1 starting...
Thread 1: c[0]= 0.000000
Thread 1: c[1]= 2.000000
Thread 1: c[2]= 4.000000
Thread 1: c[3]= 6.000000
Thread 1: c[4]= 8.000000
Thread 1: c[5]= 10.000000
Thread 1: c[6]= 12.000000
Thread 1: c[7]= 14.000000
Thread 1: c[8]= 16.000000
Thread 1: c[9]= 18.000000
Thread 1: c[10]= 20.000000
Thread 1: c[11]= 22.000000
Thread 1: c[12]= 24.000000
Thread 1: c[13]= 26.000000
Thread 1: c[14]= 28.000000
Thread 1: c[15]= 30.000000
Thread 1: c[16]= 32.000000
Thread 1: c[17]= 34.000000
Thread 1: c[18]= 36.000000
Thread 1: c[19]= 38.000000
Thread 1: c[20]= 40.000000
Thread 1: c[21]= 42.000000
Thread 1: c[22]= 44.000000
Thread 1: c[23]= 46.000000
Thread 1: c[24]= 48.000000
Thread 1: c[25]= 50.000000
Thread 1: c[26]= 52.000000
Thread 1: c[27]= 54.000000
Thread 1: c[28]= 56.000000
Thread 1: c[29]= 58.000000
Thread 1: c[30]= 60.000000
Thread 1: c[31]= 62.000000
Thread 1: c[32]= 64.000000
Thread 1: c[33]= 66.000000
Thread 2 starting...
Thread 2: c[40]= 80.000000
Thread 2: c[41]= 82.000000
Thread 2: c[42]= 84.000000
Thread 2: c[43]= 86.000000
Thread 2: c[44]= 88.000000
Thread 2: c[45]= 90.000000
Thread 2: c[46]= 92.000000
Thread 1: c[34]= 68.000000
Thread 1: c[35]= 70.000000
Thread 1: c[36]= 72.000000
Thread 1: c[37]= 74.000000
Thread 1: c[38]= 76.000000
Thread 1: c[39]= 78.000000
Thread 7 starting...
Thread 7: c[60]= 120.000000
Thread 7: c[61]= 122.000000
Thread 7: c[62]= 124.000000
Thread 7: c[63]= 126.000000
Thread 7: c[64]= 128.000000
```

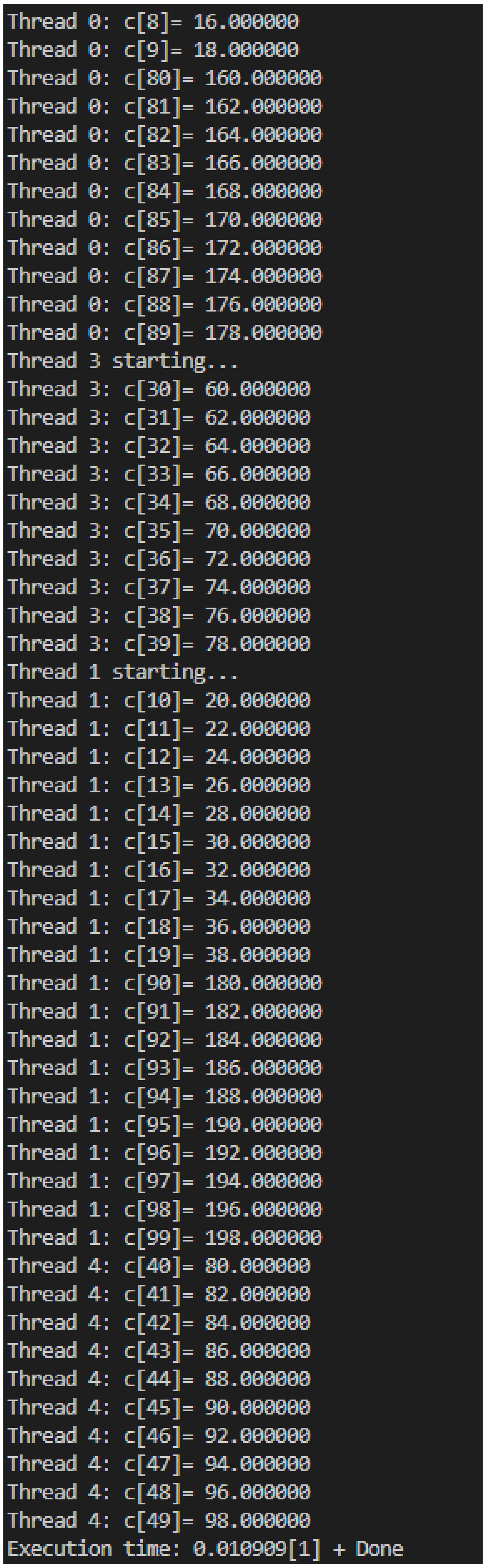


```
Thread 7: c[65]= 130.000000
Thread 7: c[66]= 132.000000
Thread 7: c[67]= 134.000000
Thread 7: c[68]= 136.000000
Thread 7: c[69]= 138.000000
Thread 7: c[70]= 140.000000
Thread 7: c[71]= 142.000000
Thread 7: c[72]= 144.000000
Thread 7: c[73]= 146.000000
Thread 7: c[74]= 148.000000
Thread 7: c[75]= 150.000000
Thread 7: c[76]= 152.000000
Thread 7: c[77]= 154.000000
Thread 7: c[78]= 156.000000
Thread 7: c[79]= 158.000000
Thread 7: c[80]= 160.000000
Thread 7: c[81]= 162.000000
Thread 7: c[82]= 164.000000
Thread 7: c[83]= 166.000000
Thread 1: c[50]= 100.000000
Thread 1: c[51]= 102.000000
Thread 1: c[52]= 104.000000
Thread 2: c[47]= 94.000000
Thread 2: c[48]= 96.000000
Thread 2: c[49]= 98.000000
Thread 2: c[90]= 180.000000
Thread 2: c[91]= 182.000000
Thread 7: c[84]= 168.000000
Thread 7: c[85]= 170.000000
Thread 7: c[86]= 172.000000
Thread 7: c[87]= 174.000000
Thread 7: c[88]= 176.000000
Thread 7: c[89]= 178.000000
Thread 1: c[53]= 106.000000
Thread 1: c[54]= 108.000000
Thread 5 starting...
Number of threads = 8
Thread 0 starting...
Thread 4 starting...
Thread 2: c[92]= 184.000000
Thread 2: c[93]= 186.000000
Thread 2: c[94]= 188.000000
Thread 2: c[95]= 190.000000
Thread 2: c[96]= 192.000000
Thread 2: c[97]= 194.000000
Thread 2: c[98]= 196.000000
Thread 2: c[99]= 198.000000
Thread 3 starting...
Thread 1: c[55]= 110.000000
Thread 1: c[56]= 112.000000
Thread 1: c[57]= 114.000000
Thread 1: c[58]= 116.000000
Thread 1: c[59]= 118.000000
Thread 6 starting...
Execution time: 0.013311[1] + Done
```

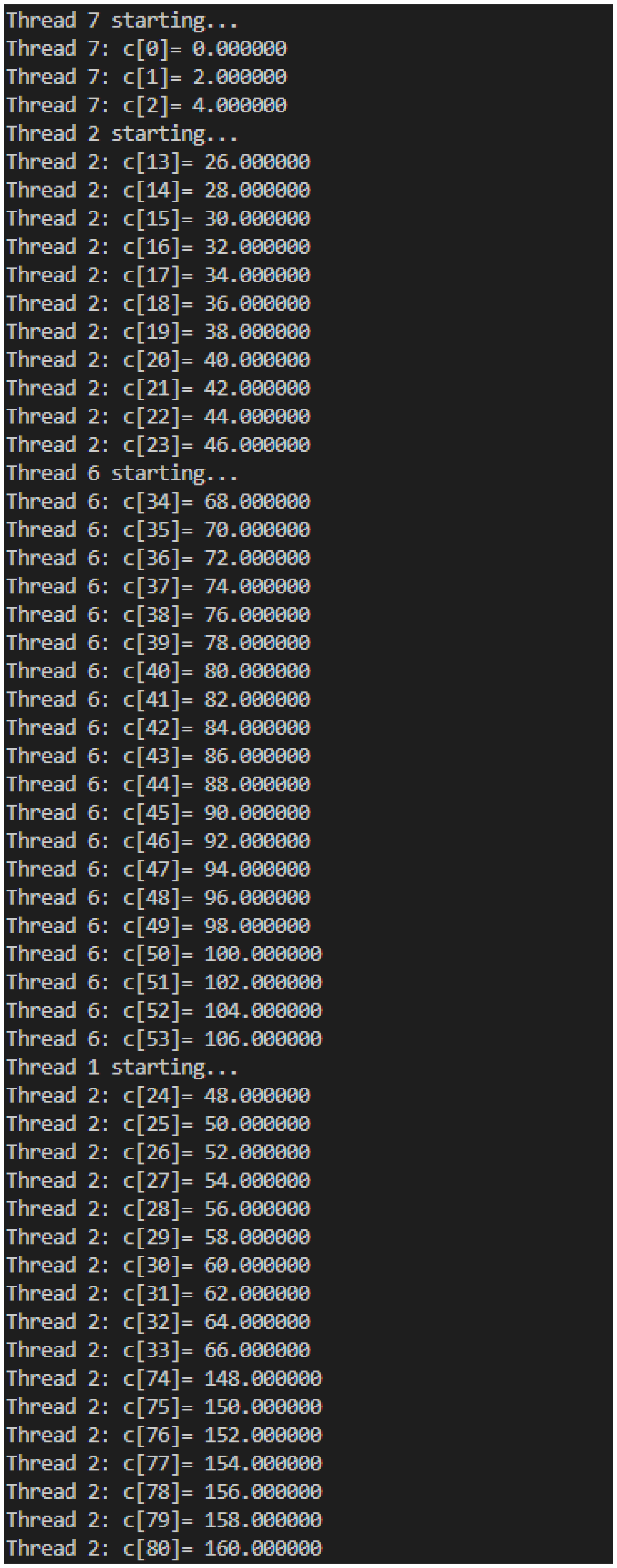
c)
Static:



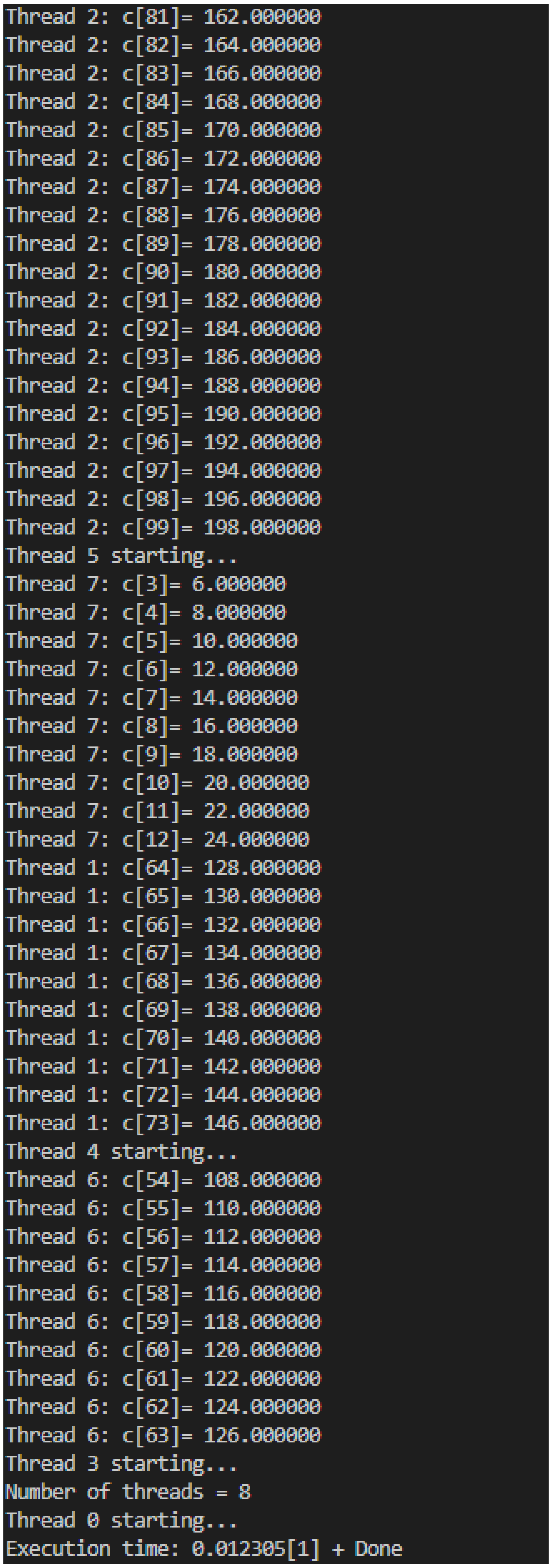
```
Thread 2 starting...
Thread 2: c[20]= 40.000000
Thread 2: c[21]= 42.000000
Thread 2: c[22]= 44.000000
Thread 2: c[23]= 46.000000
Thread 2: c[24]= 48.000000
Thread 2: c[25]= 50.000000
Thread 2: c[26]= 52.000000
Thread 2: c[27]= 54.000000
Thread 2: c[28]= 56.000000
Thread 2: c[29]= 58.000000
Thread 7 starting...
Thread 7: c[70]= 140.000000
Thread 7: c[71]= 142.000000
Thread 7: c[72]= 144.000000
Thread 7: c[73]= 146.000000
Thread 7: c[74]= 148.000000
Thread 7: c[75]= 150.000000
Thread 7: c[76]= 152.000000
Thread 7: c[77]= 154.000000
Thread 7: c[78]= 156.000000
Thread 7: c[79]= 158.000000
Thread 5 starting...
Thread 5: c[50]= 100.000000
Thread 5: c[51]= 102.000000
Thread 5: c[52]= 104.000000
Thread 5: c[53]= 106.000000
Thread 5: c[54]= 108.000000
Thread 5: c[55]= 110.000000
Thread 5: c[56]= 112.000000
Thread 5: c[57]= 114.000000
Thread 5: c[58]= 116.000000
Thread 5: c[59]= 118.000000
Thread 6 starting...
Thread 6: c[60]= 120.000000
Thread 6: c[61]= 122.000000
Thread 6: c[62]= 124.000000
Thread 6: c[63]= 126.000000
Thread 6: c[64]= 128.000000
Thread 6: c[65]= 130.000000
Thread 6: c[66]= 132.000000
Thread 6: c[67]= 134.000000
Thread 6: c[68]= 136.000000
Thread 6: c[69]= 138.000000
Thread 4 starting...
Number of threads = 8
Thread 0 starting...
Thread 0: c[0]= 0.000000
Thread 0: c[1]= 2.000000
Thread 0: c[2]= 4.000000
Thread 0: c[3]= 6.000000
Thread 0: c[4]= 8.000000
Thread 0: c[5]= 10.000000
Thread 0: c[6]= 12.000000
Thread 0: c[7]= 14.000000
```


```
Thread 0: c[8]= 16.000000
Thread 0: c[9]= 18.000000
Thread 0: c[80]= 160.000000
Thread 0: c[81]= 162.000000
Thread 0: c[82]= 164.000000
Thread 0: c[83]= 166.000000
Thread 0: c[84]= 168.000000
Thread 0: c[85]= 170.000000
Thread 0: c[86]= 172.000000
Thread 0: c[87]= 174.000000
Thread 0: c[88]= 176.000000
Thread 0: c[89]= 178.000000
Thread 3 starting...
Thread 3: c[30]= 60.000000
Thread 3: c[31]= 62.000000
Thread 3: c[32]= 64.000000
Thread 3: c[33]= 66.000000
Thread 3: c[34]= 68.000000
Thread 3: c[35]= 70.000000
Thread 3: c[36]= 72.000000
Thread 3: c[37]= 74.000000
Thread 3: c[38]= 76.000000
Thread 3: c[39]= 78.000000
Thread 1 starting...
Thread 1: c[10]= 20.000000
Thread 1: c[11]= 22.000000
Thread 1: c[12]= 24.000000
Thread 1: c[13]= 26.000000
Thread 1: c[14]= 28.000000
Thread 1: c[15]= 30.000000
Thread 1: c[16]= 32.000000
Thread 1: c[17]= 34.000000
Thread 1: c[18]= 36.000000
Thread 1: c[19]= 38.000000
Thread 1: c[90]= 180.000000
Thread 1: c[91]= 182.000000
Thread 1: c[92]= 184.000000
Thread 1: c[93]= 186.000000
Thread 1: c[94]= 188.000000
Thread 1: c[95]= 190.000000
Thread 1: c[96]= 192.000000
Thread 1: c[97]= 194.000000
Thread 1: c[98]= 196.000000
Thread 1: c[99]= 198.000000
Thread 4: c[40]= 80.000000
Thread 4: c[41]= 82.000000
Thread 4: c[42]= 84.000000
Thread 4: c[43]= 86.000000
Thread 4: c[44]= 88.000000
Thread 4: c[45]= 90.000000
Thread 4: c[46]= 92.000000
Thread 4: c[47]= 94.000000
Thread 4: c[48]= 96.000000
Thread 4: c[49]= 98.000000
Execution time: 0.010909[1] + Done
```

Guided:



```
Thread 7 starting...
Thread 7: c[0]= 0.000000
Thread 7: c[1]= 2.000000
Thread 7: c[2]= 4.000000
Thread 2 starting...
Thread 2: c[13]= 26.000000
Thread 2: c[14]= 28.000000
Thread 2: c[15]= 30.000000
Thread 2: c[16]= 32.000000
Thread 2: c[17]= 34.000000
Thread 2: c[18]= 36.000000
Thread 2: c[19]= 38.000000
Thread 2: c[20]= 40.000000
Thread 2: c[21]= 42.000000
Thread 2: c[22]= 44.000000
Thread 2: c[23]= 46.000000
Thread 6 starting...
Thread 6: c[34]= 68.000000
Thread 6: c[35]= 70.000000
Thread 6: c[36]= 72.000000
Thread 6: c[37]= 74.000000
Thread 6: c[38]= 76.000000
Thread 6: c[39]= 78.000000
Thread 6: c[40]= 80.000000
Thread 6: c[41]= 82.000000
Thread 6: c[42]= 84.000000
Thread 6: c[43]= 86.000000
Thread 6: c[44]= 88.000000
Thread 6: c[45]= 90.000000
Thread 6: c[46]= 92.000000
Thread 6: c[47]= 94.000000
Thread 6: c[48]= 96.000000
Thread 6: c[49]= 98.000000
Thread 6: c[50]= 100.000000
Thread 6: c[51]= 102.000000
Thread 6: c[52]= 104.000000
Thread 6: c[53]= 106.000000
Thread 1 starting...
Thread 2: c[24]= 48.000000
Thread 2: c[25]= 50.000000
Thread 2: c[26]= 52.000000
Thread 2: c[27]= 54.000000
Thread 2: c[28]= 56.000000
Thread 2: c[29]= 58.000000
Thread 2: c[30]= 60.000000
Thread 2: c[31]= 62.000000
Thread 2: c[32]= 64.000000
Thread 2: c[33]= 66.000000
Thread 2: c[74]= 148.000000
Thread 2: c[75]= 150.000000
Thread 2: c[76]= 152.000000
Thread 2: c[77]= 154.000000
Thread 2: c[78]= 156.000000
Thread 2: c[79]= 158.000000
Thread 2: c[80]= 160.000000
```



```
Thread 2: c[81]= 162.000000
Thread 2: c[82]= 164.000000
Thread 2: c[83]= 166.000000
Thread 2: c[84]= 168.000000
Thread 2: c[85]= 170.000000
Thread 2: c[86]= 172.000000
Thread 2: c[87]= 174.000000
Thread 2: c[88]= 176.000000
Thread 2: c[89]= 178.000000
Thread 2: c[90]= 180.000000
Thread 2: c[91]= 182.000000
Thread 2: c[92]= 184.000000
Thread 2: c[93]= 186.000000
Thread 2: c[94]= 188.000000
Thread 2: c[95]= 190.000000
Thread 2: c[96]= 192.000000
Thread 2: c[97]= 194.000000
Thread 2: c[98]= 196.000000
Thread 2: c[99]= 198.000000
Thread 5 starting...
Thread 7: c[3]= 6.000000
Thread 7: c[4]= 8.000000
Thread 7: c[5]= 10.000000
Thread 7: c[6]= 12.000000
Thread 7: c[7]= 14.000000
Thread 7: c[8]= 16.000000
Thread 7: c[9]= 18.000000
Thread 7: c[10]= 20.000000
Thread 7: c[11]= 22.000000
Thread 7: c[12]= 24.000000
Thread 1: c[64]= 128.000000
Thread 1: c[65]= 130.000000
Thread 1: c[66]= 132.000000
Thread 1: c[67]= 134.000000
Thread 1: c[68]= 136.000000
Thread 1: c[69]= 138.000000
Thread 1: c[70]= 140.000000
Thread 1: c[71]= 142.000000
Thread 1: c[72]= 144.000000
Thread 1: c[73]= 146.000000
Thread 4 starting...
Thread 6: c[54]= 108.000000
Thread 6: c[55]= 110.000000
Thread 6: c[56]= 112.000000
Thread 6: c[57]= 114.000000
Thread 6: c[58]= 116.000000
Thread 6: c[59]= 118.000000
Thread 6: c[60]= 120.000000
Thread 6: c[61]= 122.000000
Thread 6: c[62]= 124.000000
Thread 6: c[63]= 126.000000
Thread 3 starting...
Number of threads = 8
Thread 0 starting...
Execution time: 0.012305[1] + Done
```

d)
For the static scheduling, it will run based on the chunk size that was assigned. For our program, each thread will run at least 10 iterations. We run the for loop 100 times so divide that with the chunk size which gives us 10 chunks to distribute evenly to the threads. Since my system only has 8 physical threads, the 9th and 10th iterations will come around and be run by threads 0 and 1.

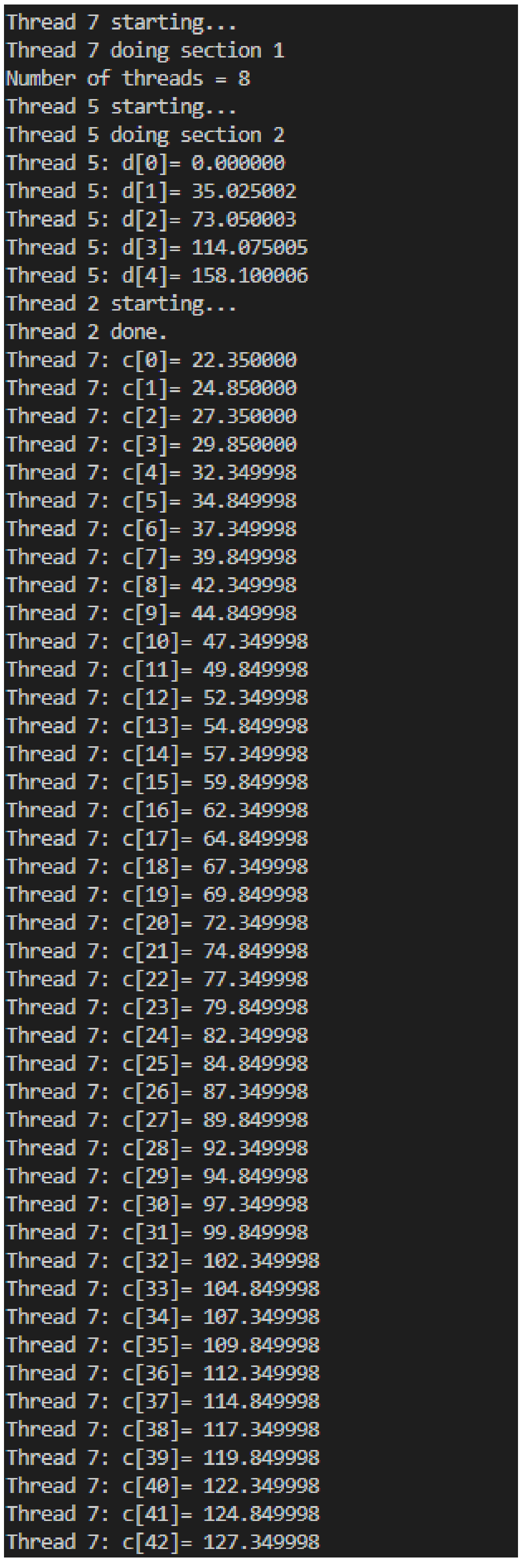
For the dynamic scheduling, the workload will also be divided with the chunk size to create chunks for distribution to the thread. However, it is different compared to static scheduling in that there is no order in assigning chunks to threads so it is quite “random”. We can see some threads do more work and some don’t even run so it is not fair or balanced.

For the guided scheduling, looks similar to dynamic scheduling in terms of unorderly randomly assigning workload to threads. One key difference is the chunk size changes during execution. It starts with a bigger chunk size and decreases as it goes through iterations.

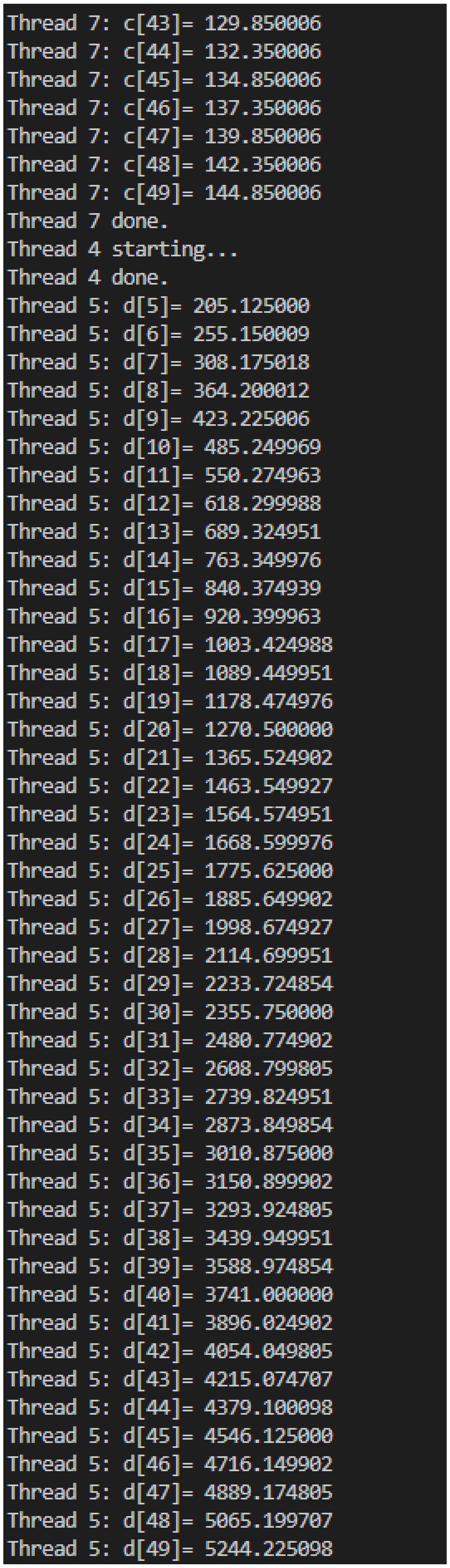
For the execution time, the guided and dynamic scheduling are similar to each other. The static scheduling ran a bit faster compared to the other two.

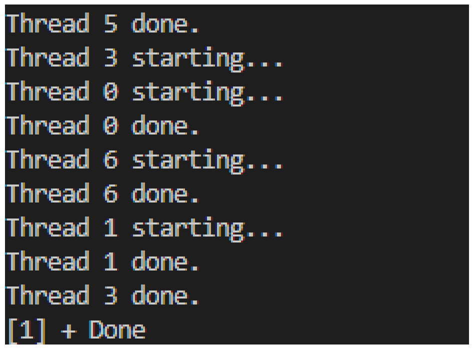
1.3:

a)



```
Thread 7 starting...
Thread 7 doing section 1
Number of threads = 8
Thread 5 starting...
Thread 5 doing section 2
Thread 5: d[0]= 0.000000
Thread 5: d[1]= 35.025002
Thread 5: d[2]= 73.050003
Thread 5: d[3]= 114.075005
Thread 5: d[4]= 158.100006
Thread 2 starting...
Thread 2 done.
Thread 7: c[0]= 22.350000
Thread 7: c[1]= 24.850000
Thread 7: c[2]= 27.350000
Thread 7: c[3]= 29.850000
Thread 7: c[4]= 32.349998
Thread 7: c[5]= 34.849998
Thread 7: c[6]= 37.349998
Thread 7: c[7]= 39.849998
Thread 7: c[8]= 42.349998
Thread 7: c[9]= 44.849998
Thread 7: c[10]= 47.349998
Thread 7: c[11]= 49.849998
Thread 7: c[12]= 52.349998
Thread 7: c[13]= 54.849998
Thread 7: c[14]= 57.349998
Thread 7: c[15]= 59.849998
Thread 7: c[16]= 62.349998
Thread 7: c[17]= 64.849998
Thread 7: c[18]= 67.349998
Thread 7: c[19]= 69.849998
Thread 7: c[20]= 72.349998
Thread 7: c[21]= 74.849998
Thread 7: c[22]= 77.349998
Thread 7: c[23]= 79.849998
Thread 7: c[24]= 82.349998
Thread 7: c[25]= 84.849998
Thread 7: c[26]= 87.349998
Thread 7: c[27]= 89.849998
Thread 7: c[28]= 92.349998
Thread 7: c[29]= 94.849998
Thread 7: c[30]= 97.349998
Thread 7: c[31]= 99.849998
Thread 7: c[32]= 102.349998
Thread 7: c[33]= 104.849998
Thread 7: c[34]= 107.349998
Thread 7: c[35]= 109.849998
Thread 7: c[36]= 112.349998
Thread 7: c[37]= 114.849998
Thread 7: c[38]= 117.349998
Thread 7: c[39]= 119.849998
Thread 7: c[40]= 122.349998
Thread 7: c[41]= 124.849998
Thread 7: c[42]= 127.349998
```


```
Thread 7: c[43]= 129.850006
Thread 7: c[44]= 132.350006
Thread 7: c[45]= 134.850006
Thread 7: c[46]= 137.350006
Thread 7: c[47]= 139.850006
Thread 7: c[48]= 142.350006
Thread 7: c[49]= 144.850006
Thread 7 done.
Thread 4 starting...
Thread 4 done.
Thread 5: d[5]= 205.125000
Thread 5: d[6]= 255.150009
Thread 5: d[7]= 308.175018
Thread 5: d[8]= 364.200012
Thread 5: d[9]= 423.225006
Thread 5: d[10]= 485.249969
Thread 5: d[11]= 550.274963
Thread 5: d[12]= 618.299988
Thread 5: d[13]= 689.324951
Thread 5: d[14]= 763.349976
Thread 5: d[15]= 840.374939
Thread 5: d[16]= 920.399963
Thread 5: d[17]= 1003.424988
Thread 5: d[18]= 1089.449951
Thread 5: d[19]= 1178.474976
Thread 5: d[20]= 1270.500000
Thread 5: d[21]= 1365.524902
Thread 5: d[22]= 1463.549927
Thread 5: d[23]= 1564.574951
Thread 5: d[24]= 1668.599976
Thread 5: d[25]= 1775.625000
Thread 5: d[26]= 1885.649902
Thread 5: d[27]= 1998.674927
Thread 5: d[28]= 2114.699951
Thread 5: d[29]= 2233.724854
Thread 5: d[30]= 2355.750000
Thread 5: d[31]= 2480.774902
Thread 5: d[32]= 2608.799805
Thread 5: d[33]= 2739.824951
Thread 5: d[34]= 2873.849854
Thread 5: d[35]= 3010.875000
Thread 5: d[36]= 3150.899902
Thread 5: d[37]= 3293.924805
Thread 5: d[38]= 3439.949951
Thread 5: d[39]= 3588.974854
Thread 5: d[40]= 3741.000000
Thread 5: d[41]= 3896.024902
Thread 5: d[42]= 4054.049805
Thread 5: d[43]= 4215.074707
Thread 5: d[44]= 4379.100098
Thread 5: d[45]= 4546.125000
Thread 5: d[46]= 4716.149902
Thread 5: d[47]= 4889.174805
Thread 5: d[48]= 5065.199707
Thread 5: d[49]= 5244.225098
```

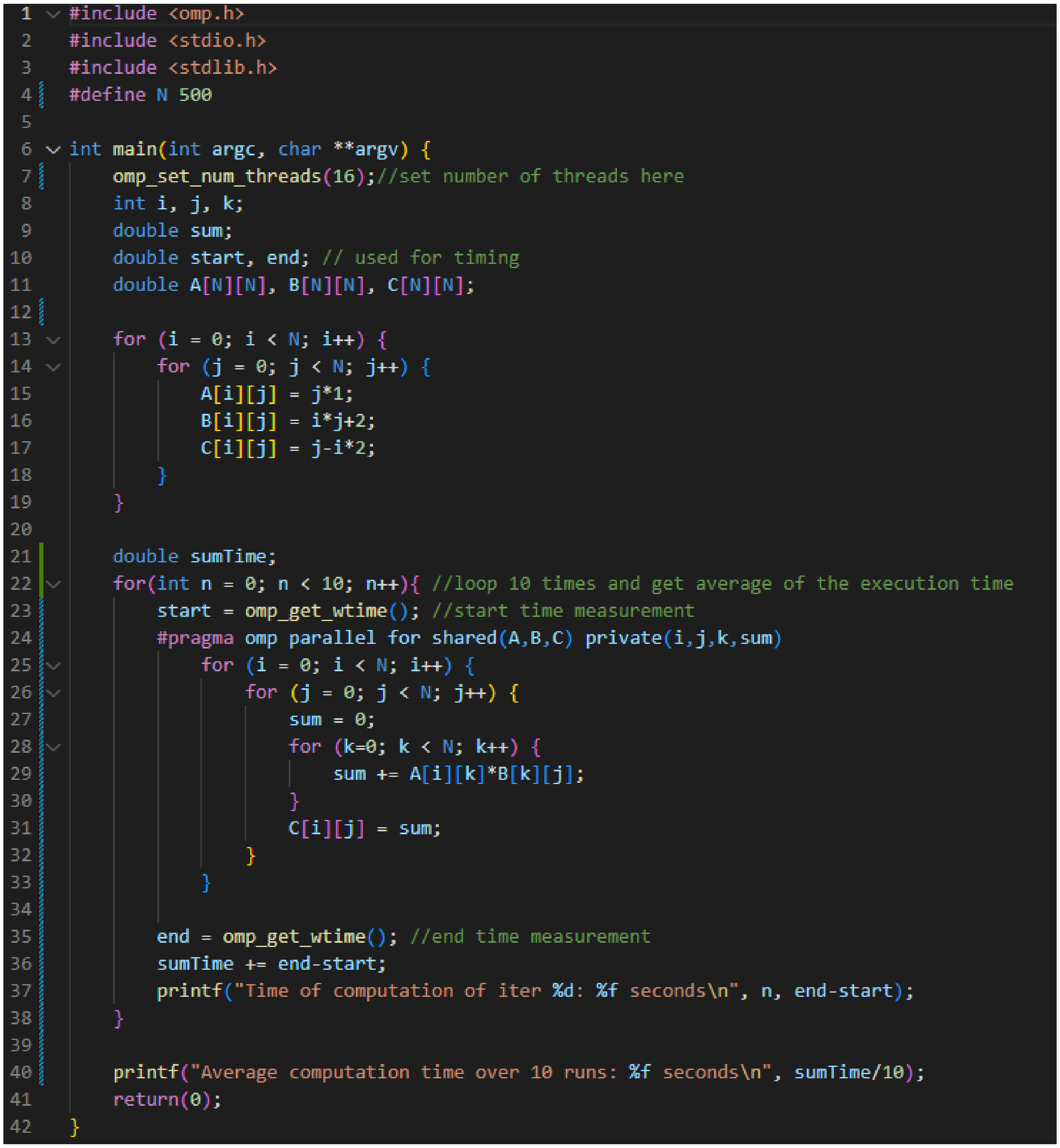


```
Thread 5 done.
Thread 3 starting...
Thread 0 starting...
Thread 0 done.
Thread 6 starting...
Thread 6 done.
Thread 1 starting...
Thread 1 done.
Thread 3 done.
[1] + Done
```

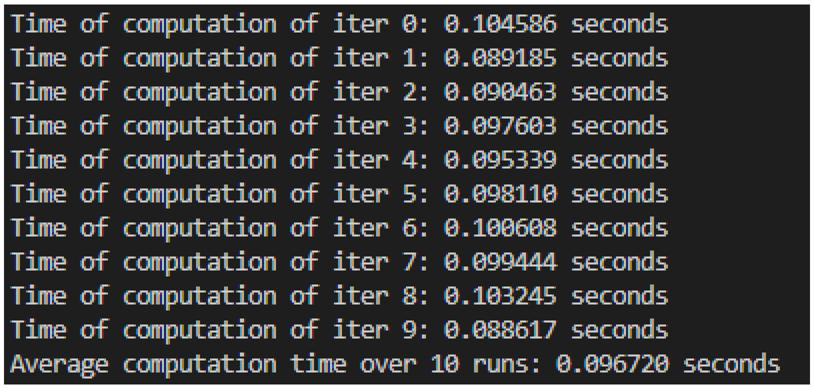
b)
The most obvious feature of using section construction is that each declared section is assigned to a thread. The threads can also interleave with each other as the program runs. Since our program only has 2 sections, only 2 threads are allowed to be assigned to the sections which thread 7 is assigned to section 1 and thread 5 is assigned to section 2 as shown above.
Also, we can see that the shared variables are a, b, c, d, nthreads which are accessed by both sections. The private variables are i, tid.

1.4:

a) Outer loop, N=500

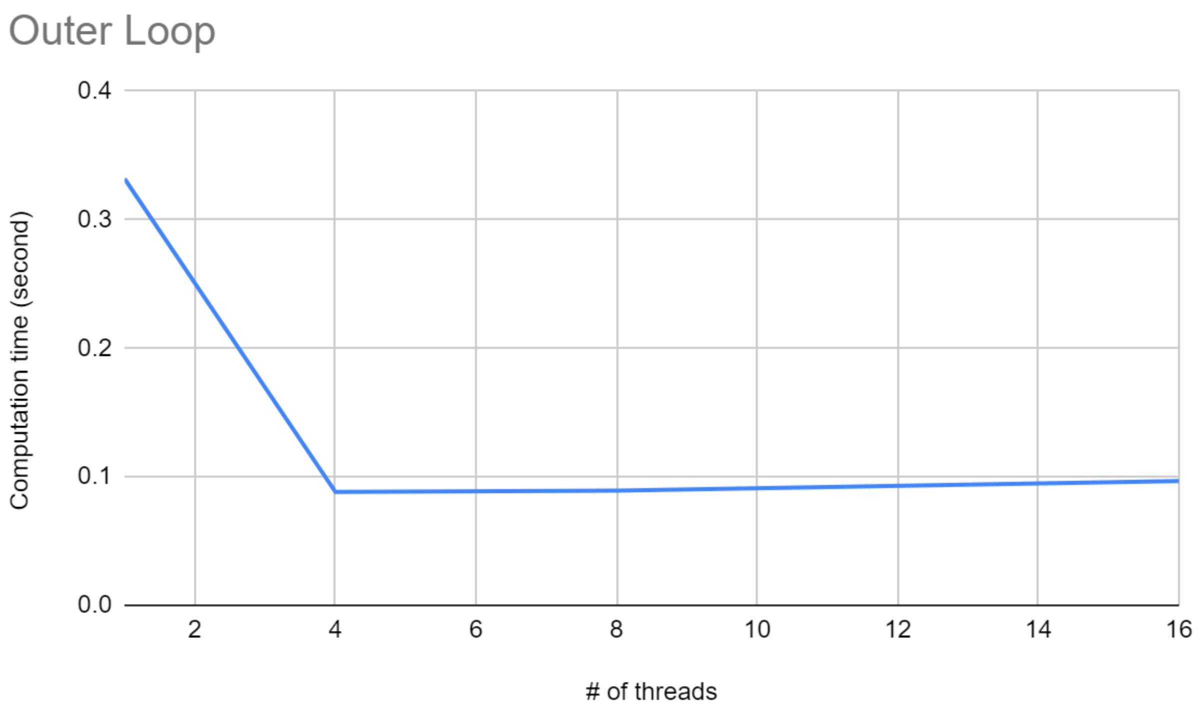


```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 500

int main(int argc, char **argv) {
    omp_set_num_threads(16);//set number of threads here
    int i, j, k;
    double sum;
    double start, end; // used for timing
    double A[N][N], B[N][N], C[N][N];

    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            A[i][j] = j*1;
            B[i][j] = i*j+2;
            C[i][j] = j-i*2;
        }
    }

    double sumTime;
    for(int n = 0; n < 10; n++){ //loop 10 times and get average of the execution time
        start = omp_get_wtime(); //start time measurement
        #pragma omp parallel for shared(A,B,C) private(i,j,k,sum)
            for (i = 0; i < N; i++) {
                for (j = 0; j < N; j++) {
                    sum = 0;
                    for (k=0; k < N; k++) {
                        sum += A[i][k]*B[k][j];
                    }
                    C[i][j] = sum;
                }
            }

        end = omp_get_wtime(); //end time measurement
        sumTime += end-start;
        printf("Time of computation of iter %d: %f seconds\n", n, end-start);
    }

    printf("Average computation time over 10 runs: %f seconds\n", sumTime/10);
    return(0);
}
```

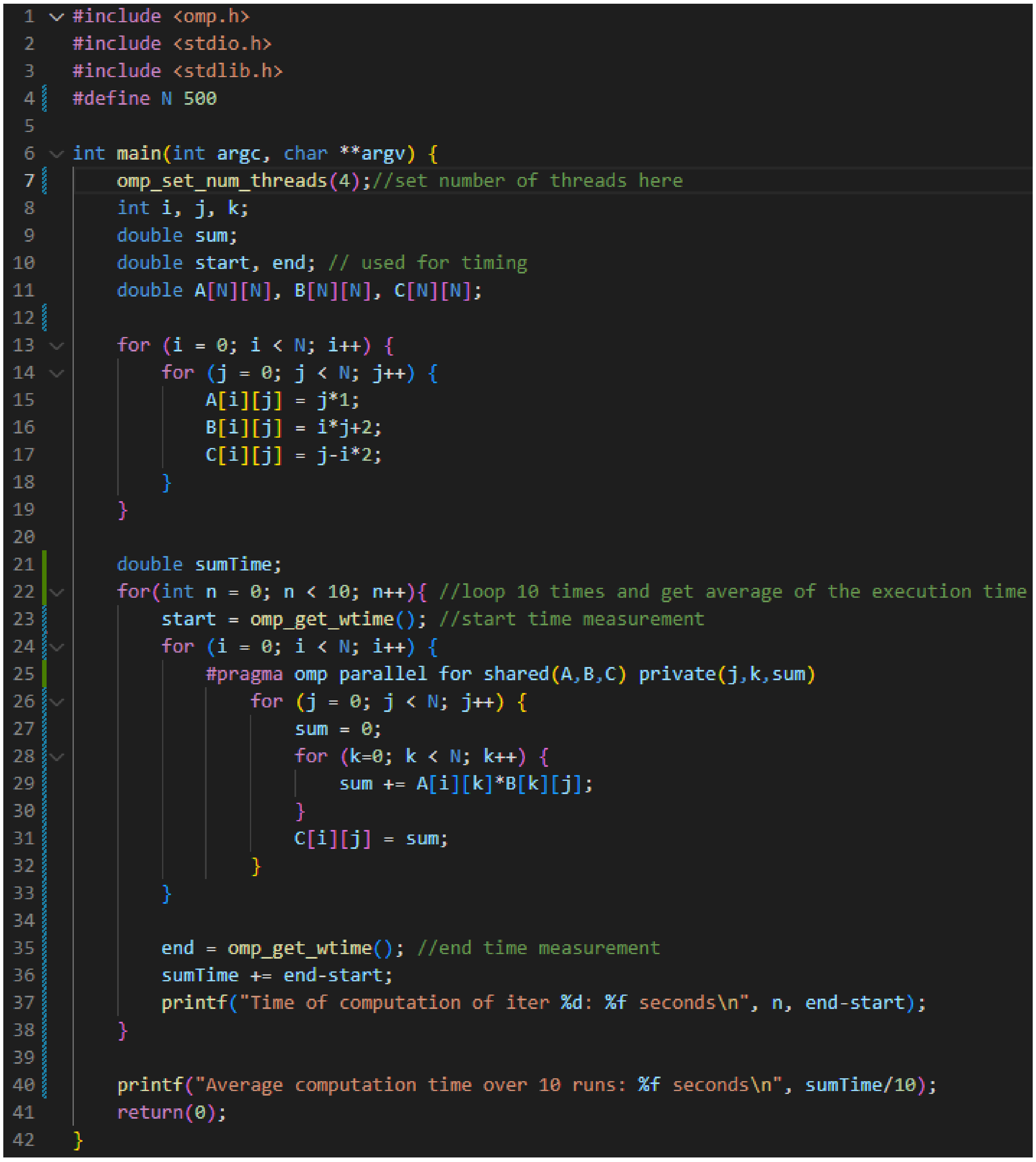


```
Time of computation of iter 0: 0.104586 seconds
Time of computation of iter 1: 0.089185 seconds
Time of computation of iter 2: 0.090463 seconds
Time of computation of iter 3: 0.097603 seconds
Time of computation of iter 4: 0.095339 seconds
Time of computation of iter 5: 0.098110 seconds
Time of computation of iter 6: 0.100608 seconds
Time of computation of iter 7: 0.099444 seconds
Time of computation of iter 8: 0.103245 seconds
Time of computation of iter 9: 0.088617 seconds
Average computation time over 10 runs: 0.096720 seconds
```

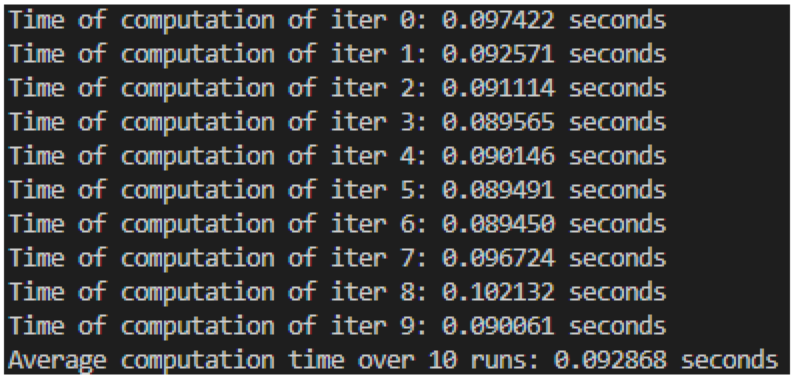
# of threads	Computation time (second)
1	0.331962
4	0.088154
8	0.089129
16	0.096720



b) Middle loop, N=500

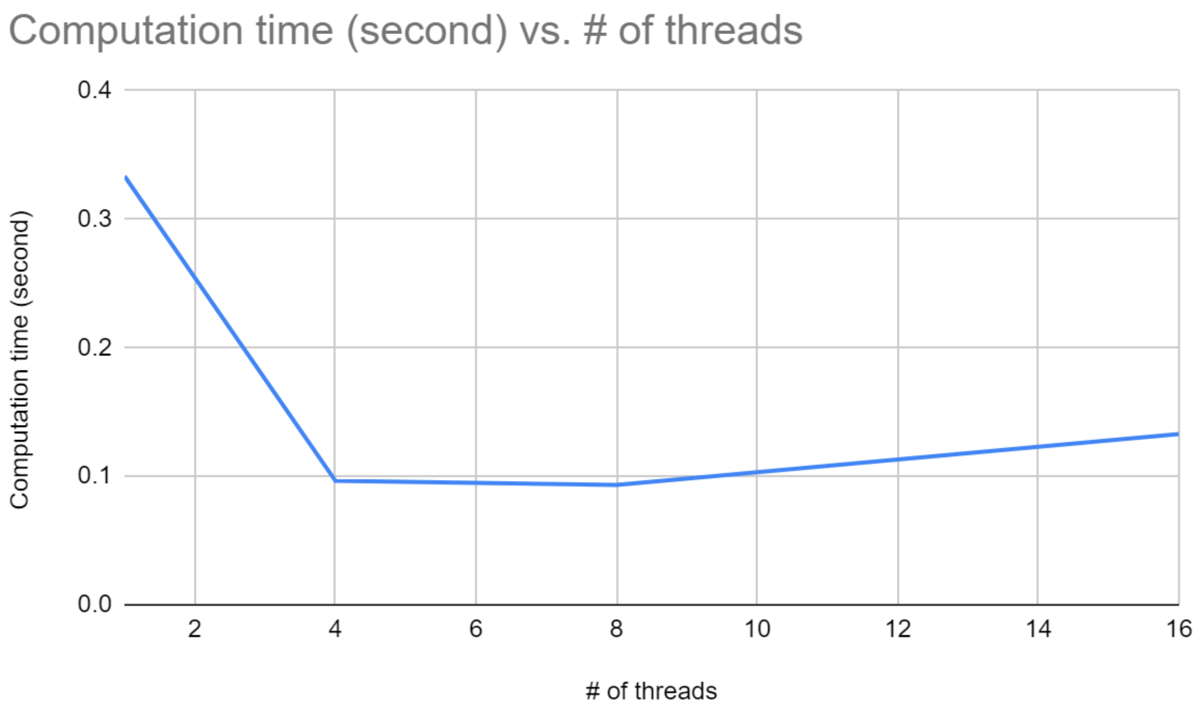


```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 500

int main(int argc, char **argv) {
    omp_set_num_threads(4);//set number of threads here
    int i, j, k;
    double sum;
    double start, end; // used for timing
    double A[N][N], B[N][N], C[N][N];

    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            A[i][j] = j*1;
            B[i][j] = i*j+2;
            C[i][j] = j-i*2;
        }
    }

    double sumTime;
    for(int n = 0; n < 10; n++){ //loop 10 times and get average of the execution time
        start = omp_get_wtime(); //start time measurement
        for (i = 0; i < N; i++) {
            #pragma omp parallel for shared(A,B,C) private(j,k,sum)
                for (j = 0; j < N; j++) {
                    sum = 0;
                    for (k=0; k < N; k++) {
                        sum += A[i][k]*B[k][j];
                    }
                    C[i][j] = sum;
                }
        }

        end = omp_get_wtime(); //end time measurement
        sumTime += end-start;
        printf("Time of computation of iter %d: %f seconds\n", n, end-start);
    }

    printf("Average computation time over 10 runs: %f seconds\n", sumTime/10);
    return(0);
}
```



```
Time of computation of iter 0: 0.097422 seconds
Time of computation of iter 1: 0.092571 seconds
Time of computation of iter 2: 0.091114 seconds
Time of computation of iter 3: 0.089565 seconds
Time of computation of iter 4: 0.090146 seconds
Time of computation of iter 5: 0.089491 seconds
Time of computation of iter 6: 0.089450 seconds
Time of computation of iter 7: 0.096724 seconds
Time of computation of iter 8: 0.102132 seconds
Time of computation of iter 9: 0.090061 seconds
Average computation time over 10 runs: 0.092868 seconds
```

# of threads	Computation time (second)
1	0.333530
4	0.096440
8	0.093328
16	0.132786



c)
From the result above, we can see that we got faster computation time with more threads. However, it seems like the most optimal numbers of threads are 4 or 8. It is because of more overhead and more communication when running with more threads.