University of California, Riverside
Department of Computer Science and Engineering

Title: Programming Assignment 1

Course: Multiprocessor Architecture and Programming

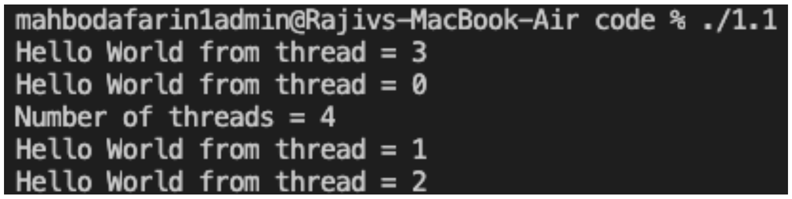
Mahbod Afarin

Winter 2021

1.1 “hello world” program

(a) Screenshot from running the hello world program on your computer and explanation of output and thread order.

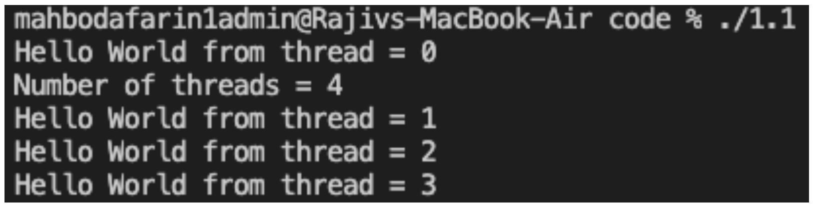
Figure 1 shows a screenshot from running the hello world program on my computer. As it is clear from the figure, since I did not set a specific number of threads, the system set the number of threads to 4 by default.



```
mahbodafarin1admin@Rajivs-MacBook-Air code % ./1.1
Hello World from thread = 3
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 1
Hello World from thread = 2
```

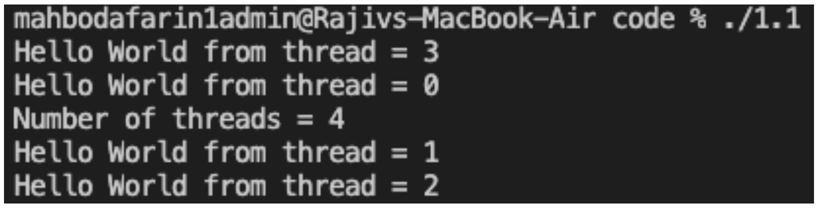
Figure 1: Screenshot of the output for first run

Then I executed the program four times. Figure 1, figure 2, and figure 3 show screenshot of the output for second, third, and fourth execution respectively. As it is clear from the figures, the order of the output threads is different in each execution. The reason for that is the threads are interleaving in time and each time I run that, I will get different interleaving of those threads. Thread will context switch and the order of this context switch is different in each run.



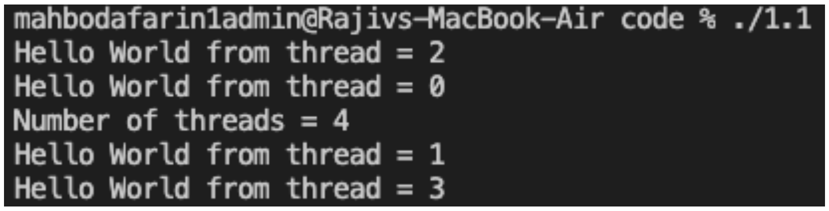
```
mahbodafarin1admin@Rajivs-MacBook-Air code % ./1.1
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 1
Hello World from thread = 2
Hello World from thread = 3
```

Figure 2: Screenshot of the output for second run



```
mahbodafarin1admin@Rajivs-MacBook-Air code % ./1.1
Hello World from thread = 3
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 1
Hello World from thread = 2
```

Figure 3: Screenshot of the output for third run

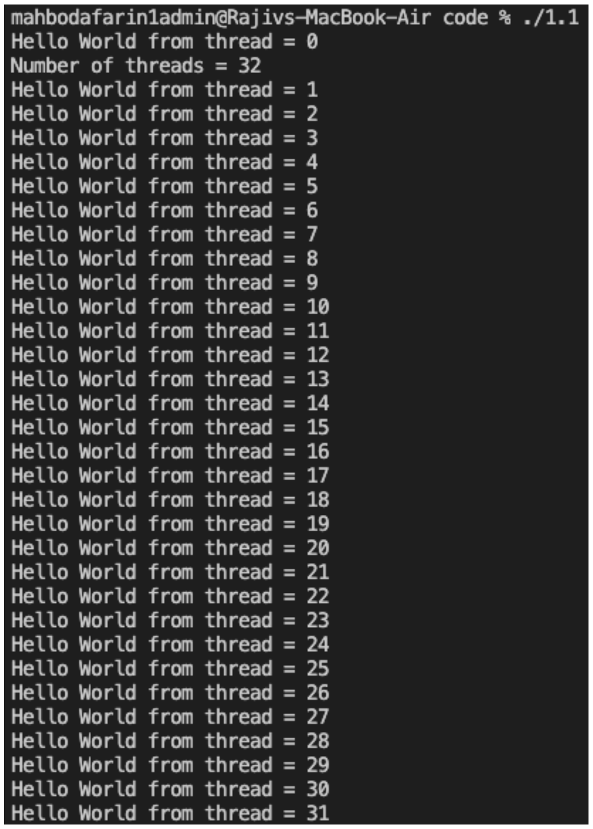


```
mahbodafarin1admin@Rajivs-MacBook-Air code % ./1.1
Hello World from thread = 2
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 1
Hello World from thread = 3
```

Figure 4: Screenshot of the output for fourth run

(b) Screenshots of the output from running the program with 32 threads and explanation of what you have observed.

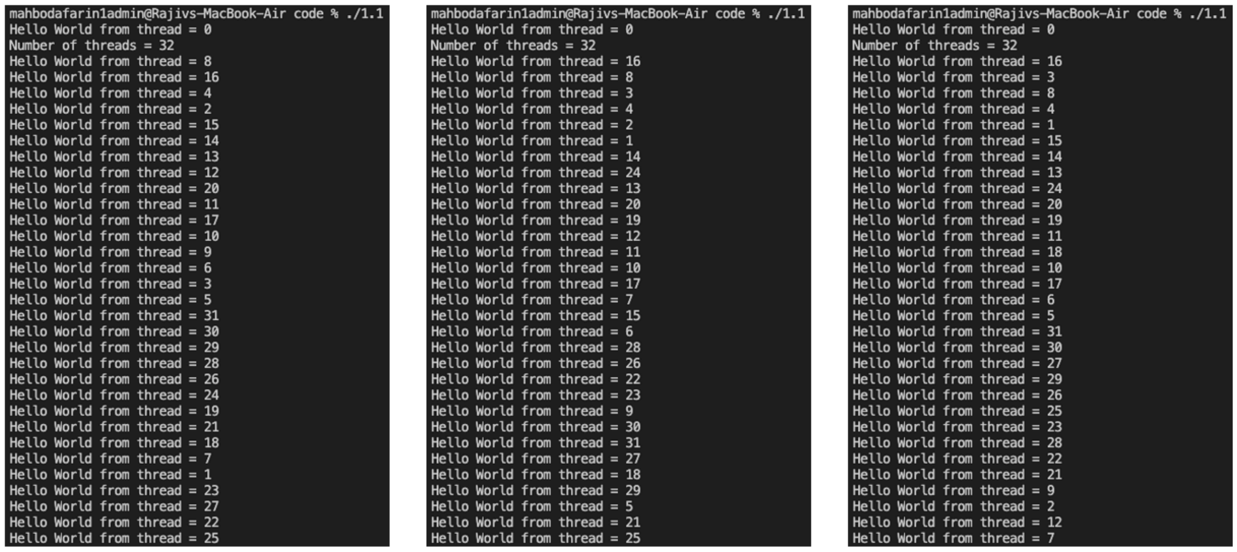
Using *omp_set_num_threads (32)* command before *pragma*, we can change the thread numbers to 32. Now, 32 threads will execute the program. Therefore, we have a team of 32 threads. Each thread will run the code on that structure block and each thread is running redundantly the same code. Figure 5 shows the output of the program when we set the number of the threads to 32. As we can see in the figure, the number of the threads is 32.



```
mahbodafarin1admin@Rajivs-MacBook-Air code % ./1.1
Hello World from thread = 0
Number of threads = 32
Hello World from thread = 1
Hello World from thread = 2
Hello World from thread = 3
Hello World from thread = 4
Hello World from thread = 5
Hello World from thread = 6
Hello World from thread = 7
Hello World from thread = 8
Hello World from thread = 9
Hello World from thread = 10
Hello World from thread = 11
Hello World from thread = 12
Hello World from thread = 13
Hello World from thread = 14
Hello World from thread = 15
Hello World from thread = 16
Hello World from thread = 17
Hello World from thread = 18
Hello World from thread = 19
Hello World from thread = 20
Hello World from thread = 21
Hello World from thread = 22
Hello World from thread = 23
Hello World from thread = 24
Hello World from thread = 25
Hello World from thread = 26
Hello World from thread = 27
Hello World from thread = 28
Hello World from thread = 29
Hello World from thread = 30
Hello World from thread = 31
```

Figure 5: Screenshot of the output for first run with 32 thread

Figure 6, shows the screenshot for second, third, and fourth run with 32 threads. As we can see in the figure, the order of threads is also different in this case because of the same reason.



```
mahbodafarin1admin@Rajivs-MacBook-Air code % ./1.1
Hello World from thread = 0
Number of threads = 32
Hello World from thread = 8
Hello World from thread = 16
Hello World from thread = 4
Hello World from thread = 2
Hello World from thread = 15
Hello World from thread = 14
Hello World from thread = 13
Hello World from thread = 12
Hello World from thread = 20
Hello World from thread = 11
Hello World from thread = 17
Hello World from thread = 10
Hello World from thread = 9
Hello World from thread = 6
Hello World from thread = 3
Hello World from thread = 5
Hello World from thread = 31
Hello World from thread = 30
Hello World from thread = 29
Hello World from thread = 28
Hello World from thread = 26
Hello World from thread = 24
Hello World from thread = 19
Hello World from thread = 21
Hello World from thread = 18
Hello World from thread = 7
Hello World from thread = 1
Hello World from thread = 23
Hello World from thread = 27
Hello World from thread = 22
Hello World from thread = 25
```

(a)

```
mahbodafarin1admin@Rajivs-MacBook-Air code % ./1.1
Hello World from thread = 0
Number of threads = 32
Hello World from thread = 16
Hello World from thread = 8
Hello World from thread = 3
Hello World from thread = 4
Hello World from thread = 2
Hello World from thread = 1
Hello World from thread = 14
Hello World from thread = 24
Hello World from thread = 13
Hello World from thread = 20
Hello World from thread = 19
Hello World from thread = 12
Hello World from thread = 11
Hello World from thread = 10
Hello World from thread = 17
Hello World from thread = 7
Hello World from thread = 15
Hello World from thread = 6
Hello World from thread = 28
Hello World from thread = 26
Hello World from thread = 22
Hello World from thread = 23
Hello World from thread = 9
Hello World from thread = 30
Hello World from thread = 31
Hello World from thread = 27
Hello World from thread = 18
Hello World from thread = 29
Hello World from thread = 5
Hello World from thread = 21
Hello World from thread = 25
```

(b)

```
mahbodafarin1admin@Rajivs-MacBook-Air code % ./1.1
Hello World from thread = 0
Number of threads = 32
Hello World from thread = 16
Hello World from thread = 3
Hello World from thread = 8
Hello World from thread = 4
Hello World from thread = 1
Hello World from thread = 15
Hello World from thread = 14
Hello World from thread = 13
Hello World from thread = 24
Hello World from thread = 20
Hello World from thread = 19
Hello World from thread = 11
Hello World from thread = 18
Hello World from thread = 10
Hello World from thread = 17
Hello World from thread = 6
Hello World from thread = 5
Hello World from thread = 31
Hello World from thread = 30
Hello World from thread = 27
Hello World from thread = 29
Hello World from thread = 26
Hello World from thread = 25
Hello World from thread = 23
Hello World from thread = 28
Hello World from thread = 22
Hello World from thread = 21
Hello World from thread = 9
Hello World from thread = 2
Hello World from thread = 12
Hello World from thread = 7
```

(c)

Figure 6: Screenshot of the output for second, third, and fourth run with 32 thread

1.2 work-sharing

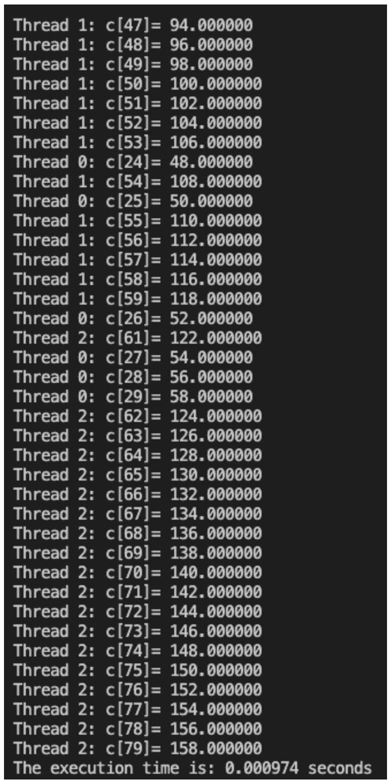
(a) A copy of the source program with timing added.

The below code shows the program with timing added. As it is clear from the code, I defined *start* and *end* variables with indicate start time and end time respectively. Then I put the parallel part of the program between *start = omp_get_wtime();* and *end = omp_get_wtime();*. Finally, I found the execution time by subtracting the start time and the end time.

```
#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, end;

    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0; // initialize arrays

    chunk = CHUNKSIZE;

    start = omp_get_wtime();
    #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 */
    end = omp_get_wtime();
    printf("The execution time is: %f seconds\n", end - start);
    return(0);
}
```

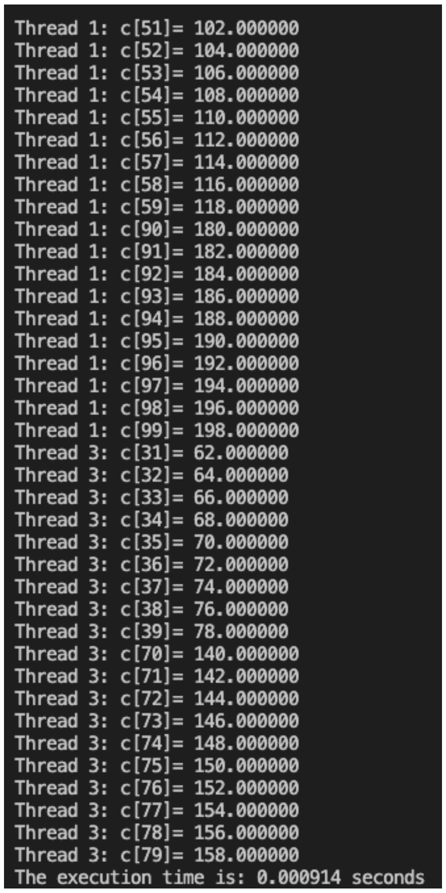
(b) Screenshot from compiling and running the program with the original dynamic scheduling.

Figure 7 and figure 8 show the screenshot of the running the program with dynamic and static scheduling.



```
Thread 1: c[47]= 94.000000
Thread 1: c[48]= 96.000000
Thread 1: c[49]= 98.000000
Thread 1: c[50]= 100.000000
Thread 1: c[51]= 102.000000
Thread 1: c[52]= 104.000000
Thread 1: c[53]= 106.000000
Thread 0: c[24]= 48.000000
Thread 1: c[54]= 108.000000
Thread 0: c[25]= 50.000000
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 0: c[26]= 52.000000
Thread 2: c[61]= 122.000000
Thread 0: c[27]= 54.000000
Thread 0: c[28]= 56.000000
Thread 0: c[29]= 58.000000
Thread 2: c[62]= 124.000000
Thread 2: c[63]= 126.000000
Thread 2: c[64]= 128.000000
Thread 2: c[65]= 130.000000
Thread 2: c[66]= 132.000000
Thread 2: c[67]= 134.000000
Thread 2: c[68]= 136.000000
Thread 2: c[69]= 138.000000
Thread 2: c[70]= 140.000000
Thread 2: c[71]= 142.000000
Thread 2: c[72]= 144.000000
Thread 2: c[73]= 146.000000
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
The execution time is: 0.000974 seconds
```

Figure 7: Screenshot from running the program with the dynamic scheduling

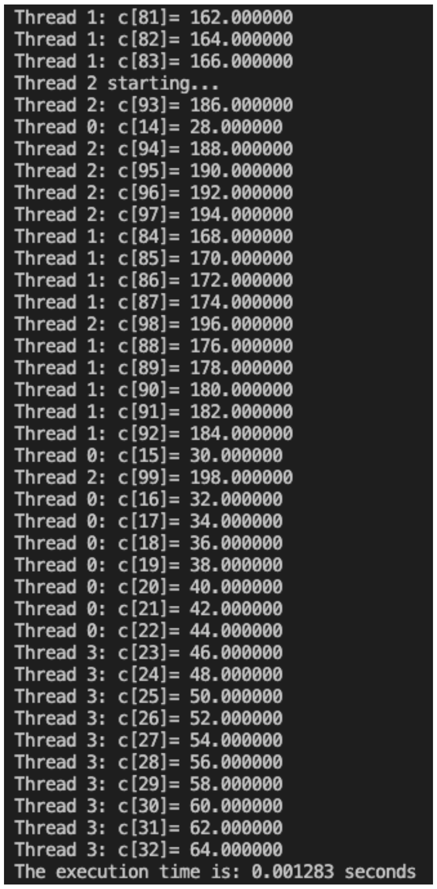


```
Thread 1: c[51]= 102.000000
Thread 1: c[52]= 104.000000
Thread 1: c[53]= 106.000000
Thread 1: c[54]= 108.000000
Thread 1: c[55]= 110.000000
Thread 1: c[56]= 112.000000
Thread 1: c[57]= 114.000000
Thread 1: c[58]= 116.000000
Thread 1: c[59]= 118.000000
Thread 1: c[90]= 180.000000
Thread 1: c[91]= 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 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 3: c[70]= 140.000000
Thread 3: c[71]= 142.000000
Thread 3: c[72]= 144.000000
Thread 3: c[73]= 146.000000
Thread 3: c[74]= 148.000000
Thread 3: c[75]= 150.000000
Thread 3: c[76]= 152.000000
Thread 3: c[77]= 154.000000
Thread 3: c[78]= 156.000000
Thread 3: c[79]= 158.000000
The execution time is: 0.000914 seconds
```

Figure 8: Screenshot from running the program with the static scheduling

(c) Screenshots from running the program with static and with guided scheduling

Figure 9 shows the screenshot of the running the program with guided scheduling.



```
Thread 1: c[81]= 162.000000
Thread 1: c[82]= 164.000000
Thread 1: c[83]= 166.000000
Thread 2 starting...
Thread 2: c[93]= 186.000000
Thread 0: c[14]= 28.000000
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 1: c[84]= 168.000000
Thread 1: c[85]= 170.000000
Thread 1: c[86]= 172.000000
Thread 1: c[87]= 174.000000
Thread 2: c[98]= 196.000000
Thread 1: c[88]= 176.000000
Thread 1: c[89]= 178.000000
Thread 1: c[90]= 180.000000
Thread 1: c[91]= 182.000000
Thread 1: c[92]= 184.000000
Thread 0: c[15]= 30.000000
Thread 2: c[99]= 198.000000
Thread 0: c[16]= 32.000000
Thread 0: c[17]= 34.000000
Thread 0: c[18]= 36.000000
Thread 0: c[19]= 38.000000
Thread 0: c[20]= 40.000000
Thread 0: c[21]= 42.000000
Thread 0: c[22]= 44.000000
Thread 3: c[23]= 46.000000
Thread 3: c[24]= 48.000000
Thread 3: c[25]= 50.000000
Thread 3: c[26]= 52.000000
Thread 3: c[27]= 54.000000
Thread 3: c[28]= 56.000000
Thread 3: c[29]= 58.000000
Thread 3: c[30]= 60.000000
Thread 3: c[31]= 62.000000
Thread 3: c[32]= 64.000000
The execution time is: 0.001283 seconds
```

Figure 9: Screenshot from running the program with the guided scheduling

(d) Your conclusions about the different scheduling approaches.

The ***#pragma omp for*** will break up a loop between threads and in this way, a specific work will share between different threads. The main question is that how it will break the look between different threads. We have three kinds of the scheduling, static, dynamic, and guided.

- Static scheduling: in static scheduling means at **compile time** figure out the close form expression that takes loop iterations and breaks them up into blocks and then parcels those out to threads. So, with ***#pragma omp for schedule(static,chunk)*** we can set the type of the scheduling and chunk is defining the chunk size of the divided work. The static scheduling is pre-determined and predictable by the programmer.
- Dynamic scheduling: it says that take the loop iterations and put them into a logical task queue and then go off and grab an iteration the thread finishes the work on that iteration and then it goes back and grab the next iteration. The main point is that it is deciding at **run time**, not compile time. Like static scheduling, we can determine the chunk size in dynamic scheduling. Dynamic scheduling is unpredictable and we have highly variable work per iteration.

- Guided scheduling: in this scheduling, threads dynamically grab blocks of iterations. The size of the block starts large and shrink down to size of the chunk as the calculation proceed. This scheduling is no use very often these days.

Table 1 shows the run time of static, dynamic, and guided scheduling. The run time for static scheduling is 0.000914 seconds and it has the best runtime compared to the other scheduling techniques. The reason is that the static scheduling works better for loops where each iteration takes roughly equal time, and in this way, we have little overhead. In the for loop, all the tasks have roughly equal time and it is only a simple addition, and that is why static scheduling has the best run time in this example. The dynamic scheduling has an overhead in runtime for scheduling and in this case, because the tasks have roughly equal time, we only have an overhead in runtime for dynamic scheduling and that's why the time of the dynamic scheduling is bigger than the time of the static scheduling. For loops where each iteration can take very different amounts of time, dynamic schedules, work best as the work will be split more evenly across threads. The guided scheduling has the worst execution time because this scheduling policy is similar to a dynamic schedule, except that the chunk size changes as the program runs. It begins with big chunks, but then adjusts to smaller chunk sizes if the workload is imbalanced. Therefore, for this loop, it has the worst execution time because each iteration of the loop has roughly the same execution time and taking a big chunk size at first and then shrink it to smaller chunk size does not make sense because at the beginning of the execution, we assign more workload to each thread, while that workload can be parallelized and executed with more threads easily.

Table 1: Run time of the static, dynamic, and guided scheduling

	Static Scheduling	**Dynamic Scheduling**	**Guided scheduling**
Run Time	0.000914 sec	0.000974 sec	0.001283 sec

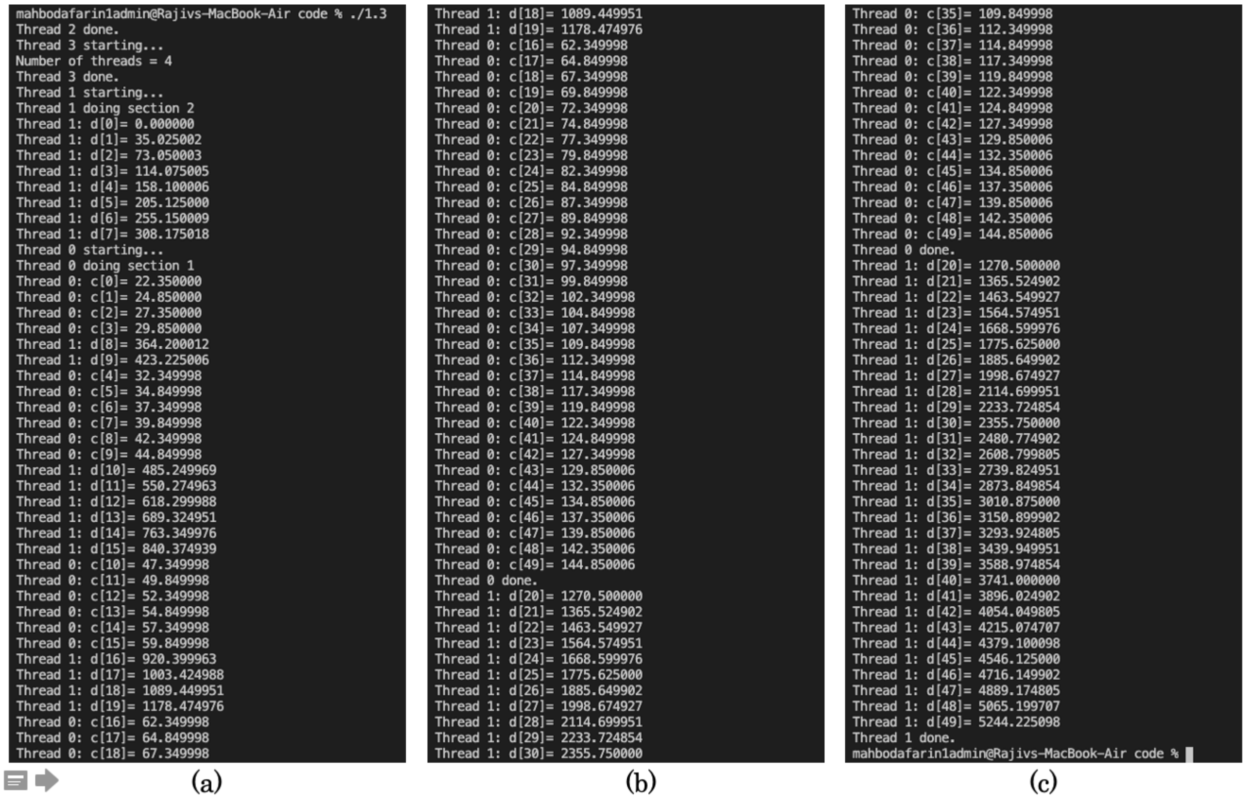
Conclusion:

- For loops where each iteration takes roughly equal time, static schedules work best because there is a little overhead.
- For loops where each iteration can take very different amounts of time, dynamic scheduling work best because the work will be split more evenly across threads.
- Guided schedule provides a trade-off between the two approach.
- Choosing the best schedule depends on understanding your loop.

1.3 Work-sharing with the sections construct

(a) Screenshot from compiling and running the program.

Figure 10 shows a screenshot from running the program.



```
mahbodafarin1admin@Rajivs-MacBook-Air code % ./1.3
Thread 2 done.
Thread 3 starting...
Number of threads = 4
Thread 3 done.
Thread 1 starting...
Thread 1 doing section 2
Thread 1: d[0]= 0.000000
Thread 1: d[1]= 35.025002
Thread 1: d[2]= 73.050003
Thread 1: d[3]= 114.075005
Thread 1: d[4]= 158.100006
Thread 1: d[5]= 205.125000
Thread 1: d[6]= 255.150009
Thread 1: d[7]= 308.175018
Thread 0 starting...
Thread 0 doing section 1
Thread 0: c[0]= 22.350000
Thread 0: c[1]= 24.850000
Thread 0: c[2]= 27.350000
Thread 0: c[3]= 29.850000
Thread 1: d[8]= 364.200012
Thread 1: d[9]= 423.225006
Thread 0: c[4]= 32.349998
Thread 0: c[5]= 34.849998
Thread 0: c[6]= 37.349998
Thread 0: c[7]= 39.849998
Thread 0: c[8]= 42.349998
Thread 0: c[9]= 44.849998
Thread 1: d[10]= 485.249969
Thread 1: d[11]= 550.274963
Thread 1: d[12]= 618.299988
Thread 1: d[13]= 689.324951
Thread 1: d[14]= 763.349976
Thread 1: d[15]= 840.374939
Thread 0: c[10]= 47.349998
Thread 0: c[11]= 49.849998
Thread 0: c[12]= 52.349998
Thread 0: c[13]= 54.849998
Thread 0: c[14]= 57.349998
Thread 0: c[15]= 59.849998
Thread 1: d[16]= 920.399963
Thread 1: d[17]= 1003.424988
Thread 1: d[18]= 1089.449951
Thread 1: d[19]= 1178.474976
Thread 0: c[16]= 62.349998
Thread 0: c[17]= 64.849998
Thread 0: c[18]= 67.349998
```

(a)

```
Thread 1: d[18]= 1089.449951
Thread 1: d[19]= 1178.474976
Thread 0: c[16]= 62.349998
Thread 0: c[17]= 64.849998
Thread 0: c[18]= 67.349998
Thread 0: c[19]= 69.849998
Thread 0: c[20]= 72.349998
Thread 0: c[21]= 74.849998
Thread 0: c[22]= 77.349998
Thread 0: c[23]= 79.849998
Thread 0: c[24]= 82.349998
Thread 0: c[25]= 84.849998
Thread 0: c[26]= 87.349998
Thread 0: c[27]= 89.849998
Thread 0: c[28]= 92.349998
Thread 0: c[29]= 94.849998
Thread 0: c[30]= 97.349998
Thread 0: c[31]= 99.849998
Thread 0: c[32]= 102.349998
Thread 0: c[33]= 104.849998
Thread 0: c[34]= 107.349998
Thread 0: c[35]= 109.849998
Thread 0: c[36]= 112.349998
Thread 0: c[37]= 114.849998
Thread 0: c[38]= 117.349998
Thread 0: c[39]= 119.849998
Thread 0: c[40]= 122.349998
Thread 0: c[41]= 124.849998
Thread 0: c[42]= 127.349998
Thread 0: c[43]= 129.850006
Thread 0: c[44]= 132.350006
Thread 0: c[45]= 134.850006
Thread 0: c[46]= 137.350006
Thread 0: c[47]= 139.850006
Thread 0: c[48]= 142.350006
Thread 0: c[49]= 144.850006
Thread 0 done.
Thread 1: d[20]= 1270.500000
Thread 1: d[21]= 1365.524902
Thread 1: d[22]= 1463.549927
Thread 1: d[23]= 1564.574951
Thread 1: d[24]= 1668.599976
Thread 1: d[25]= 1775.625000
Thread 1: d[26]= 1885.649902
Thread 1: d[27]= 1998.674927
Thread 1: d[28]= 2114.699951
Thread 1: d[29]= 2233.724854
Thread 1: d[30]= 2355.750000
```

(b)

```
Thread 0: c[35]= 109.849998
Thread 0: c[36]= 112.349998
Thread 0: c[37]= 114.849998
Thread 0: c[38]= 117.349998
Thread 0: c[39]= 119.849998
Thread 0: c[40]= 122.349998
Thread 0: c[41]= 124.849998
Thread 0: c[42]= 127.349998
Thread 0: c[43]= 129.850006
Thread 0: c[44]= 132.350006
Thread 0: c[45]= 134.850006
Thread 0: c[46]= 137.350006
Thread 0: c[47]= 139.850006
Thread 0: c[48]= 142.350006
Thread 0: c[49]= 144.850006
Thread 0 done.
Thread 1: d[20]= 1270.500000
Thread 1: d[21]= 1365.524902
Thread 1: d[22]= 1463.549927
Thread 1: d[23]= 1564.574951
Thread 1: d[24]= 1668.599976
Thread 1: d[25]= 1775.625000
Thread 1: d[26]= 1885.649902
Thread 1: d[27]= 1998.674927
Thread 1: d[28]= 2114.699951
Thread 1: d[29]= 2233.724854
Thread 1: d[30]= 2355.750000
Thread 1: d[31]= 2480.774902
Thread 1: d[32]= 2608.799805
Thread 1: d[33]= 2739.824951
Thread 1: d[34]= 2873.849854
Thread 1: d[35]= 3010.875000
Thread 1: d[36]= 3150.899902
Thread 1: d[37]= 3293.924805
Thread 1: d[38]= 3439.949951
Thread 1: d[39]= 3588.974854
Thread 1: d[40]= 3741.000000
Thread 1: d[41]= 3896.024902
Thread 1: d[42]= 4054.049805
Thread 1: d[43]= 4215.074707
Thread 1: d[44]= 4379.100098
Thread 1: d[45]= 4546.125000
Thread 1: d[46]= 4716.149902
Thread 1: d[47]= 4889.174805
Thread 1: d[48]= 5065.199707
Thread 1: d[49]= 5244.225098
Thread 1 done.
mahbodafarin1admin@Rajivs-MacBook-Air code %
```

(c)

Figure 10: Screenshot of running the program

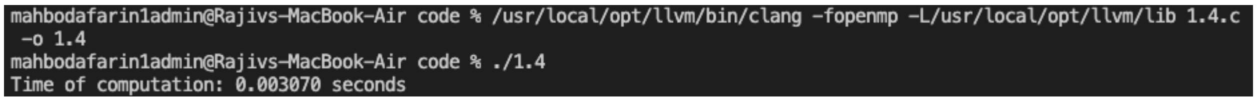
(b) Your conclusions.

As it was clear from the code, we have two sections. The first section is calculating the elements of array c and the second section is calculating the elements of array d. As it shows in the figure 10, thread 0 is responsible to calculate the results of the first section (c array) and thread 1 is responsible to calculate the second section (d array). There were 2 sections and one thread was responsible for calculating the result of the section one and some threads was responsible for calculating the results of the section two and we had context switching between the threads of those two sections.

In general, the section is one way to distribute different tasks to different threads. Each section marks the different block, which represents a task. The requirement is that each block must be independent of the other blocks. Then each thread executes one block at a time. Each block is executed only once by one thread. There are not any assumptions about the order of the execution of threads. We define the distribution of the tasks to the threads in the implementation phase. If there are more tasks than threads, some of the threads will execute multiple blocks. If there are more threads than tasks, some of the threads will be idle.

1.4 Matrix Multiplication

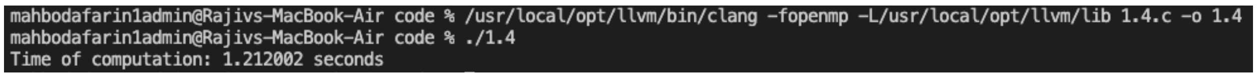
First, I executed the program. The screenshot of the output is in figure 10. As we can see in the picture, the execution time for running this program is 0.003070 seconds.



```
mahbodafarin1admin@Rajivs-MacBook-Air code % /usr/local/opt/llvm/bin/clang -fopenmp -L/usr/local/opt/llvm/lib 1.4.c
 -o 1.4
mahbodafarin1admin@Rajivs-MacBook-Air code % ./1.4
Time of computation: 0.003070 seconds
```

Figure 11: Execution of the main program

Then I should find the maximum size of the matrix which I can use in my system without getting a segmentation fault. The maximum size for N in my system was 590 and the output screenshot of running the program with this value of N has shown in the figure 10. As its clear from the figure, the execution time is 1.212002 seconds and compared to the previous time its much bigger because we increase the size of the matrix.



```
mahbodafarin1admin@Rajivs-MacBook-Air code % /usr/local/opt/llvm/bin/clang -fopenmp -L/usr/local/opt/llvm/lib 1.4.c -o 1.4
mahbodafarin1admin@Rajivs-MacBook-Air code % ./1.4
Time of computation: 1.212002 seconds
```

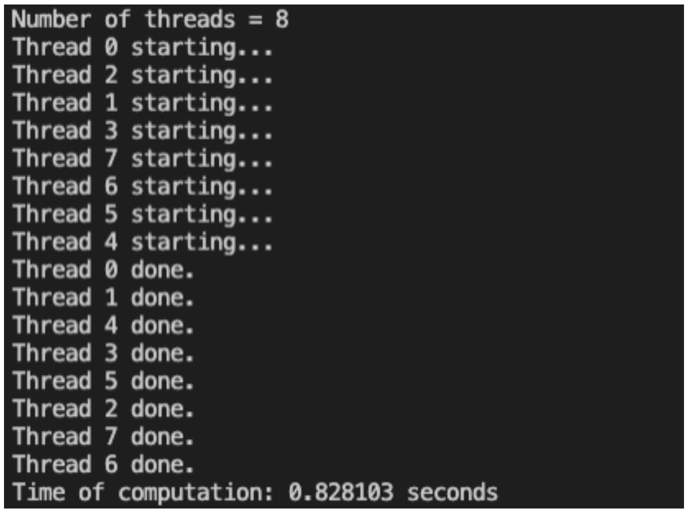
Figure 12: Screenshot of the output when the value of N is 590

(a) For the outer matrix multiplication loop

1- Source program listing: for this part I should add the necessary *pragma* to parallelize the outer *for* loop. The below code shows the parallel version of the program.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 590
#define CHUNKSIZE 10

int main(int argc, char *argv[]) {
    omp_set_num_threads(8);//set number of threads here
    int i, j, k, nthreads, tid, chunk;
    double sum;
    double start, end; // used for timing
    double A[N][N], B[N][N], C[N][N];

    chunk = CHUNKSIZE;


    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;
        }
    }

    start = omp_get_wtime(); //start time measurement
    #pragma omp parallel shared(A,B,C,nthreads) private(i,j,k,tid,sum)
    {
        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++)
        {
            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;
            }
        }
        printf("Thread %d done.\n",tid);

    }

    end = omp_get_wtime(); //end time measurement
    printf("Time of computation: %f seconds\n", end-start);
    return(0);
}
```

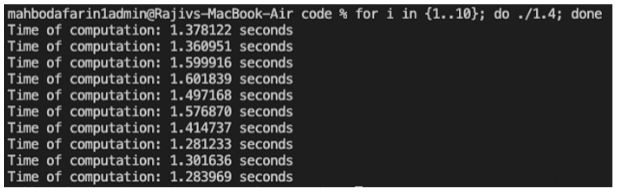
2- One screenshot from compiling and running the program: The below figure shows a screen shot from running the program for N = 590. As we can see in the figure, the execution time is 0.828103 seconds while the runtime of the sequential version of the program was 1.212002 seconds.



```
Number of threads = 8
Thread 0 starting...
Thread 2 starting...
Thread 1 starting...
Thread 3 starting...
Thread 7 starting...
Thread 6 starting...
Thread 5 starting...
Thread 4 starting...
Thread 0 done.
Thread 1 done.
Thread 4 done.
Thread 3 done.
Thread 5 done.
Thread 2 done.
Thread 7 done.
Thread 6 done.
Time of computation: 0.828103 seconds
```

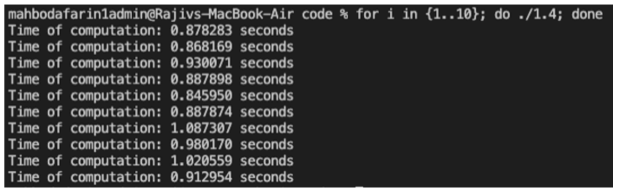
Figure 13: Screenshot from running the program

3- Graphical results of the average timings: Figure 14 shows the screenshot of the running the program for 1, 4, 8, and 16 threads.



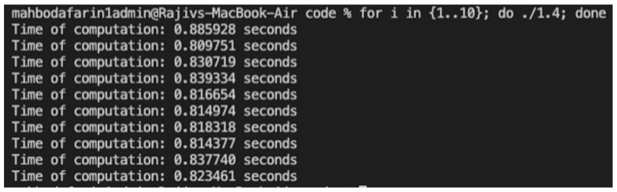
```
mahbodafarin1admin@Rajivs-MacBook-Air code % for i in {1..10}; do ./1.4; done
Time of computation: 1.378122 seconds
Time of computation: 1.360951 seconds
Time of computation: 1.599916 seconds
Time of computation: 1.601839 seconds
Time of computation: 1.497168 seconds
Time of computation: 1.576870 seconds
Time of computation: 1.414737 seconds
Time of computation: 1.281233 seconds
Time of computation: 1.301636 seconds
Time of computation: 1.283969 seconds
```

(a) Running the program with 1 thread



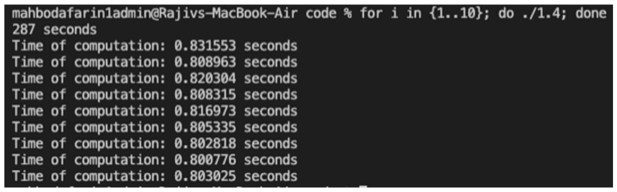
```
mahbodafarin1admin@Rajivs-MacBook-Air code % for i in {1..10}; do ./1.4; done
Time of computation: 0.878283 seconds
Time of computation: 0.868169 seconds
Time of computation: 0.930071 seconds
Time of computation: 0.887898 seconds
Time of computation: 0.845950 seconds
Time of computation: 0.887874 seconds
Time of computation: 1.087307 seconds
Time of computation: 0.980170 seconds
Time of computation: 1.020559 seconds
Time of computation: 0.912954 seconds
```

(b) Running the program with 4 thread



```
mahbodafarin1admin@Rajivs-MacBook-Air code % for i in {1..10}; do ./1.4; done
Time of computation: 0.885928 seconds
Time of computation: 0.809751 seconds
Time of computation: 0.830719 seconds
Time of computation: 0.839334 seconds
Time of computation: 0.816654 seconds
Time of computation: 0.814974 seconds
Time of computation: 0.818318 seconds
Time of computation: 0.814377 seconds
Time of computation: 0.837740 seconds
Time of computation: 0.823461 seconds
```

(c) Running the program with 8 thread



```
mahbodafarin1admin@Rajivs-MacBook-Air code % for i in {1..10}; do ./1.4; done
287 seconds
Time of computation: 0.831553 seconds
Time of computation: 0.808963 seconds
Time of computation: 0.820304 seconds
Time of computation: 0.808315 seconds
Time of computation: 0.816973 seconds
Time of computation: 0.805335 seconds
Time of computation: 0.802818 seconds
Time of computation: 0.800776 seconds
Time of computation: 0.803025 seconds
```

(a) Running the program with 16 thread

Figure 14: Screenshot of running the program using 1, 4, 8, and 16 threads

Table 2 shows the results of the figure 14 in the form of table. As we can see in the table 2, the average execution time of running the program using 1 thread is 1.4296441 seconds. Running the program using 1 thread means that we are running it in serial. When we used 4 threads, the average execution time for 10 runs is 0.9299235 seconds; therefore, 4 Threads run the program faster than 1 thread. The average execution time for 8 threads for 10 runs was 0.8291256 seconds, and it means that 8 threads run the program faster than 4 threads. The average execution time of running the program using 16 thread is 0.8226349 seconds, and this number is roughly equal to the average running for 8 threads, and even for some runs, the results of the 16 threads is worse than the results of the 8 threads. The reason is that we used the maximum parallelism of the program using 8 threads and 16 threads dose not makes it better and even in some cases the context switching, communication between threads, memory bound, and I/O bound makes the execution time worse than when we have 8 threads.

Table 2: Results of the running the program using 1, 4, 8, and 16 threads

Run Number	**Execution Time (Seconds)**			
	1 thread	**4 threads**	**8 threads**	**16 threads**
1	1.378122	0.878283	0.885928	0.928287
2	1.360951	0.868169	0.809751	0.831553
3	1.599916	0.930071	0.830719	0.808963
4	1.601839	0.887898	0.839334	0.820304
5	1.497168	0.845950	0.816654	0.808315
6	1.576870	0.887874	0.814974	0.816973
7	1.414737	1.087307	0.818318	0.805335
8	1.281233	0.980170	0.814377	0.802818
9	1.301636	1.020559	0.837740	0.800776
10	1.283969	0.912954	0.823461	0.803025
Average	**1.4296441**	**0.9299235**	**0.8291256**	**0.8226349**

Figure 15 shows the graphical results of the table 2. Using 1 thread means that we run the program serially and we have the worse execution time. When we use 4 threads, the execution time is better compared to 1 thread. When we used 8 threads the execution time is slightly better compared to 4 threads, but using 16 threads does not make it better because we reached the maximum parallelism of the program using 8 threads.

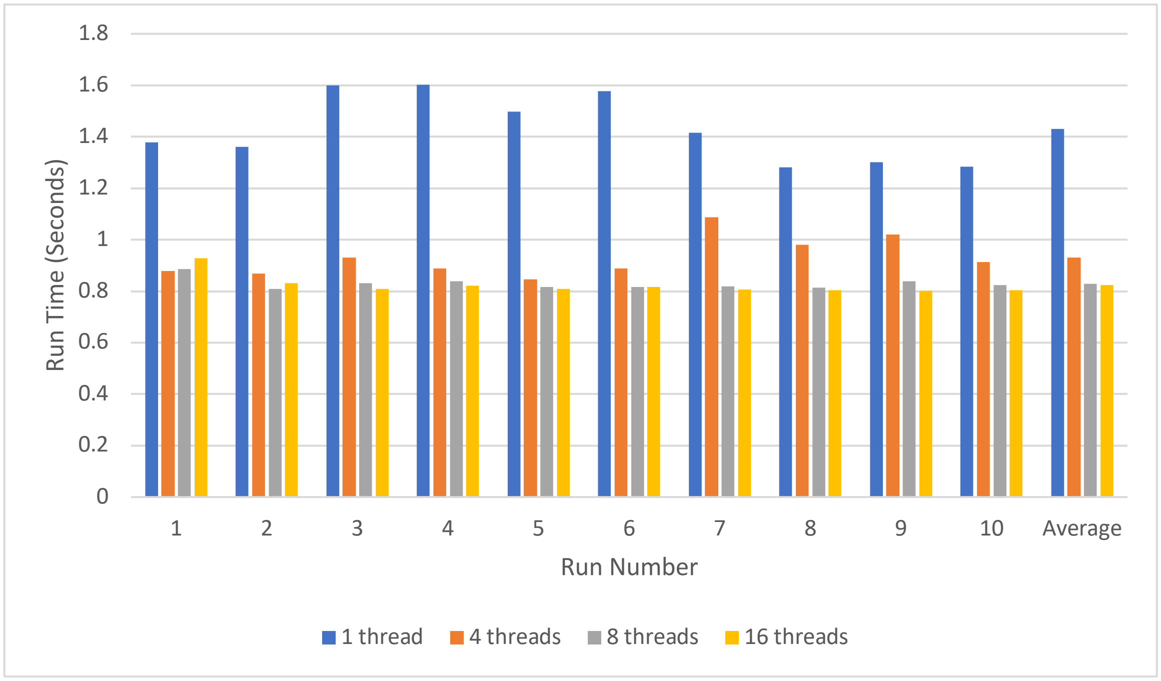


Figure 15: Graphical results for the running the program using 1, 4, 8, and 16 threads

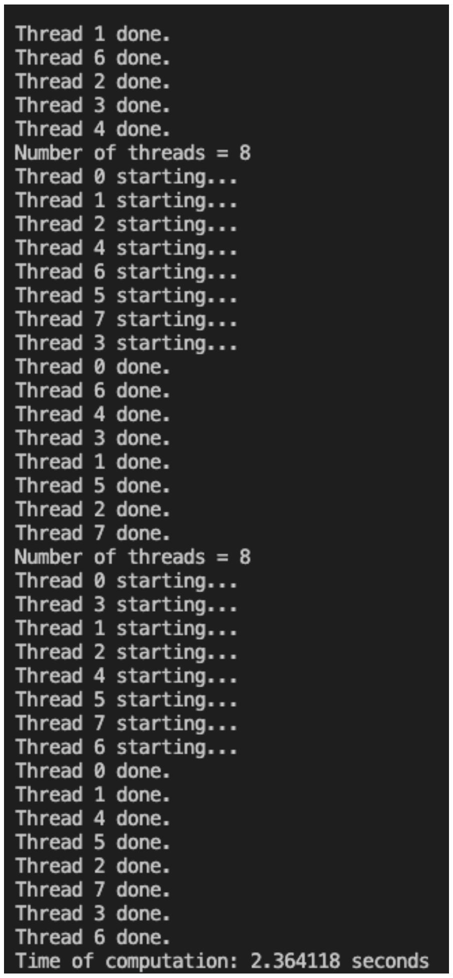
(b) For the middle matrix multiplication loop parallelized

1- Source program listing: for this part I should add the necessary *pragma* to parallelize the middle *for* loop. The below code shows the parallel version of the program.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 590
#define CHUNKSIZE 10

int main(int argc, char *argv[]) {
    omp_set_num_threads(8);//set number of threads here
    int i, j, k, nthreads, tid, chunk;
    double sum;
    double start, end; // used for timing
    double A[N][N], B[N][N], C[N][N];

    chunk = CHUNKSIZE;
```

```c
    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;
        }
    }

    start = omp_get_wtime(); //start time measurement
    for (i = 0; i < N; i++)
    {
        #pragma omp parallel shared(A,B,C,nthreads,i) private(j,k,tid,sum)
        {
            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 (j = 0; j < N; j++)
            {
                sum = 0;
                for (k=0; k < N; k++)
                {
                    sum += A[i][k]*B[k][j];
                }
                C[i][j] = sum;
            }
        //printf("Thread %d done.\n",tid);
        }
    }

    end = omp_get_wtime(); //end time measurement
    printf("Time of computation: %f seconds\n", end-start);
    return(0);
}
```

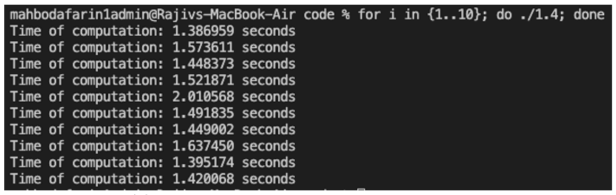
2- One screenshot from compiling and running the program: The below figure shows a screen shot from running the program for N = 590. As we can see in the picture, the runtime is 2.364118 seconds and it is bigger compare to the previous one. The reason is that we only parallelized the middle loop, and for every iteration of the outer loop, it will create 8 threads to calculate the outer loop in parallel. Therefore, we did not use the maximum parallelism.



```
Thread 1 done.
Thread 6 done.
Thread 2 done.
Thread 3 done.
Thread 4 done.
Number of threads = 8
Thread 0 starting...
Thread 1 starting...
Thread 2 starting...
Thread 4 starting...
Thread 6 starting...
Thread 5 starting...
Thread 7 starting...
Thread 3 starting...
Thread 0 done.
Thread 6 done.
Thread 4 done.
Thread 3 done.
Thread 1 done.
Thread 5 done.
Thread 2 done.
Thread 7 done.
Number of threads = 8
Thread 0 starting...
Thread 3 starting...
Thread 1 starting...
Thread 2 starting...
Thread 4 starting...
Thread 5 starting...
Thread 7 starting...
Thread 6 starting...
Thread 0 done.
Thread 1 done.
Thread 4 done.
Thread 5 done.
Thread 2 done.
Thread 7 done.
Thread 3 done.
Thread 6 done.
Time of computation: 2.364118 seconds
```

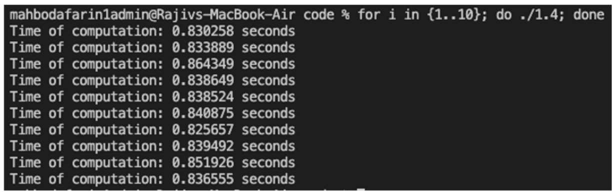
Figure 16: Screenshot from running the program

3- Graphical results of the average timings: Figure 17 shows the screenshot of the running the program for 1, 4, 8, and 16 threads.



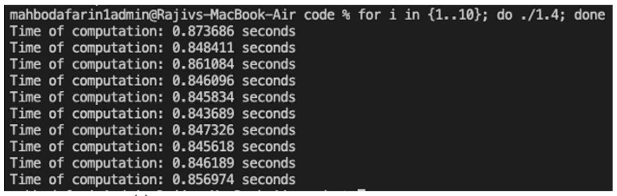
```
mahbodafarin1admin@Rajivs-MacBook-Air code % for i in {1..10}; do ./1.4; done
Time of computation: 1.386959 seconds
Time of computation: 1.573611 seconds
Time of computation: 1.448373 seconds
Time of computation: 1.521871 seconds
Time of computation: 2.010568 seconds
Time of computation: 1.491835 seconds
Time of computation: 1.449002 seconds
Time of computation: 1.637450 seconds
Time of computation: 1.395174 seconds
Time of computation: 1.420068 seconds
```

(a) Running the program with 1 thread



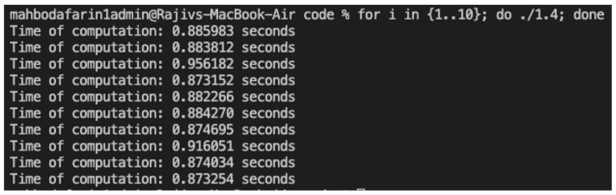
```
mahbodafarin1admin@Rajivs-MacBook-Air code % for i in {1..10}; do ./1.4; done
Time of computation: 0.830258 seconds
Time of computation: 0.833889 seconds
Time of computation: 0.864349 seconds
Time of computation: 0.838649 seconds
Time of computation: 0.838524 seconds
Time of computation: 0.840875 seconds
Time of computation: 0.825657 seconds
Time of computation: 0.839492 seconds
Time of computation: 0.851926 seconds
Time of computation: 0.836555 seconds
```

(b) Running the program with 4 thread



```
mahbodafarin1admin@Rajivs-MacBook-Air code % for i in {1..10}; do ./1.4; done
Time of computation: 0.873686 seconds
Time of computation: 0.848411 seconds
Time of computation: 0.861084 seconds
Time of computation: 0.846096 seconds
Time of computation: 0.845834 seconds
Time of computation: 0.843689 seconds
Time of computation: 0.847326 seconds
Time of computation: 0.845618 seconds
Time of computation: 0.846189 seconds
Time of computation: 0.856974 seconds
```

(c) Running the program with 8 thread



```
mahbodafarin1admin@Rajivs-MacBook-Air code % for i in {1..10}; do ./1.4; done
Time of computation: 0.885983 seconds
Time of computation: 0.883812 seconds
Time of computation: 0.956182 seconds
Time of computation: 0.873152 seconds
Time of computation: 0.882266 seconds
Time of computation: 0.884270 seconds
Time of computation: 0.874695 seconds
Time of computation: 0.916051 seconds
Time of computation: 0.874034 seconds
Time of computation: 0.873254 seconds
```

(a) Running the program with 16 thread

Figure 17: Screenshot of running the program using 1, 4, 8, and 16 threads

Table 3 shows the results of the figure 17 in the form of table. As we can see in the table, the average run time for 1 thread is roughly equal to previous one because we are running the program in serial in both cases. The average runtime for 4 threads is the best and the reason is that using 4 threads we reached the maximum parallelism for middle loop. When we increase the number of threads, the average runtime becomes worser because we reached the maximum parallelism using 4 threads and increasing the number of threads only increase the overhead and context switching between threads, make the execution time worser.

Table 3: Results of the running the program using 1, 4, 8, and 16 threads

Run Number	**Execution Time (Seconds)**			
	1 thread	**4 threads**	**8 threads**	**16 threads**
1	1.386959	0.830258	0.873686	0.885983
2	1.573611	0.833889	0.848411	0.883812
3	1.448373	0.864349	0.861084	0.956182
4	1.521871	0.838649	0.846096	0.873152
5	2.010568	0.838524	0.845834	0.882266
6	1.491835	0.840875	0.843689	0.884270
7	1.449002	0.825657	0.847326	0.874695
8	1.637450	0.839492	0.845618	0.916051
9	1.395174	0.851926	0.846189	0.874034
10	1.420068	0.836555	0.856974	0.873254
Average	**1.5334911**	**0.8400174**	**0.8514907**	**0.8903699**

Figure 18 shows the graphical results of the table 3.

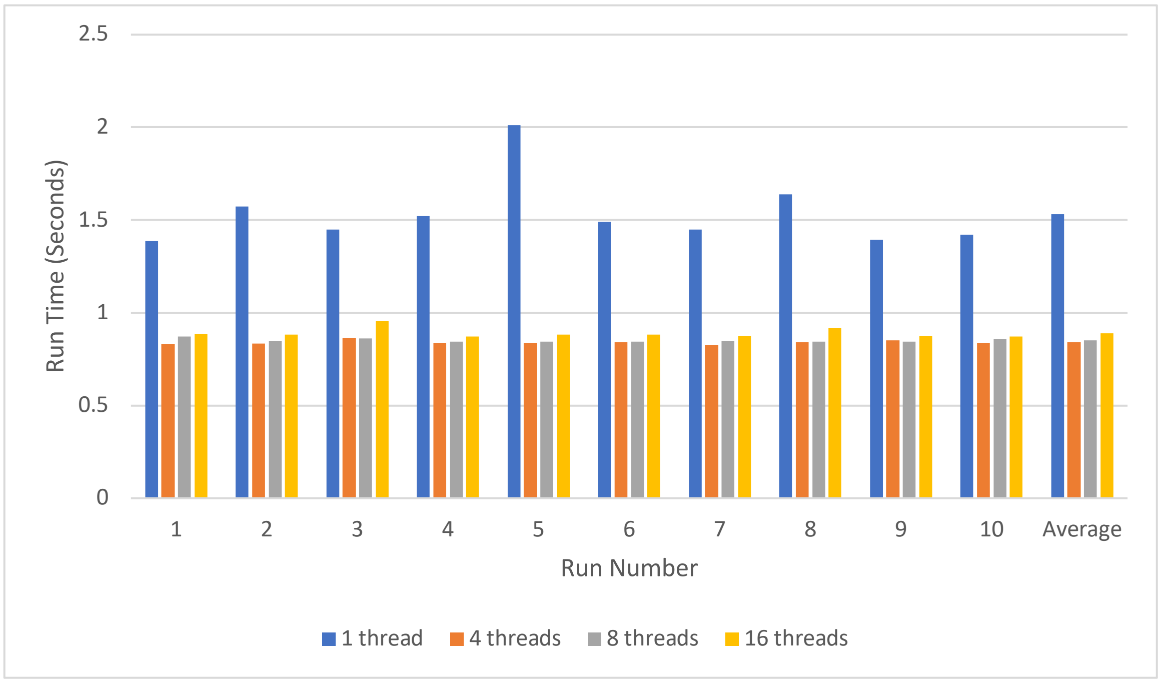


Figure 18: Graphical results for the running the program using 1, 4, 8, and 16 threads

(c) Your conclusions and explanations of the results

Here is a list of conclusions about the results:

- For the outer matrix multiplication loop, when we have one thread, the average run time is 1.4296441 seconds. It is like we run the program in serial because just using one thread means that we run the program in serial. When we use 4 threads, the average execution time is 0.9299235 seconds. Using 4 threads means that 4 threads is running the program in parallel and that is why we had a spectacular improvement in average execution time with 4 threads compared to 1 thread. The average execution time for 8 threads is 0.8291256 seconds. With 8 threads we had more parallelism compared to 4 threads and that is the reason for getting better average execution time for 8 threads compared to 4 threads. When we use 16 threads, the average execution time is 0.8226349 seconds and it is roughly the same as average execution time for 8 threads. Even in some cases the execution time for 16 threads is worse than the execution time for 8 threads. The reason is that when we used 8 threads, we reached the maximum parallelism and using 16 threads does not improve it significantly and even in some cases the execution time is worser due to the overhead of having more than needed threads and the context switching, context switch between threads, communication between threads, memory bound, and I/O bound make the execution time worser.

- For the middle matrix multiplication loop, the average execution time using 1 thread is 15334911 seconds and it is roughly equal to the average execution time with 1 thread for outer loop matrix multiplication. The reason for that is for both cases we are using only 1 thread for computation and using only 1 thread means that we are running the program in serial; therefore, both average timing results are roughly the same. The average execution time for 4 threads is 0.8400174 seconds and as we expected, we had a significant improvement compared to using 1 thread. For the middle loop parallelism, we reached the maximum parallelism using 4 threads, and using 8 and 16 threads dose not makes the run time better and even makes it worser. The average run time using 8 threads is 0.8514907 seconds and it is slightly worser compared to using 4 thread. The average run time for using 16 threads is 0.8903699 seconds and it is significantly worse than the average execution time for using 4 threads and the reason is that we already reached the maximum parallelism with 4 threads and using 8 or 16 threads won't improve it and even make it worse due to overhead and context switch between threads, communication between threads, memory bound, and I/O bound.
- When we parallelize the outer loop, we have a better parallelism compared to only parallelizing the middle loop. When we only parallelize the middle, for every iteration of the outer loop, it will create threads for running the middle loop in parallel.