Group-14

Md Tahseen Anam (19941202-T678)

Nafi Uz Zaman (19930919-T550)

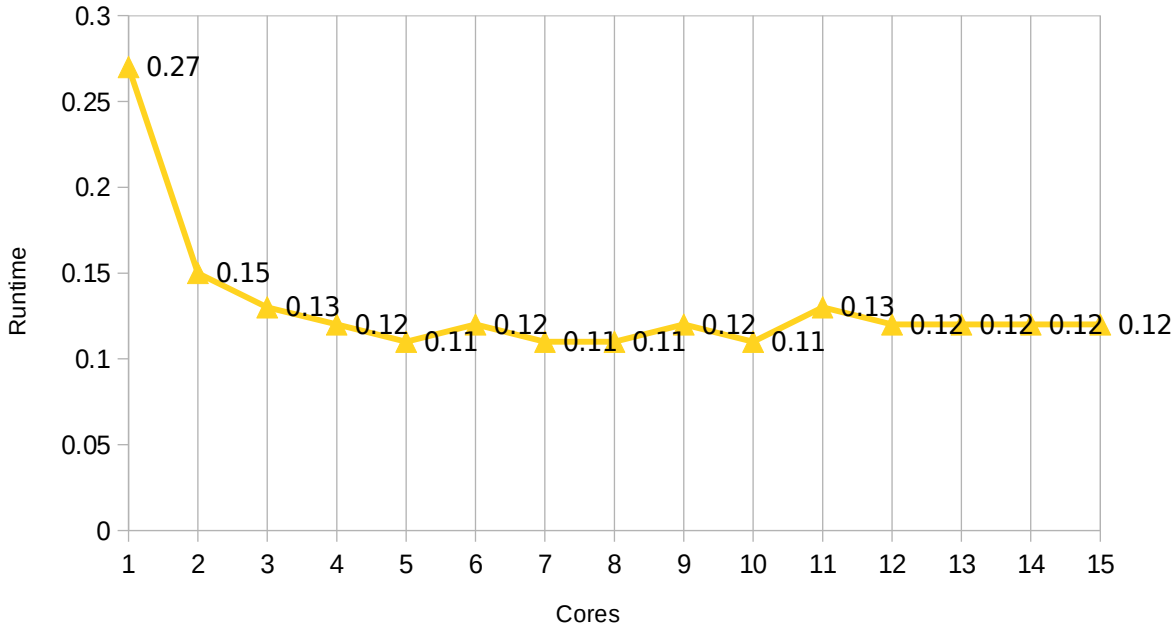
Exercise 1: Sieve of Eratosthenes

Using OpenMp we modified our previous version of this program. Calculating the number of threads was also done differently. The number of threads was equal to twice the number of cores. This did not cause any drastic changes to the program as shown by the speed up curve. Instead of using arrays extensively, we opted to use vectors to store the values. This allowed us to eliminate the process of manually allocating dynamic memory and made writing the code much easier and it is also more readable now. The speed up curve is shown below:

Reporting time to compute primes from 1 – 100,000 with increasing cores:

Cores	Runtime in seconds
1	.27
2	.15
3	.13
4	.12
5	.11
6	.12
7	.11
8	.11
9	.12
10	.11
11	.13
12	.12
13	.12
14	.12
15	.12

Speed up curve:



The time reached an almost constant value after a specific amount to cores. We believe this happened because the PC that this program was tested on only had 4 cores and a maximum number of 8 threads. Therefore, even after increasing the number of cores beyond “4” we got approximately the same values for runtime. This might happen because Open MP caps the number of threads to the number of logical threads that the hardware can provide. Or it could be that the number of threads was capped to the number of threads set by the environment variable by Open MP, which in this case was 8.

Overall, we have to say that this version of the program was easier to write. The part where we had to compute the *“seeds”* was kept the same however, the parallel part was easier to write. We did not have to create chunks manually therefore, we did not have to calculate the size of each array equal to a chunk size. We did not have to use mutexes and define an explicit critical section as the that was taken care of Open MP and its directives. The loop construct in Open MP made it easier to share the work load amongst threads without the need of explicitly assigning each thread to particular chunk. Posix threads required us to write more code and keep track of carefully defining shared and private variables and to set and destroy mutex locks.

Exercise 2: Conway's Game of Life

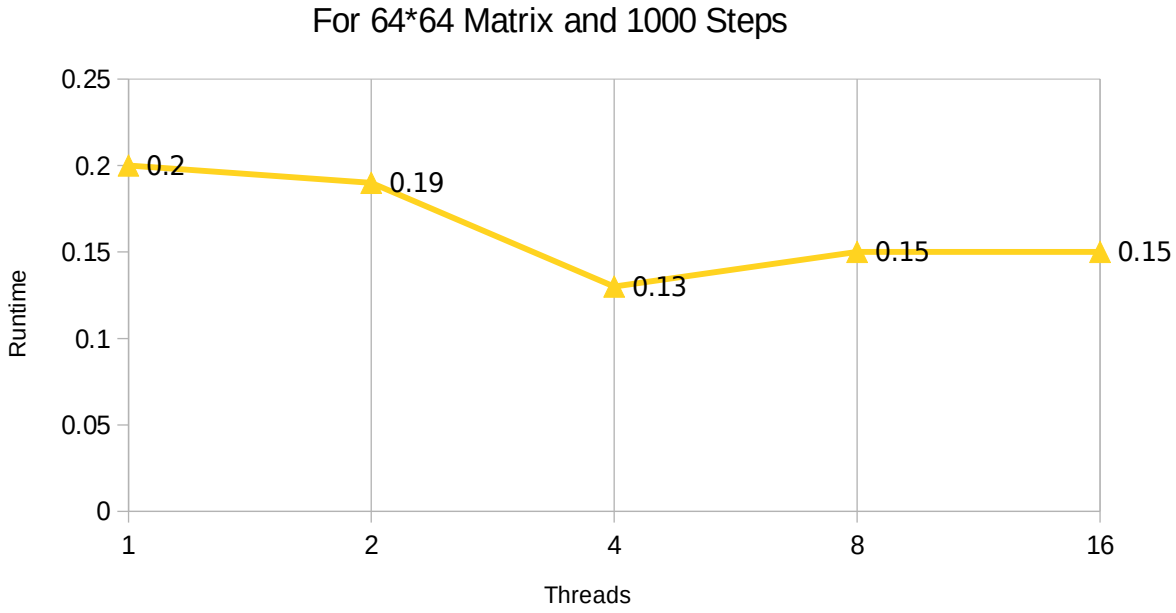
Sequential runtime performance:

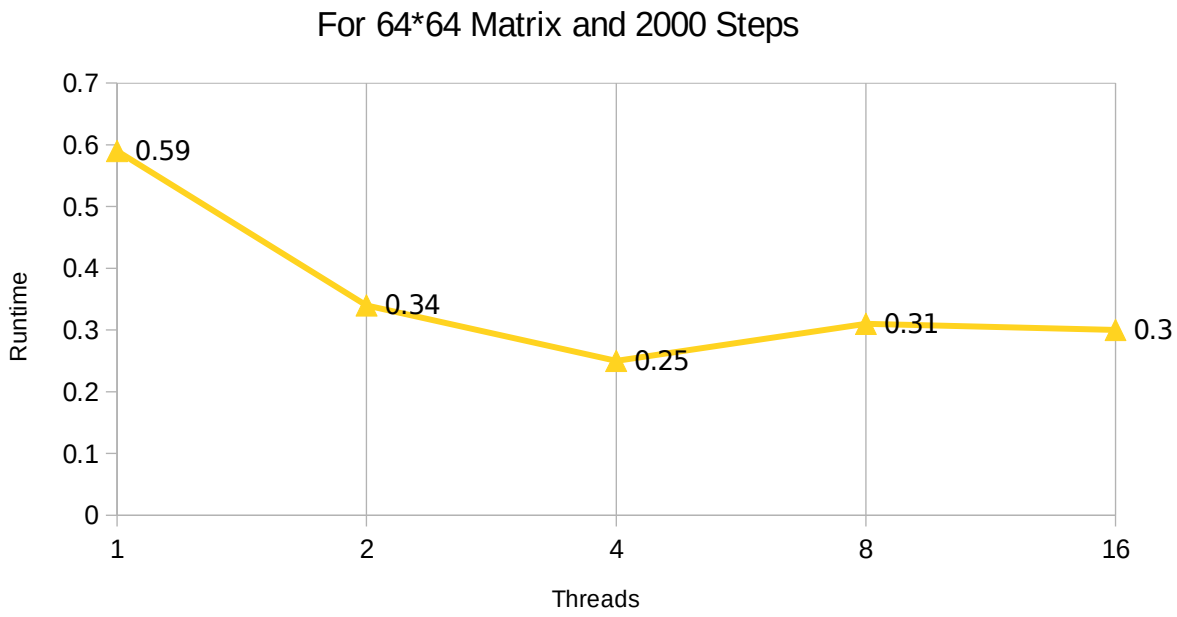
Matrix Size	Number of Steps	Runtime in seconds
64	1000	.052
64	2000	.11
1024	1000	13.76
1024	2000	27.4
4096	1000	220.8
4096	2000	445.5

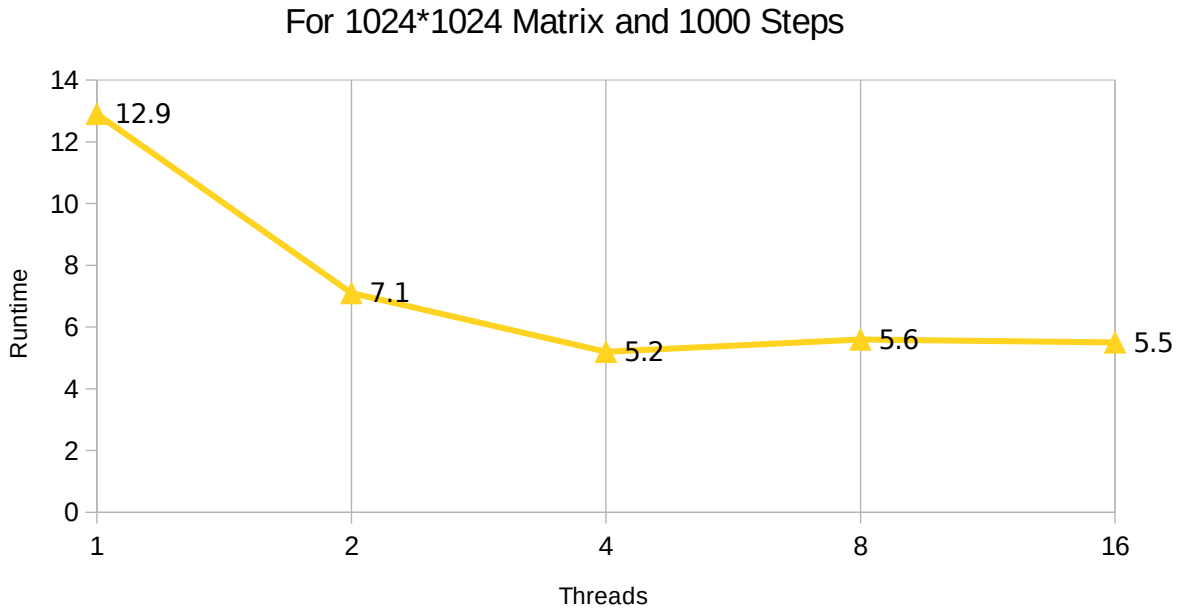
Parallel runtime performance using Open MP:

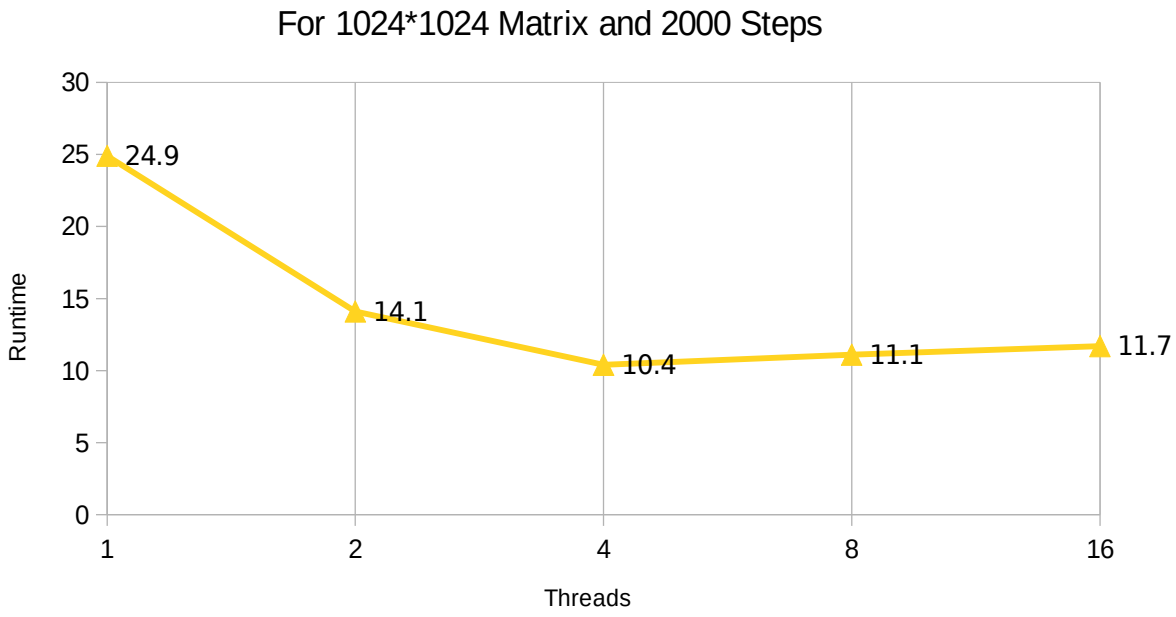
Matrix Size	Number of Steps	Number of Cores	Runtime in seconds
64	1000	1,2,4,8,16	0.20,0.19,0.13,0.15,0.15
64	2000	1,2,4,8,16	0.59,0.34,0.25,0.31,0.30
1024	1000	1,2,4,8,16	12.9,7.1,5.2,5.6,5.5
1024	2000	1,2,4,8,16	24.9,14.1,10.4,11.1,11.7
4096	1000	1,2,4,8,16	137.1,78.1,63.7,65.1,64.8
4096	2000	1,2,4,8,16	278.8,153.2,127,130.6,131.8

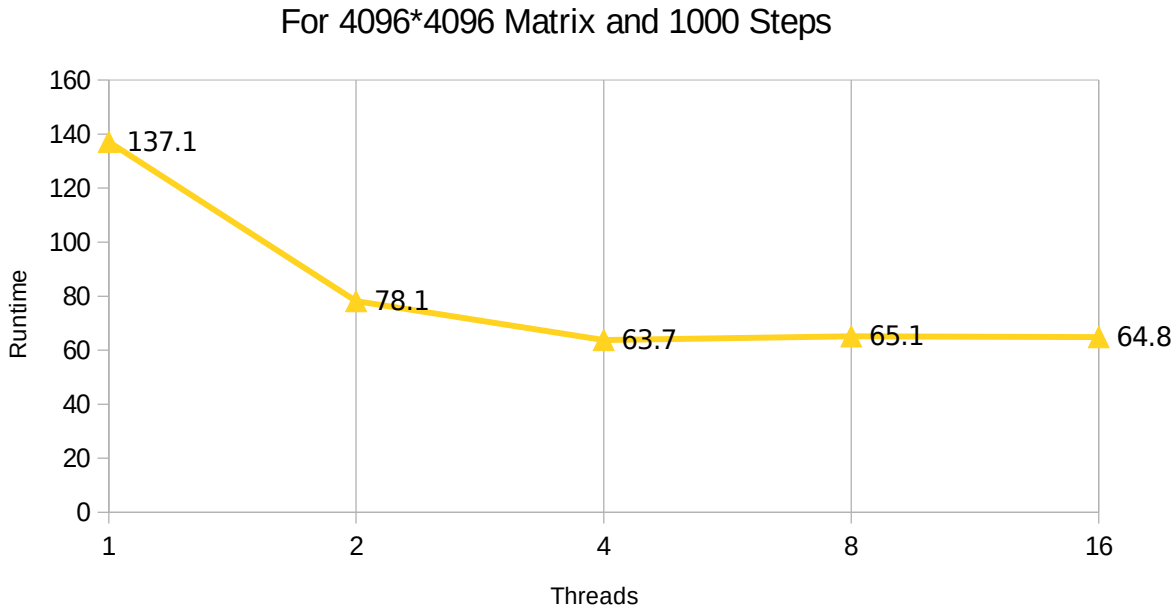
Speed up curve:

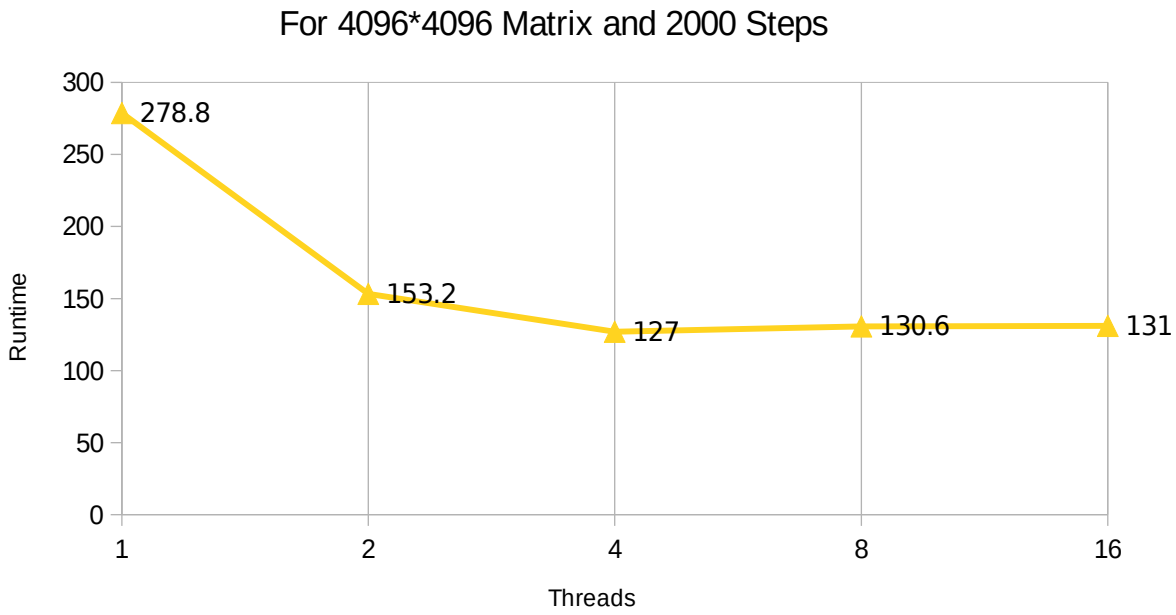



For 64*64 Matrix and 2000 Steps
Runtime
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0.59
0.34
0.25
0.31
0.3
1
2
4
8
16
Threads


For 1024*1024 Matrix and 1000 Steps
Runtime
14
12
10
8
6
4
2
0
12.9
7.1
5.2
5.6
5.5
1
2
4
8
16
Threads


For 1024*1024 Matrix and 2000 Steps
Runtime
30
25
20
15
10
5
0
24.9
14.1
10.4
11.1
11.7
1
2
4
8
16
Threads


For 4096*4096 Matrix and 1000 Steps
Runtime
160
140
120
100
80
60
40
20
0
137.1
78.1
63.7
65.1
64.8
1
2
4
8
16
Threads



The results that we got after parallelizing the program were improved than when the program was sequential. As seen by the runtimes, in the tables, increasing the number of cores decreased the runtimes significantly. We got varying numbers for the runtime after adjusting the number of steps and cores.

Exercise 3: Matrix Multiplication in OpenMP

After writing the code and experimenting with it, we found some results that were quite odd. When the outermost or the outer 2 loops were parallelized, some of the index values that we were getting were quite odd. This happened frequently when the just 1 or 2 of the corresponding loops were parallelized. The pictures below will give a better representation of our findings:

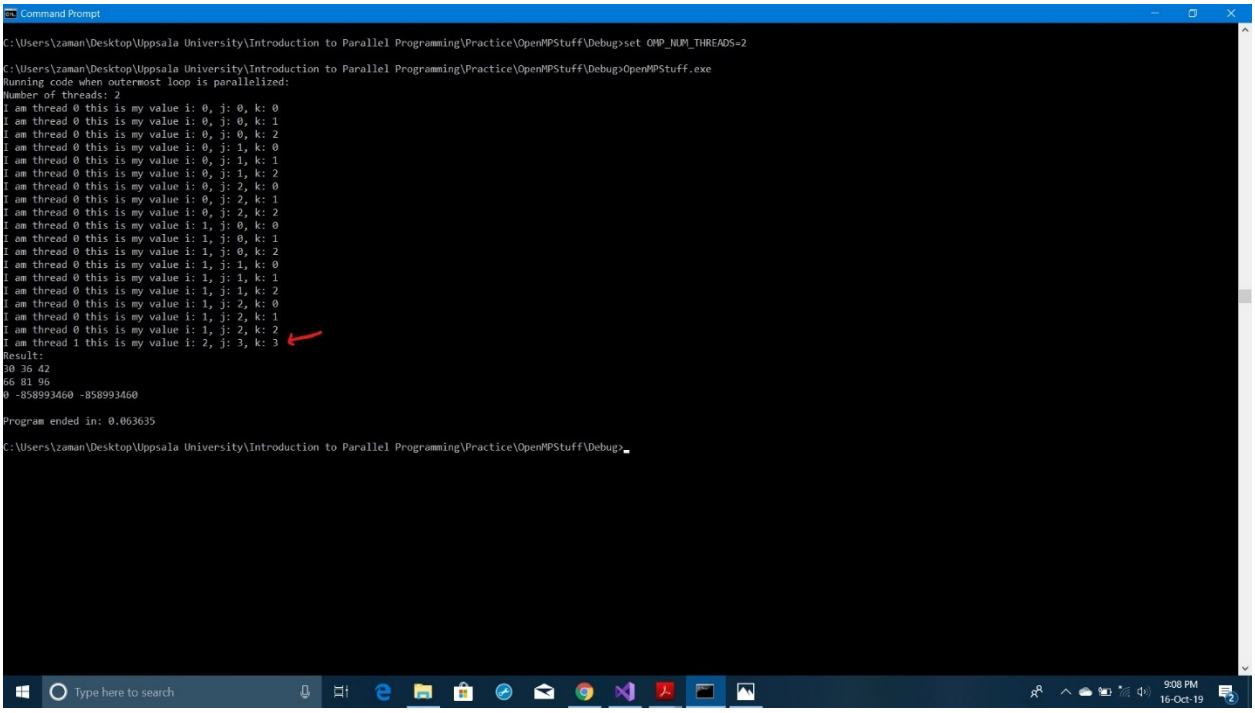


Fig 1: Result when outermost loop is parallelized

The matrices used for this experiment were 3X3 square matrices. As shown by the red arrow, we were getting a value of 3 when iterating through one of the loops; in this case it was the inner most loop. The maximum value that the iterating variables in each loop could reach was supposed to be 2. This caused an undefined behavior and thus caused the result to contain garbage values in the same indexes as that of the value of the loop variable, where the loop variable value was undefined. The same can be seen when the outer 2 loops were parallelized:

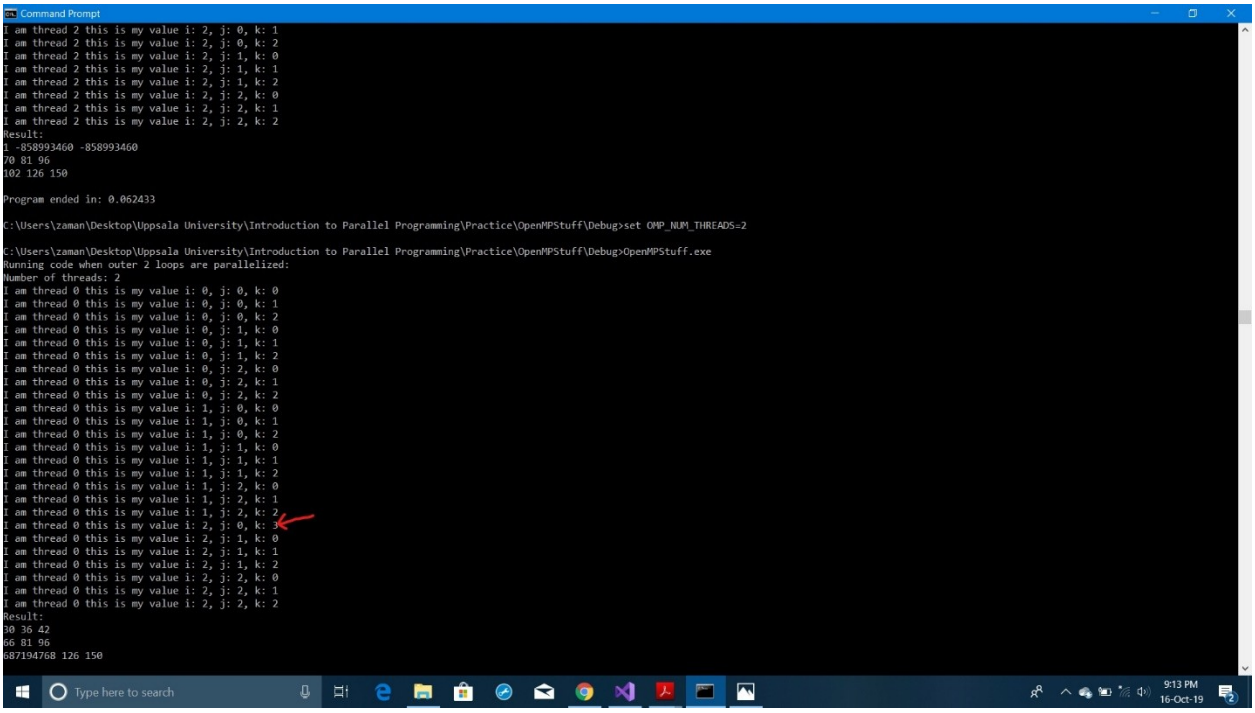


Fig 2: Result when outer 2 loops are parallelized

The program did however, run successfully and produced the correct results when all 3 loops were parallelized as shown below:

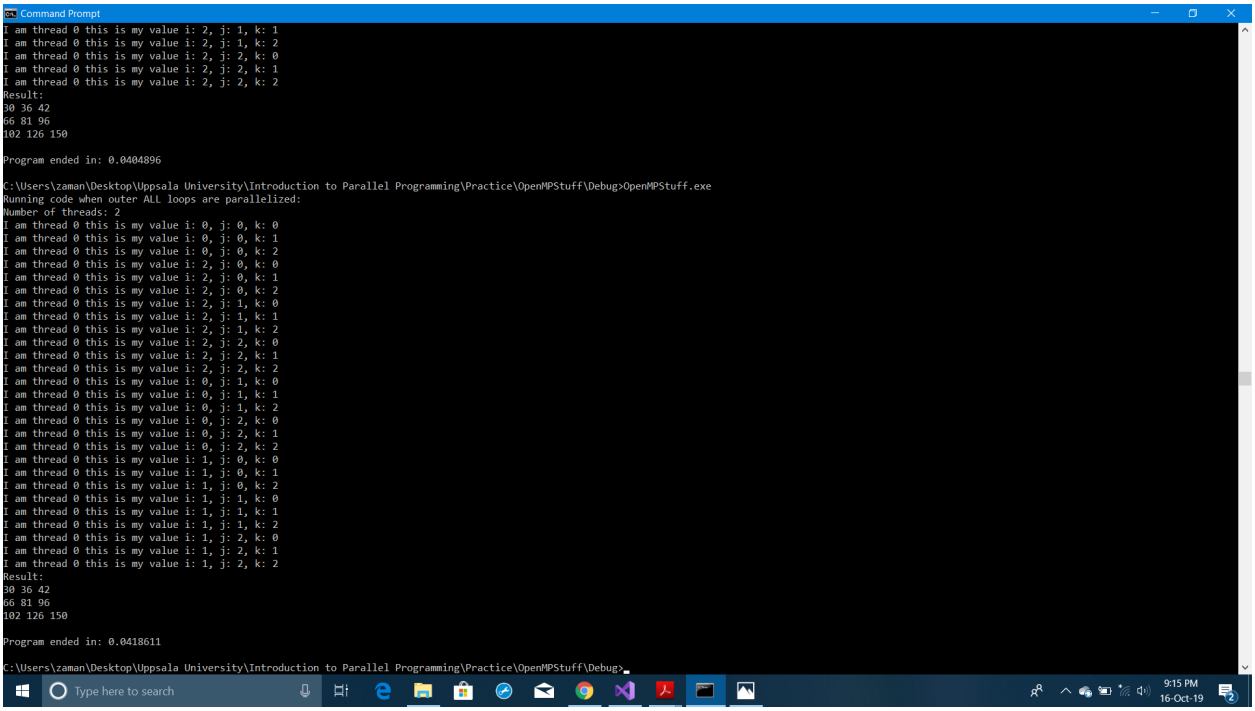


Fig 3: Result when ALL loops are parallelized

Exercise 4: Sparse Matrix Product in OpenMP

For this exercise, we have downloaded **AF23560** matrix file from matrix market and used that file as our sparse matrix. The file is included with the source code. The file contains a 23560 by 23560 sparse matrix in compress row format. Using that file, we have also created our vector of random numbers between 1 to 100. We then use OpenMP to calculate the product of our sparse matrix with the vector. The user needs to provide the values for the number of threads and matrix size. As our sparse matrix is 23560 by 23560, user can not provide a value greater than 23560 as matrix size. So, for example if the user provides a matrix size 64, we take the sub-matrix of size 64 by 64 from the 23560 by 23560 sparse matrix and calculate the product with the vector. We monitor the time taken to calculate the product and thus measure the performance. Below is a table showing the performance measured for different values of threads and matrix size. First we are giving the information of the machine that is used to measure the performance.

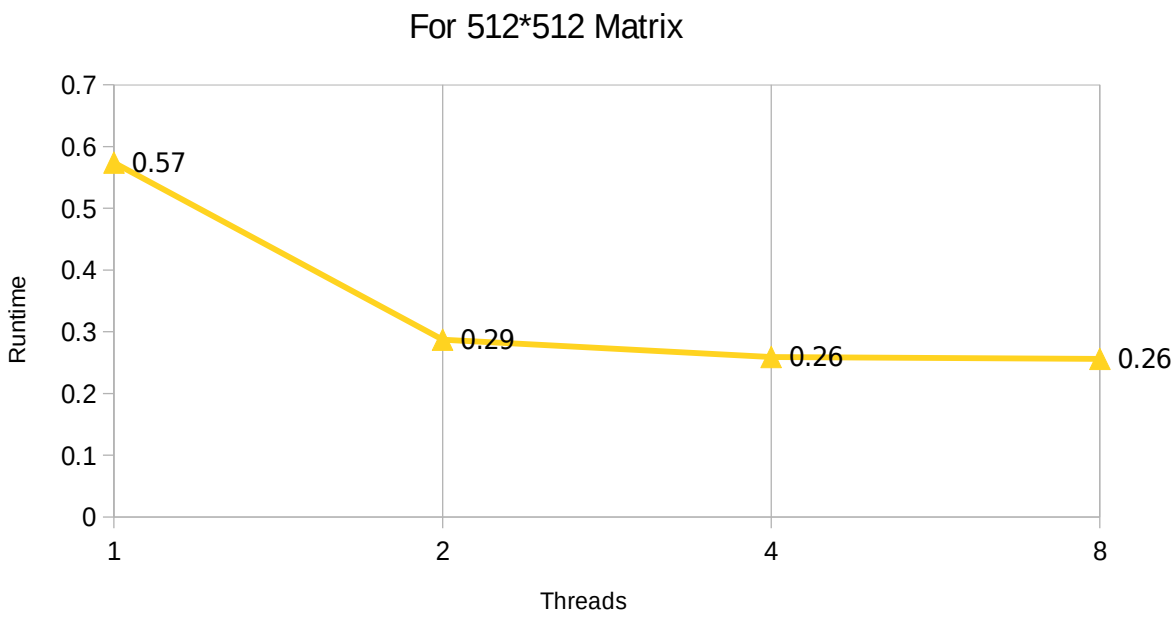
Socket	1
Cores per socket	2
Threads per core	2

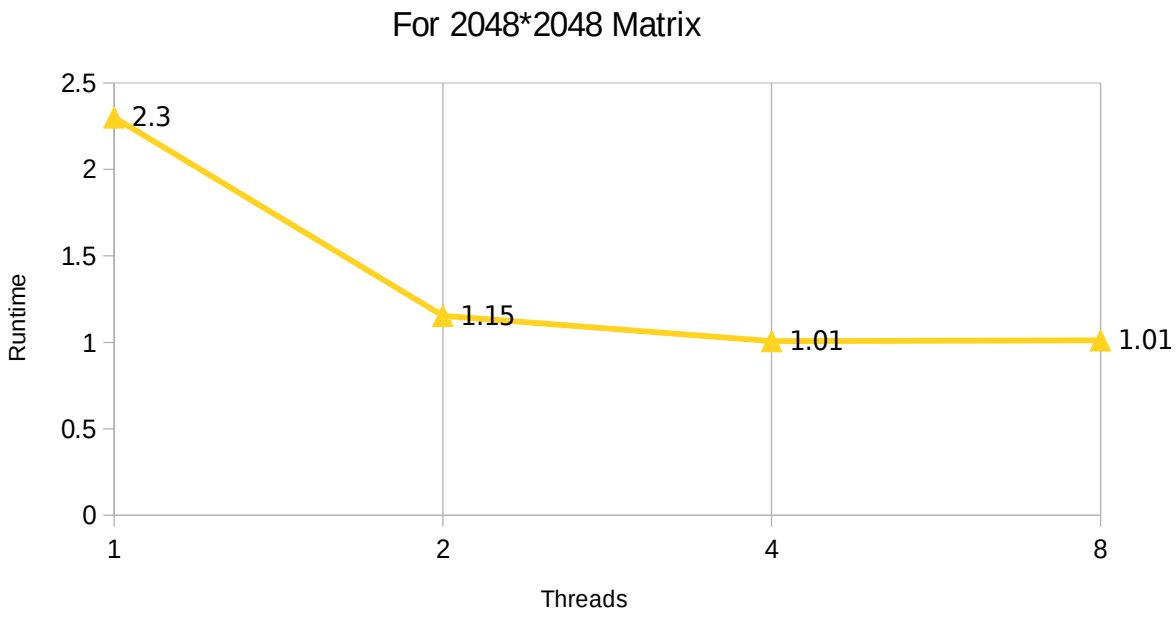
Number of Threads	Matrix Size (N*N)	Performance (Seconds)
1	512	0.574068
2	512	0.287140
4	512	0.259122
8	512	0.256265
1	2048	2.301408
2	2048	1.152544
4	2048	1.007016
8	2048	1.011043
1	4096	4.584483

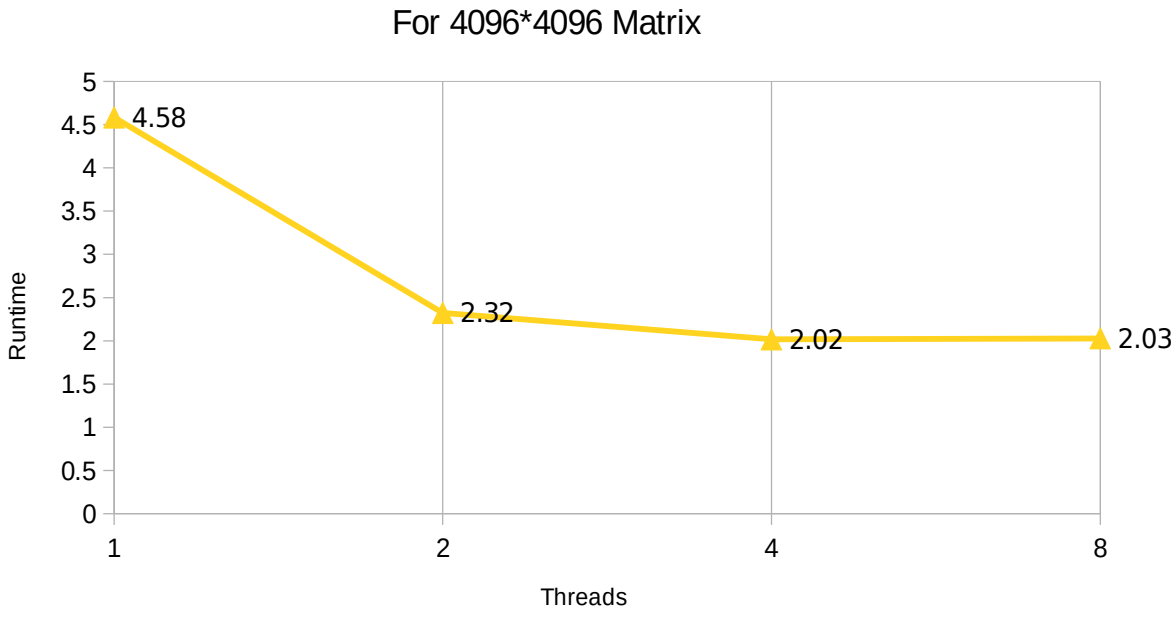
2	4096	2.321554
4	4096	2.015312
8	4096	2.027230
1	23560	26.386954
2	23560	13.322595
4	23560	11.496394
8	23560	11.684436

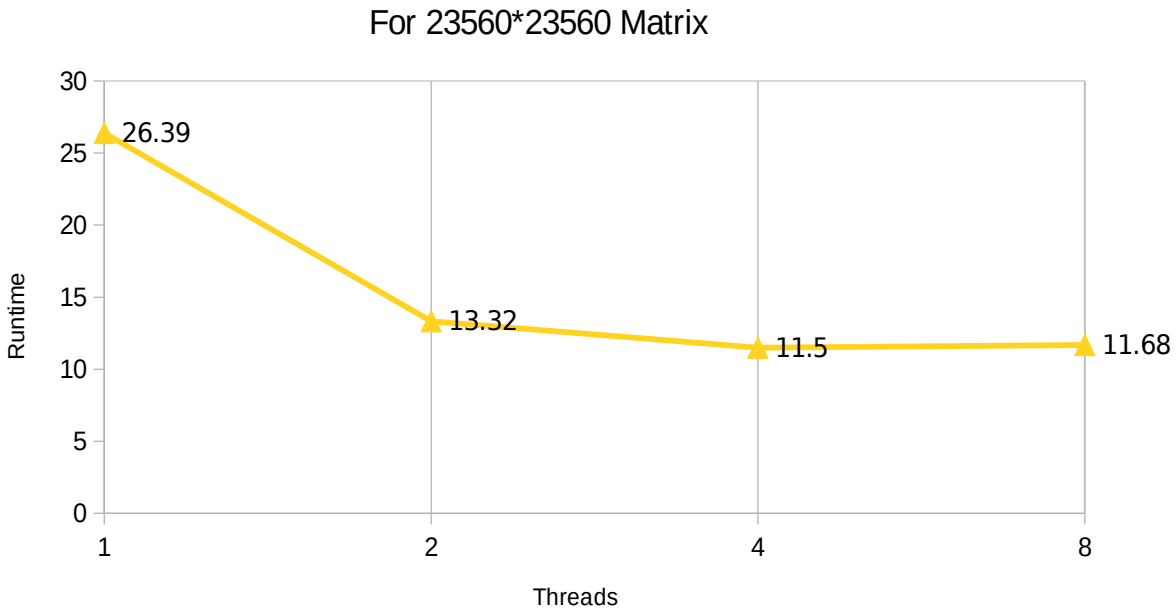
As we can see, the time gets decreased to almost half when we increase the thread number from 1 to 2. We can also observe that our performance gets better as we increase thread number upto 4 threads and after that, the performance does not increases.

Speed up curve:




For 2048*2048 Matrix
Runtime
Threads
2.5
2
1.5
1
0.5
0
1
2
4
8
2.3
1.15
1.01
1.01


For 4096*4096 Matrix
Runtime
Threads
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
1
2
4
8
4.58
2.32
2.02
2.03


For 23560*23560 Matrix
30
25
20
15
10
5
0
Runtime
26.39
13.32
11.5
11.68
1
2
4
8
Threads