# CA670 Concurrent Programming

| | |
|---|---|
| Name | Archana Kalapgar |
| Student Number | 19210184 |
| Programme | MCM (Cloud) |
| Module Code | CA670 |
| Assignment Title | Assignment Two – Efficient Large Matrix Multiplication |
| Submission date | 19-April-2020 |
| Module coordinator | David Sinclair |

I declare that this material, which I now submit for assessment, is entirely my work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. I have read and understood the Assignment Regulations set out in the module documentation. I have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged, and the source cited are identified in the assignment references. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

I have read and understood the referencing guidelines found recommended in the assignment guidelines.

Name: Archana Kalapgar
Date: 19-April-2020

# Assignment Two – Efficient Large Matrix Multiplication in OpenMP

**PROBLEM STATEMENT:** To develop an efficient large matrix multiplication algorithm in OpenMP.

**LARGE MATRIX MULTIPLICATION:** The goal of this assignment is to obtain the multiplication of a large two-dimension Matrix (2-D Matrix). There are several ways for computing the matrix multiplication but a blocked approach which is also called the partition approach seems to be a better solution than the traditional sequential approach. Here basically the idea is to break the complex arrays into something easier to deal with. The matrix is divided into multiple partitions, each thread performs the multiplication of the elements, reside in that simpler partition and finally sums up the results in a parallel fashion.

**OpenMP [2]:** Open Multi Programming is one of the significant methods of parallel programming. It is an application programming interface (API) which provides a flexible interface for developing multithreaded applications. It consists of compiler directives and libraries which are necessary for running parallel programs.

**DESIGN DESCRIPTION:** The basic idea behind the working of an OpenMP program is that any normal sequential program can be converted into OpenMP program by adding just one line of a statement: "#pragma omp parallel". The ' #pragma ' directive is the method specified by the C standard for providing additional information to the compiler. This is called a parallel construct, the following lines of code after the parallel construct will be executed by multiple threads. This is a very basic idea of how a simple OpenMP program works. In this assignment, similar openMP constructs are used to compute the matrix multiplication of all the elements of a large array. There are two methods in this assignment, Parallel matrix multiplication with pragma i.e openMP approach as well as parallel matrix multiplication without pragma i.e traditional approach to compare the efficiency.

## PROGRAM DESIGN:

1. **DIMENSION OF MATRIX:** This method specifies the dimension of the matrix multiplication:

```
// Specifies Dimension of Matrix
void DimensionOfMatrix(int dimension);
```

The dimensions of a matrix are the number of rows by the number of columns. The maximum dimension of the matrix is initialized to be 2000x2000. In the main method, for loop generates a two-dimension matrix of minimum size 200 which iterates by 200 till it reaches initialized maximum size of 2000. This dimension returns the size of the matrix.

2. **PRODUCE 2-D MATRIX WITH RANDOM NUMBERS AND GENERATE ZERO MATRIX TO STORE FINAL RESULT:** The two signature methods are specified as below:

```
/*Method to generate random square matrix for matrix computation
 assigning MIN_VAL 1 to MAX_VAL 10 */
TYPE** randomSquareMatrix(int dimension);
//Method to store the result of matrix multiplication
TYPE** zeroSquareMatrix(int dimension);
```

For loop in random square matrix function, produces matrixA*matrixB by inserting random numbers to the elements in the row and column. It generates this matrix by random numbers between initialised max value i.e 1 and min value i.e 10. To allocate memory dynamically at heap **malloc** function is used. Parallelized the for loops to create the matrices more efficiently. Zero square matrix method generates size zero matrixC to store the final result of the computation of matrixA*matrixB. It can be implemented in the same manner as a random square matrix by initializing with zero rather than a random value.

3. **MATRIX MULTIPLICATION METHODS:**

```
// Matrix multiplication methods ParallelWithoutPragma(TYPE** matrixA, TYPE** matrixB, TYPE** matrix
C, int dimension);
double ParallelWithPragma(TYPE** matrixA, TYPE** matrixB, TYPE** matrixC, int dimension);
```

The two methods of Parallel matrix multiplication, with and without pragma are specified as follows:

4. **PARALLEL MATRIX MULTIPLICATION WITH PRAGMA:**

In this method, matrixA, matrixB, matrixC, and size are passed which are generated by methods DimensionOfMatrix, RandomSquareMatrix and ZeroSquareMatrix. Tid function calculates the number of working threads and prints the value. This method uses OpenMP to compute the matrix multiplication. In openMP, there are a certain set of directives to optimise computation time which enhances the performance.

i. **The function that calculates the execution time of the respective matrix size in seconds:**

```
  struct timeval t0, t1;
  gettimeofday(&t0, 0)
  gettimeofday(&t1, 0);

  double elapsed = (t1.tv_sec-t0.tv_sec) * 1.0f + (t1.tv_usec - t0.tv_usec) / 1000000.0f;
```

gettimeofday is an inbuilt function which subtracts start time (t0) from the end time (t1). Here, it returns the latency time taken in seconds to complete the process.

ii. **Implicit Rules:** OpenMP has a set of rules, which deduce the data-sharing attributes of variables

### iii. Shared variables:

```
#pragma omp parallel shared(matrixA, matrixB, matrixC, size, chunk) private(i, j, k, jj, kk, tid, tmp)
```

The data-sharing attribute of variables declared outside the parallel region is called shared. Here, matrixA, matrixB, matrixC, size and chunk are shared between multiple threads.

### iv. Private variables:

All the variables declared in the private clause i.e i, j, k, jj, kk, tid, tmp are private. OpenMP replicates the private variables and assigns its local copy to each thread.

### v. Schedule variables:

```
#pragma omp for schedule (static, chunk)
```

vi.

The schedule (static, chunk) clause of the loop construct specifies that the for loop has the static scheduling type. OpenMP divides the iterations into chunks of size chunk-size and it distributes the chunks to the threads. When no chunk-size is specified, it distributes at most one chunk to each thread.

The formula for block-size:
BLOCK_SIZE <= sqrt(L1 cache) L1 cache is 256, The best suitable value for BLOCK_SIZE is 16. So, Initialised block-size as 16 and chunk size as 2 as it gives best computation results.

### vii. Blocked Multiplication:

Here, used **#pragma omp parallel** to parallelize the outermost for the loop. The statement delegates portions of the for loop for different threads. OpenMP divides iterations into initialised chunk-size 2 and block-size 16. Parallelize all the for loops by declaring pragma directives with 64 iterations to use 8 threads. Divided the workload assuming that each multiplication instruction would take the same amount of time. The workload can be divided equally among threads while dealing with dimensions of 200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800 and 2000. The multiplication result is stored in tmp, later tmp is assigned into the final resultant matrixC. The iteration continues until the result of the initialised maximum size of matrices is calculated.

```
#pragma omp for schedule (static, chunk)
for (jj = 0; jj < size; jj += block_size)
{

   for (kk = 0; kk < size; kk += block_size)
   {
      for (i = 0; i < size; i++)
      {
         for (j = jj; j < ((jj + block_size) > size ? size : (jj + block_size)); j++)
         {
            tmp = 0.0f;
```

```
            for (k = kk; k < ((kk + block_size) > size ? size : (kk + block_size)); k++)
            {
               tmp += matrixA[i][k] * matrixB[k][j];
            }
            matrixC[i][j] += tmp;
         }
      }
   }
}
```

### 5. PARALLEL MATRIX MULTIPLICATION WITHOUT PRAGMA:

This method uses a traditional blocked matrix multiplication approach to compute the matrix multiplication. In this method, the entire computation is the same as with the pragma method except it does not have the pragma directives for loop optimization. In this method, matrixA, matrixB, matrixC, and size are passed which are generated by function dimensionOfMatrix(), RandomSquareMatrix() and ZeroSquareMatrix(). The number of working threads is calculated by tid() function, which is 1 in this case. The latency time is calculated by gettimeofday() an inbuilt function which subtracts start time (t0) from the end time (t1). The final result is stored in matrixC.

## EFFICIENCY:

Image 1.1 depicts that, pragma method takes less time to compute the large matrix multiplication. Pragma method gives computation results 30% more efficiently. Matrix multiplication method with pragma utilises all cores due to which CPU utilisation is 100% as shown in Image 1.2.

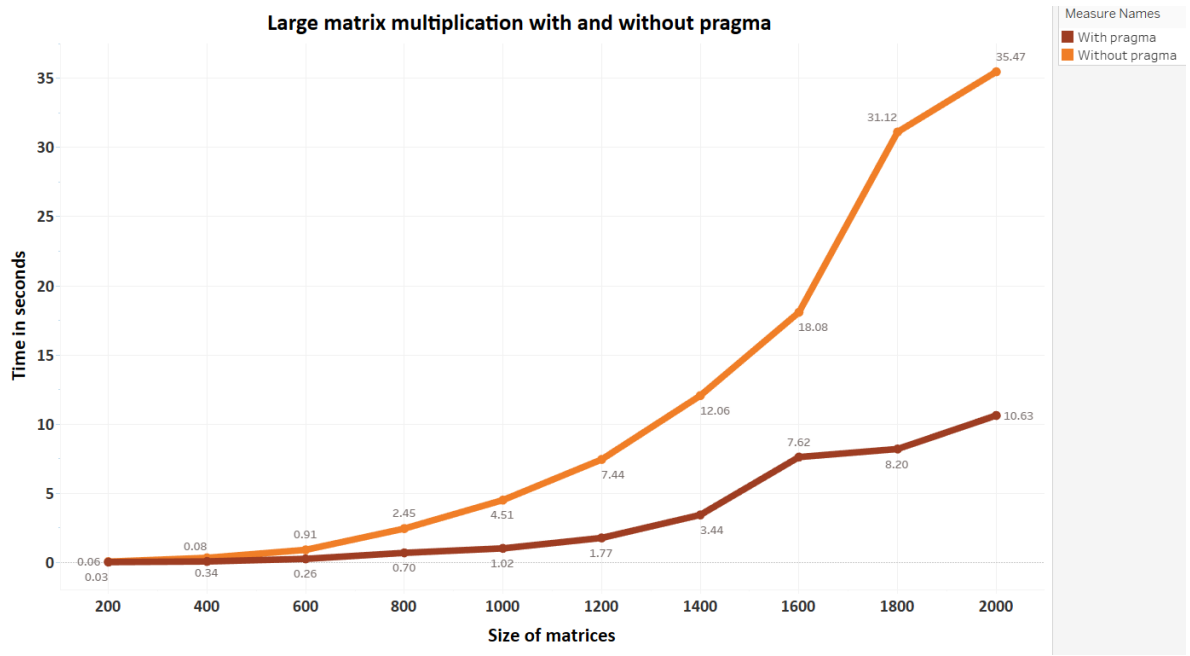


Image 1.1

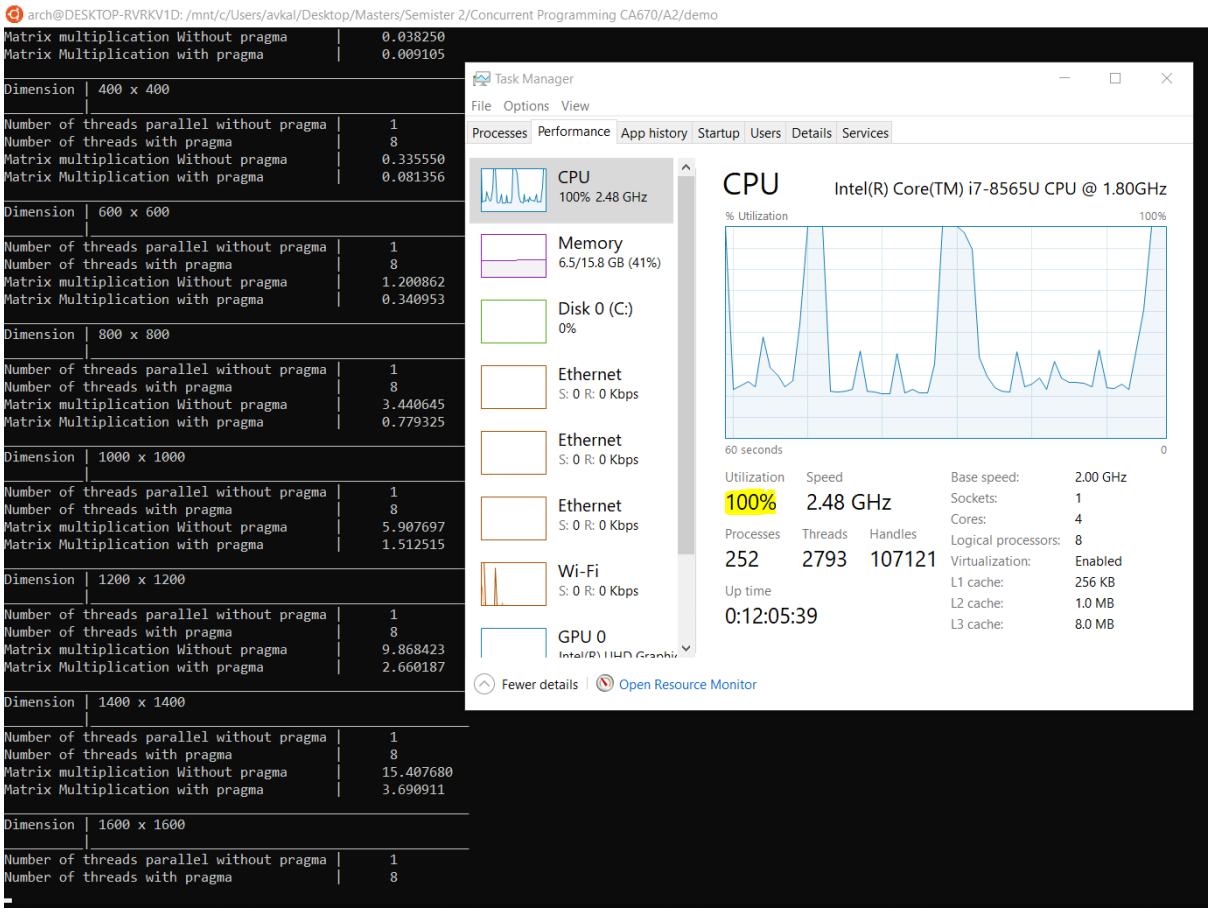


Image 1.2

## EVIDENCE:

To execute the program, run the following command as shown in Image 1.3

nt Programming CA670/A2/demo

```
/Concurrent Programming CA670/A2/demo$ gcc -g -Wall -fopenmp -o LargeMatrixMultiplication LargeMatrixMultiplication.c -lm
/Concurrent Programming CA670/A2/demo$ ./LargeMatrixMultiplication
```

Image 1.3

Table 1.4 shows the result of the program. Image 1.5 and 1.6 are proof of the result obtained after execution of the program.

| Size of matrix | Results without pragma (seconds) | Results with pragma (seconds) |
|---|---|---|
| 200x200 | 0.059 | 0.0284 |
| 400x400 | 0.335 | 0.075 |
| 600x600 | 0.9133 | 0.2613 |
| 800x800 | 2.4489 | 0.6961 |
| 1000x1000 | 4.5114 | 1.0199 |
| 1200x1200 | 7.4445 | 1.7747 |
| 1400x1400 | 12.0603 | 3.4367 |
| 1600x1600 | 18.0846 | 7.6199 |
| 1800x1800 | 31.1169 | 8.2029 |
| 2000x2000 | 35.4719 | 10.6268 |

Table 1.4

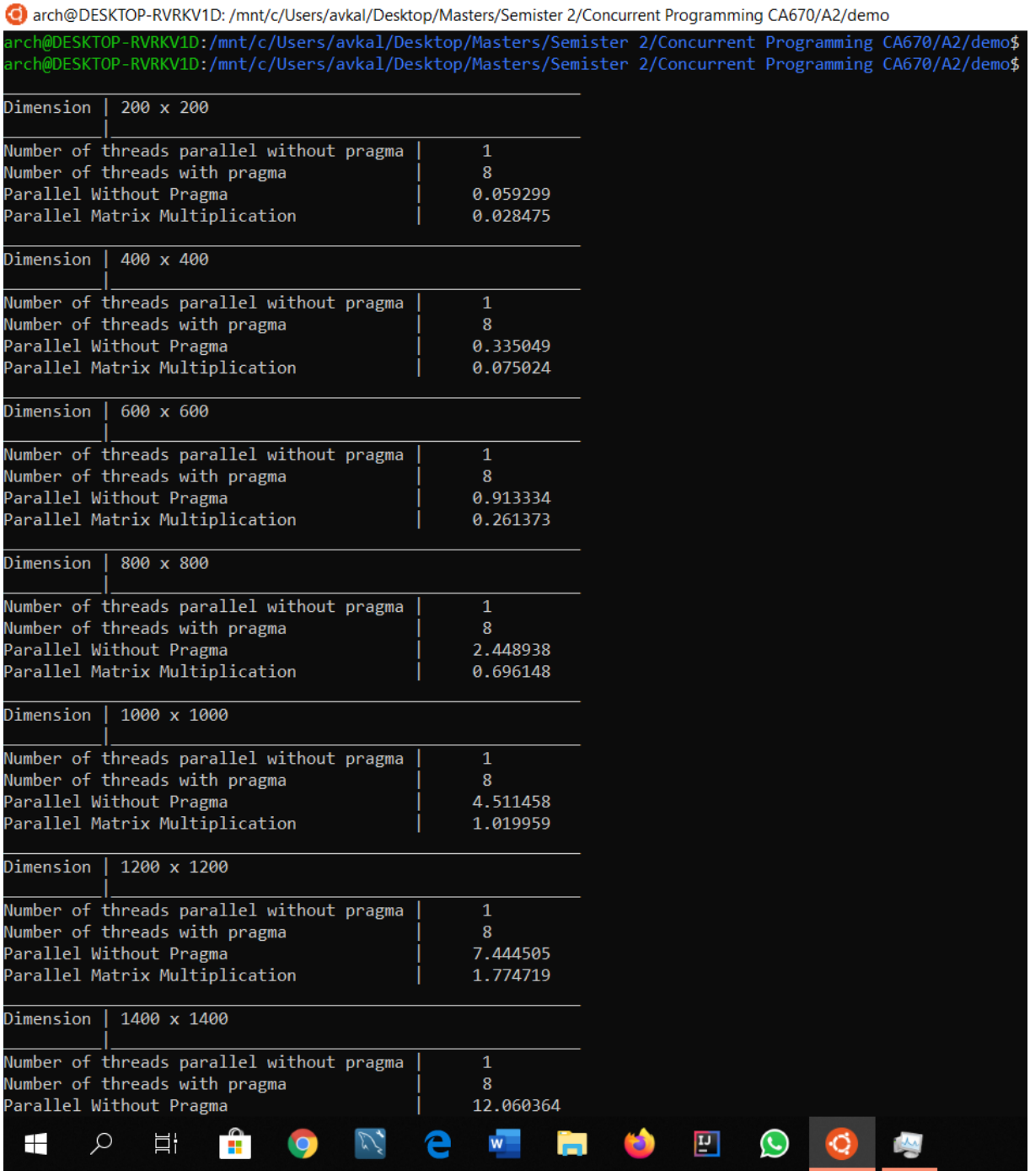
arch@DESKTOP-RVRKV1D: /mnt/c/Users/avkal/Desktop/Masters/Semister 2/Concurrent Programming CA670/A2/demo
arch@DESKTOP-RVRKV1D:/mnt/c/Users/avkal/Desktop/Masters/Semister 2/Concurrent Programming CA670/A2/demo$
arch@DESKTOP-RVRKV1D:/mnt/c/Users/avkal/Desktop/Masters/Semister 2/Concurrent Programming CA670/A2/demo$
Dimension | 200 x 200
Number of threads parallel without pragma | 1
Number of threads with pragma | 8
Parallel Without Pragma | 0.059299
Parallel Matrix Multiplication | 0.028475
Dimension | 400 x 400
Number of threads parallel without pragma | 1
Number of threads with pragma | 8
Parallel Without Pragma | 0.335049
Parallel Matrix Multiplication | 0.075024
Dimension | 600 x 600
Number of threads parallel without pragma | 1
Number of threads with pragma | 8
Parallel Without Pragma | 0.913334
Parallel Matrix Multiplication | 0.261373
Dimension | 800 x 800
Number of threads parallel without pragma | 1
Number of threads with pragma | 8
Parallel Without Pragma | 2.448938
Parallel Matrix Multiplication | 0.696148
Dimension | 1000 x 1000
Number of threads parallel without pragma | 1
Number of threads with pragma | 8
Parallel Without Pragma | 4.511458
Parallel Matrix Multiplication | 1.019959
Dimension | 1200 x 1200
Number of threads parallel without pragma | 1
Number of threads with pragma | 8
Parallel Without Pragma | 7.444505
Parallel Matrix Multiplication | 1.774719
Dimension | 1400 x 1400
Number of threads parallel without pragma | 1
Number of threads with pragma | 8
Parallel Without Pragma | 12.060364

Image 1.5

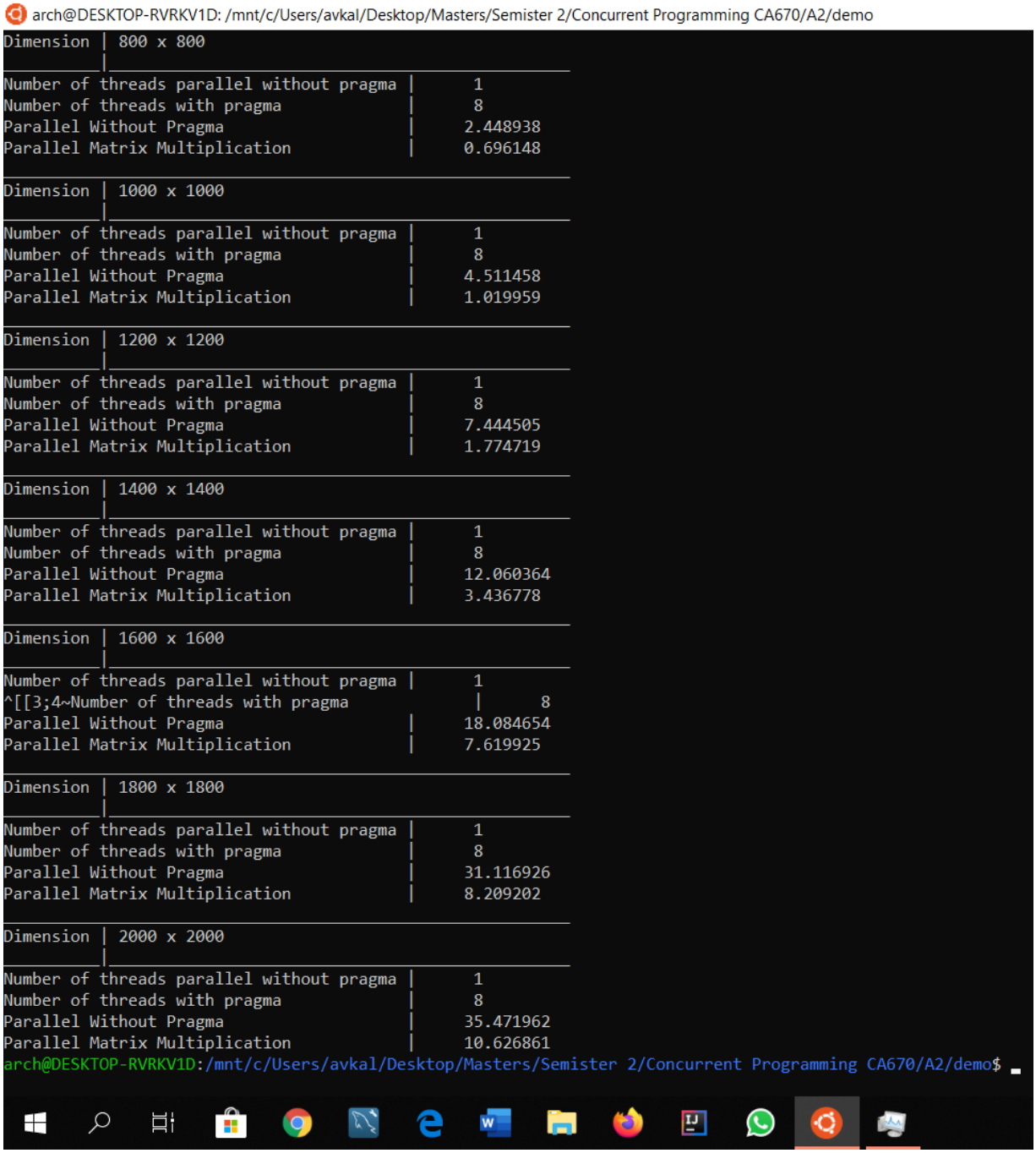
arch@DESKTOP-RVRKV1D: /mnt/c/Users/avkal/Desktop/Masters/Semister 2/Concurrent Programming CA670/A2/demo
Dimension | 800 x 800
Number of threads parallel without pragma | 1
Number of threads with pragma | 8
Parallel Without Pragma | 2.448938
Parallel Matrix Multiplication | 0.696148
Dimension | 1000 x 1000
Number of threads parallel without pragma | 1
Number of threads with pragma | 8
Parallel Without Pragma | 4.511458
Parallel Matrix Multiplication | 1.019959
Dimension | 1200 x 1200
Number of threads parallel without pragma | 1
Number of threads with pragma | 8
Parallel Without Pragma | 7.444505
Parallel Matrix Multiplication | 1.774719
Dimension | 1400 x 1400
Number of threads parallel without pragma | 1
Number of threads with pragma | 8
Parallel Without Pragma | 12.060364
Parallel Matrix Multiplication | 3.436778
Dimension | 1600 x 1600
Number of threads parallel without pragma | 1
^[[3;4~Number of threads with pragma | 8
Parallel Without Pragma | 18.084654
Parallel Matrix Multiplication | 7.619925
Dimension | 1800 x 1800
Number of threads parallel without pragma | 1
Number of threads with pragma | 8
Parallel Without Pragma | 31.116926
Parallel Matrix Multiplication | 8.209202
Dimension | 2000 x 2000
Number of threads parallel without pragma | 1
Number of threads with pragma | 8
Parallel Without Pragma | 35.471962
Parallel Matrix Multiplication | 10.626861
arch@DESKTOP-RVRKV1D:/mnt/c/Users/avkal/Desktop/Masters/Semister 2/Concurrent Programming CA670/A2/demo$

Image 1.6

## RESOURCES

https://github.com/iarchanaa/Large_Matrix_Multiplication_Using_openMP

## REFERENCES


1. “github”, [Online]. Available: https://github.com/dmitrydonchenko/Block-Matrix-Multiplication-OpenMP/blob/master/block_matrix/Source.cpp. [Accessed 19 04 2020]
2. “openmp”, [Online]. Available: https://www.openmp.org/. [Accessed 19 04 2020]
3. “stackoverflow”, [Online]. Available: https://stackoverflow.com/questions/50547661/gettimeofday-microsecond-not-limited-to-below-second. [Accessed 19 04 2020]
4. “medium”, [Online]. Available: https://medium.com/tech-vision/parallel-matrix-multiplication-c-parallel-processing-5e3aadb36f27. [Accessed 19 04 2020]
5. “jakascorner”, [Online]. Available: http://jakascorner.com/blog/2016/06/omp-for-scheduling.html. [Accessed 19 04 2020]