

CA-670 Concurrent Programming

| | |
|--------------------|---|
| Name | Nikhil Mittal |
| Student no. | 19210509 |
| Programme | MSc. Data Analytics |
| Module Code | CA670 |
| Assignment title | Efficient Large Matrix Multiplication in OpenMP |
| Submission Date | 20th April, 2020 |
| Module Coordinator | David Sinclair |

I declare that this material, which I now submit for assessment, is entirely my own 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: Nikhil Mittal

Date: 20th April 2020

Assignment 2

Efficient Large Matrix Multiplication in OpenMP

Introduction:

OpenMP is a library that supports shared memory multiprocessing. The OpenMP programming model is SMP (symmetric multi-processors, or shared-memory processors): that means when programming with OpenMP all threads share memory and data.

OpenMP has directives that allow the programmer to:

- specify the parallel region (create threads)
- specify how to parallelize loops
- specify the scope of the variables in the parallel section (private and shared)
- specify if the threads are to be synchronized
- specify how the works are divided between threads (scheduling)[1]

OpenMP consists of a set of compiler #pragmas that control how the program works. The pragmas are designed so that even if the compiler does not support them, the program will still yield correct behavior, but without any parallelism.[2]

The Design of the Program:

```
void Serial_Mat_Multi()
{
    T_Starting = omp_get_wtime(); //returns time gone in seconds
    for (int i = 0; i<Aa; i++)
    {
        for (int j = 0; j<Bb; j++)
        {
            double temp = 0;
            for (int k = 0; k<Ab; k++)
            {
                temp += a[i][k] * b[k][j];
            }
        }
    }
    T_Stopping = omp_get_wtime();
    cout << "Matrix Multiplication Time for Serial: " << T_Stopping - T_Starting << " seconds" << endl;
}

void Parallel_Mat_Multi()
{
    int i, j, k;
    T_Starting = omp_get_wtime();

    omp_set_num_threads(NUMBER_OF_THREAD); //number of threads running parallelly
#pragma omp parallel for schedule(dynamic,50) private(i,j,k) shared(a,b,c) //taking a group of threads
    for (i = 0; i<Aa; i++)
    {
        for (j = 0; j<Bb; j++)
        {
            double temp = 0;
            for (k = 0; k<Ab; k++)
            {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }

    T_Stopping = omp_get_wtime();

    cout << "Matrix Multiplication Time for Parallel: " << T_Stopping - T_Starting << " seconds." << endl;
}
```

The program was compiled and run on Microsoft Visual Studio 13.

The programs consist of 2 blocks Serial_Mat_Multi() & Parallel_Mat_Multi() and in which serial function is being used to multiply normally matrix multiplication and get the result using the `omp_get_wtime()` while the parallel block is being multiplied using `pragma` and `threads` so that matrix is multiplied parallelly and get the efficiency of the matrix using `omp_get_wtime`.

The program has three function blocks: -

Matrix_Get is used to get the matrix size from the user.

Serial_Mat_Multi() is used to multiply the matrix normally serially with one thread and get the matrix multiplication time in seconds.

Parallel_Mat_Multi() is used to multiply the matrix parallelly with the different number of threads and the matrix multiplication time in seconds.

The program has two blocks. In the serial block, only one thread is performing the task. For measuring the time taken to perform this task, the timer used is provided by `<omp.h>`. So, at the start of the function to sum up the values of the array, started the timer i.e.

```
T_Starting = omp_get_wtime();
```

Once the function calculates the sum of the large array, the time is noted again i.e.

```
T_Stopping = omp_get_wtime()
```

Now the actual time taken to perform the task would be printed

```
cout << "Matrix Multiplication Time for Serial: " << T_Stopping - T_Starting << "seconds" << endl;
```

The second block performs the same task but with multiple threads. In the second block

```
#pragma omp parallel for schedule(dynamic,50) private(i,j,k) shared(a,b,c)
```

is used to tell the compiler that the parallel block starts from here. The above statement will create a private copy of the variable for each thread and will divide the loop in each thread equally. Each thread will get a part of the array and they will calculate the sum of that part

and will store the result in their private variable (i,j,k). Once all the threads have finished

their task, the master thread will sum up all the values given by the threads and the result will

be stored in i,j,k variable. The header `<omp.h>` must be included to have multiple threads.

The number of threads can be changed by calling the `OMP_NUM_THREADS` variable

inside the parallel block function by

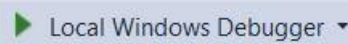
```
omp_set_num_threads(NUMBER_OF_THREAD)
```

It can be set manually outside the code by running the following command.

```
#define NUMBER_OF_THREAD 4
```

In the above example, the number of threads is set to 4.

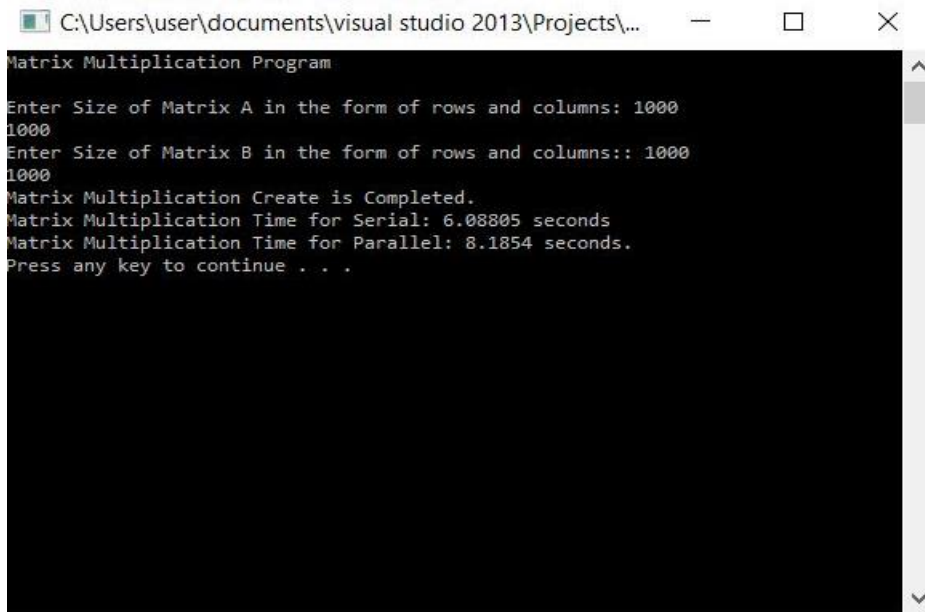
The program is compiled and executed by pressing this



Efficiency Proof of the program

The program was run with a different number of threads. Below are the outputs for the different number of threads.

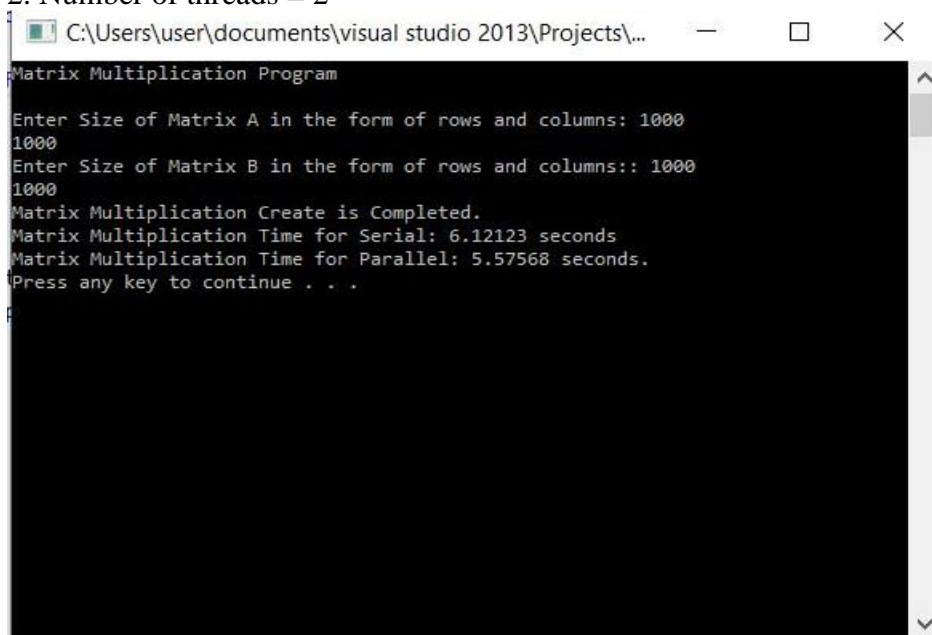
1. Number of threads =1



```
C:\Users\user\documents\visual studio 2013\Projects\...
Matrix Multiplication Program
Enter Size of Matrix A in the form of rows and columns: 1000
1000
Enter Size of Matrix B in the form of rows and columns:: 1000
1000
Matrix Multiplication Create is Completed.
Matrix Multiplication Time for Serial: 6.08805 seconds
Matrix Multiplication Time for Parallel: 8.1854 seconds.
Press any key to continue . . .
```

It can be seen, when the number of threads= 1, the parallel block and serial block are not the same as the parallel block has only one thread which makes the multiplication more by a second

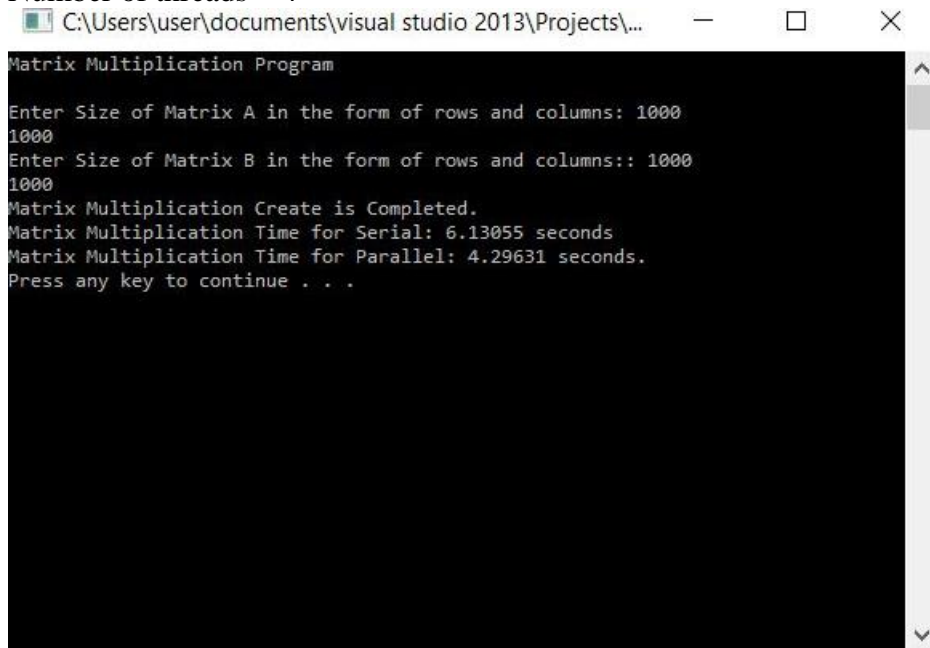
2. Number of threads = 2



```
C:\Users\user\documents\visual studio 2013\Projects\...
Matrix Multiplication Program
Enter Size of Matrix A in the form of rows and columns: 1000
1000
Enter Size of Matrix B in the form of rows and columns:: 1000
1000
Matrix Multiplication Create is Completed.
Matrix Multiplication Time for Serial: 6.12123 seconds
Matrix Multiplication Time for Parallel: 5.57568 seconds.
Press any key to continue . . .
```

It can be seen, when the number of threads is 2, the time taken by the parallel block is less than of the serial one as the threads are increased so multiplication time is decreasing.

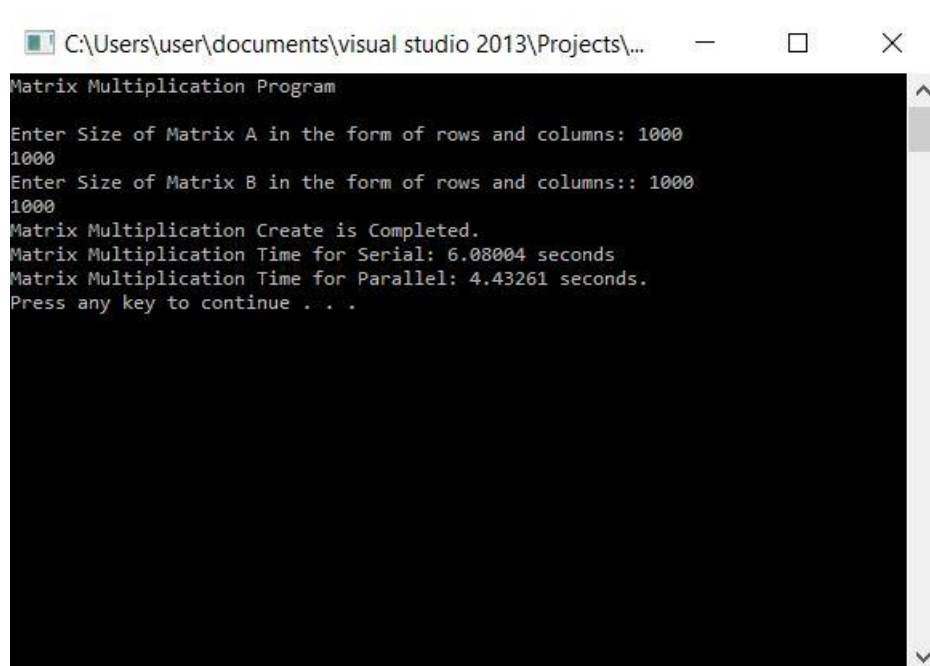
2. Number of threads = 4



```
Matrix Multiplication Program
Enter Size of Matrix A in the form of rows and columns: 1000
1000
Enter Size of Matrix B in the form of rows and columns:: 1000
1000
Matrix Multiplication Create is Completed.
Matrix Multiplication Time for Serial: 6.13055 seconds
Matrix Multiplication Time for Parallel: 4.29631 seconds.
Press any key to continue . . .
```

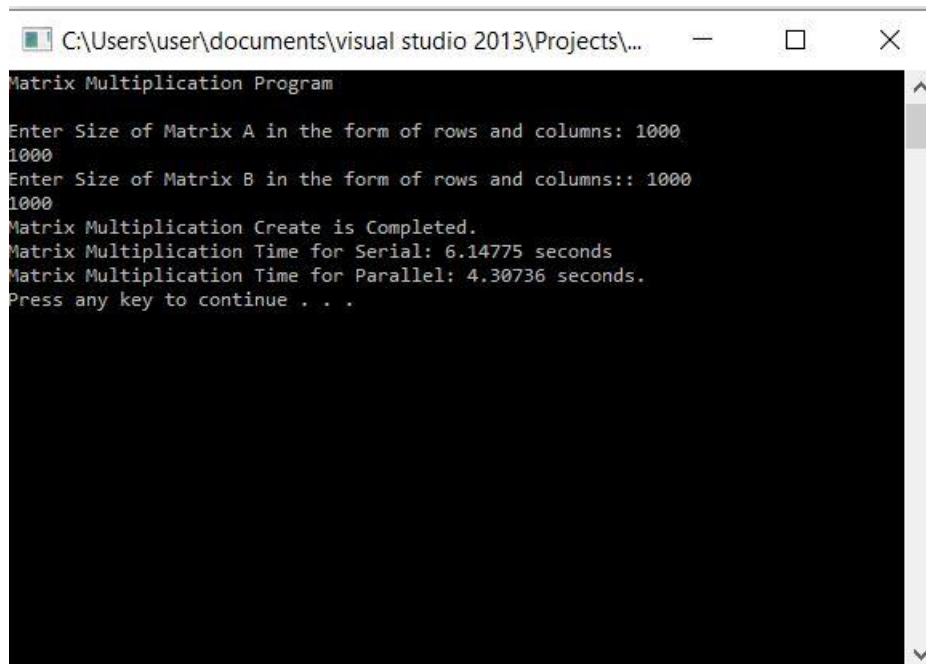
As the threads are 4, then the parallel block is working faster because threads are increased which makes parallel matrix multiplication much faster.

4. Number of threads = 6



```
Matrix Multiplication Program
Enter Size of Matrix A in the form of rows and columns: 1000
1000
Enter Size of Matrix B in the form of rows and columns:: 1000
1000
Matrix Multiplication Create is Completed.
Matrix Multiplication Time for Serial: 6.08004 seconds
Matrix Multiplication Time for Parallel: 4.43261 seconds.
Press any key to continue . . .
```

5. Number of threads = 8



```
C:\Users\user\documents\visual studio 2013\Projects\...
Matrix Multiplication Program
Enter Size of Matrix A in the form of rows and columns: 1000
1000
Enter Size of Matrix B in the form of rows and columns:: 1000
1000
Matrix Multiplication Create is Completed.
Matrix Multiplication Time for Serial: 6.14775 seconds
Matrix Multiplication Time for Parallel: 4.30736 seconds.
Press any key to continue . . .
```

It is clear from the above outputs that the time taken by the parallel block is less than the time taken by the serial block. The time taken by the parallel block tends to decrease as the number of threads increases. The system used to test the code is a 2-core computer. And if the number of threads is more than 2, there's no difference in the time taken for the program to end. But we can see after the thread is more than 2 the time r=of parallel multiplication is almost the same for threads 4,6,8. And if there are more threads than the cores in the CPU, the code can slow down due to overhead thread formation.

References:

- [1] <http://www.bowdoin.edu/~ltoma/teaching/cs3225-GIS/fall16/Lectures/openmp.html>
- [2] <https://bisqwit.iki.fi/story/howto/openmp/>
- [3] OpenMP: A parallel Hello World Program
(<https://www.youtube.com/watch?v=Ka3rBhwMgXg>)
- [4] <https://www.openmp.org/wp-content/uploads/OpenMP3.0-SummarySpec.pdf>
- [5] <https://www.openmp.org/spec-html/5.0/openmpsu160.html>