CA670 Concurrent Programming Assignment - II

Efficient Large Matrix Multiplication in OpenMP

Name: **Akshai Ramesh**

Student Id number: **19210382**

E-mail addresses: akshai.ramesh2@mail.dcu.ie, akshairamesh21@gmail.com

Programme: MSc in Computing (Data Analytics)

Module Code: CA670

Date of Submission: 18-04-2020

A solution submitted to Dublin City University, School of Computing for module CA670 Concurrent Programming, 2019/2020. I understand that the University regards breaches of academic integrity and plagiarism as grave and serious. I have read and understood the DCU Academic Integrity and Plagiarism Policy. I accept the penalties that may be imposed should I engage in practice or practices that breach this policy. I have identified and included the source of all facts, ideas, opinions, viewpoints of others in the assignment references. Direct quotations, paraphrasing, discussion of ideas from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the sources cited are identified in the assignment references. 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. By signing this form or by submitting this material online I confirm that 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. By signing this form or by submitting material for assessment online I confirm that I have read and understood DCU Academic Integrity and Plagiarism Policy (available at: http://www.dcu.ie/registry/examinations/index.shtml)

Name: Akshai Ramesh

Date:18/04/2020

AIM

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

DESIGN

To perform efficient large matrix multiplication, the Blocked Matrix Multiplication Algorithm has been used. First let's consider the Naive approach.
The standard naive approach for matrix multiplication is as follows :

```
for i = 1 to n do
             for j = 1 to n do
                         for k = 1 to n do
                                     c[i,j] = c[i,j] + a[i,k] * b[k,j] ; od
                         od
             od
```

The time complexity for the naive approach is $O(n^3)$. There are $2n^3$ arithmetic operations carried out for only $3n^2$ elements.This can be improved by a huge factor by caching the smaller blocks of data.

Reasons for choosing Block Matrix Multiplication Algorithm:

- In terms of number of operations, the time complexity is way better.
- It is a cache-oblivious algorithm.
- The advantage of this approach is that smaller subarrays can be moved into the fast local memory and can be used repeatedly.

The pseudo-code for blocked matrix multiplication is as follows :

```
for row_iter = 1 to n step b do
      for col_iter = 1 to n step b do

            for pivot = 1 to n step b do

                  for i = row_iter to min(row_iter + b - 1, n) do
```

```
            for j = col_iter to min(col_iter + b - 1, n) do

                              for pivot1 = pivot to min(pivot + b - 1, n)
do

                        c[i, j] = c[i, j] + a[i, pivot1] * b[pivot1, j];
                                    od

                              od

                        od

            od

      od

   od
```

The results obtained using the Block Matrix Multiplication algorithm is same as that of the Naive approach. But, the operations are performed in a different sequence unlike the unblocked code snippet. In Block Matrix Multiplication, for a set of values of row_iter, col_iter and pivot, $2b^3$ operations are carried out and $3b^2$ values are referred to. The smaller value for b will allow $3b^2$ values to fit in the local memory and thus, achieving B-fold reuse.

CODING APPROACH

The code below consists of an optimized approach to implement parallelism using OpenMP.

- **Main.c** : There are 3 matrices A,B and C. A is a matrix of dimension m x n, B is a matrix of dimension n x p. C is the resulting matrix of m x p dimension with values computed as :
  $C = const1\ x\ A\ x\ B + const2\ x\ C$

The number of threads, num_threads and number of blocks, num_blocks is varied across matrices of varying dimensions from 1024 x 1024 to 1536 x 1536 in steps of 256 and the main function tests the efficiency of program.

- **MatMul_omp.c** : The block multiplication algorithm is implemented which allows the execution of parallel threads. The number of threads for each computation takes value from 1, 2 and 4 threads.

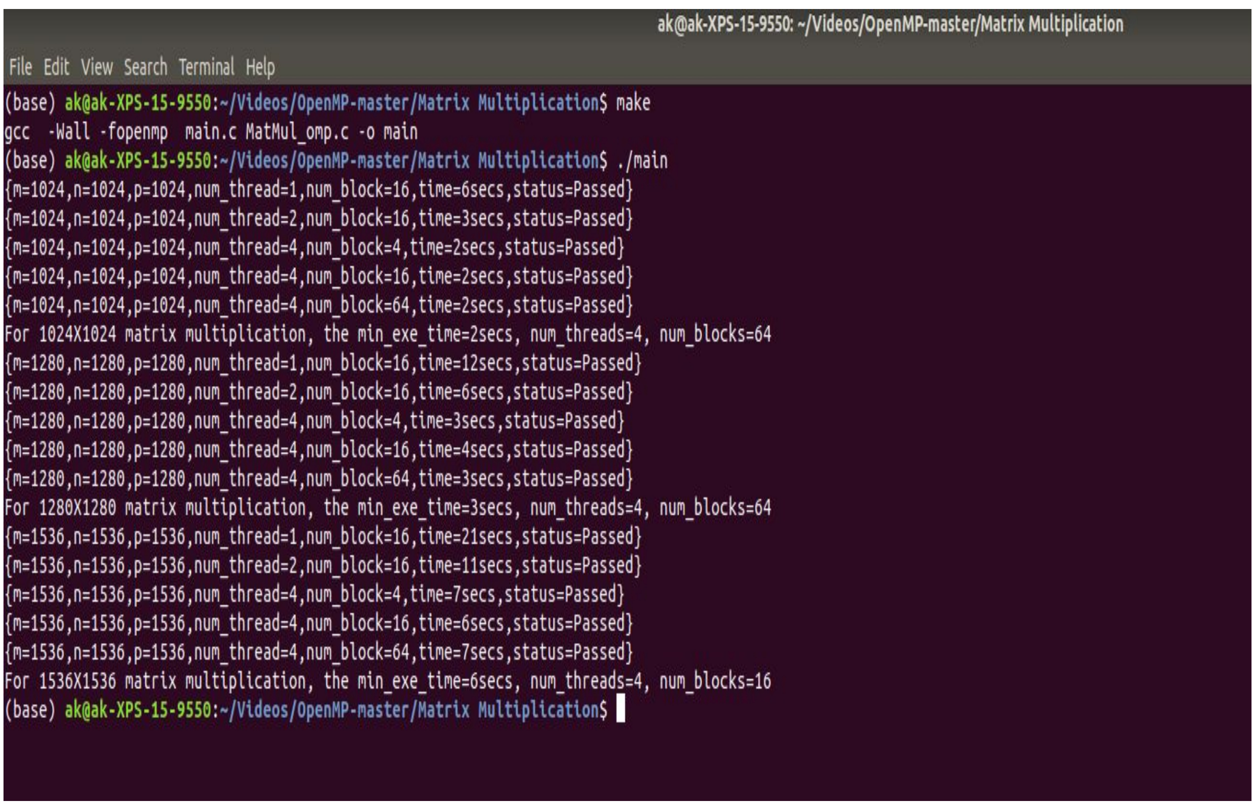
- **Makefile** : The makefile to be executed.

- **Matrix.h** : The header file containing required packages and constant values initialized.

Block Matrix Multiplication Algorithm has the benefit of **Cache Fitting**, as larger matrices are divided into smaller blocks of size b.

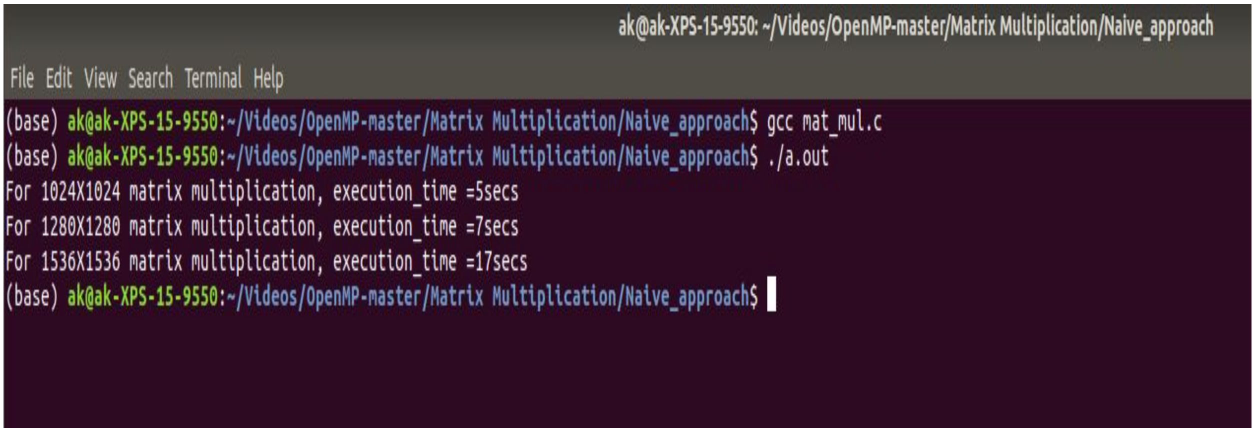
OUTPUT SNIPPETS

OUTPUT 1 : BLOCK APPROACH EXECUTION TIME



```
ak@ak-XPS-15-9550: ~/Videos/OpenMP-master/Matrix Multiplication
File Edit View Search Terminal Help
(base) ak@ak-XPS-15-9550:~/Videos/OpenMP-master/Matrix Multiplication$ make
gcc  -Wall -fopenmp  main.c MatMul_omp.c -o main
(base) ak@ak-XPS-15-9550:~/Videos/OpenMP-master/Matrix Multiplication$ ./main
{m=1024,n=1024,p=1024,num_thread=1,num_block=16,time=6secs,status=Passed}
{m=1024,n=1024,p=1024,num_thread=2,num_block=16,time=3secs,status=Passed}
{m=1024,n=1024,p=1024,num_thread=4,num_block=4,time=2secs,status=Passed}
{m=1024,n=1024,p=1024,num_thread=4,num_block=16,time=2secs,status=Passed}
{m=1024,n=1024,p=1024,num_thread=4,num_block=64,time=2secs,status=Passed}
For 1024X1024 matrix multiplication, the min_exe_time=2secs, num_threads=4, num_blocks=64
{m=1280,n=1280,p=1280,num_thread=1,num_block=16,time=12secs,status=Passed}
{m=1280,n=1280,p=1280,num_thread=2,num_block=16,time=6secs,status=Passed}
{m=1280,n=1280,p=1280,num_thread=4,num_block=4,time=3secs,status=Passed}
{m=1280,n=1280,p=1280,num_thread=4,num_block=16,time=4secs,status=Passed}
{m=1280,n=1280,p=1280,num_thread=4,num_block=64,time=3secs,status=Passed}
For 1280X1280 matrix multiplication, the min_exe_time=3secs, num_threads=4, num_blocks=64
{m=1536,n=1536,p=1536,num_thread=1,num_block=16,time=21secs,status=Passed}
{m=1536,n=1536,p=1536,num_thread=2,num_block=16,time=11secs,status=Passed}
{m=1536,n=1536,p=1536,num_thread=4,num_block=4,time=7secs,status=Passed}
{m=1536,n=1536,p=1536,num_thread=4,num_block=16,time=6secs,status=Passed}
{m=1536,n=1536,p=1536,num_thread=4,num_block=64,time=7secs,status=Passed}
For 1536X1536 matrix multiplication, the min_exe_time=6secs, num_threads=4, num_blocks=16
(base) ak@ak-XPS-15-9550:~/Videos/OpenMP-master/Matrix Multiplication$
```

OUTPUT 2 : NAIVE APPROACH EXECUTION TIME



CODE EFFICIENCY ANALYSIS :

To observe the significance of performance through parallelism, I have compared the code efficiency of Block algorithm with Naive approach for matrix multiplication.

Table 1: Execution time for **Naive approach** with varying dimension matrices**.**

Matrix Dimensions	Execution Time
1024 x 1024	5secs
1280 x 1280	7secs
1536 x 1536	17secs

Table 2 : Execution time for **Block Matrix Multiplication Algorithm** with varying dimension matrices.

Matrix Dimensions	Num_Threads	Num_Blocks	Min. Execution Time
1024 x 1024	4	64	2secs (↓ 3)
1280 x 1280	4	64	3secs(↓ 4)
1536 x 1536	4	16	6secs(**↓ 11**)

In Block Matrix Multiplication approach it is also inferred that the execution time increases with increase in thread but decreases with dimensional increase.

CONCLUSION

As its clearly observed that we have a huge magnitude of difference ranging from 3seconds to a maximum of 11seconds between the 2 approaches, it is evident that the Block Matrix Multiplication Approach has helped to optimize the computation for large matrices.

REFERENCES

1. https://www.techiedelight.com/find-execution-time-c-program/
2. https://github.com/EvanPurkhiser/CS-OpenMPExperiments/blob/master/omp-matrix-mutliplcation/report.md
3. https://stackoverflow.com/questions/2273913/how-would-you-set-a-variable-to-the-largest-number-possible-in-c
4. https://stackoverflow.com/questions/7050798/incompatible-implicit-declaration-of-built-in-function-malloc
5. https://stackoverflow.com/questions/14532169/why-is-a-na%C3%AFve-c-matrix-multiplication-100-times-slower-than-blas