Efficient Large Matrix Multiplication in OpenMP

In this paper, three methods for large matrix multiplication are evaluated and compared with the simple matrix multiplication method in order to report the most efficient method among them based on various matrix sizes. All the three methods are executed using concurrent programming methods of OpenMP in C language.

Description of the design of the program:

Directive #define is used to define the rows and columns of the matrixes. The following matrixes are used in the code for multiplication:

- A, B and C for Non-Square Matrix Multiplication (Method 2 and Method 3)
- D, E, F and G for Square Matrix Multiplication (Method 1)

The macros defined are:

- m = number of rows of Matrix A & Matrix C || rows and columns for matrixes D, E, F and G
- n = number of columns of Matrix A & rows of Matrix B
- o = number of columns of Matrix B & Matrix C

Function get_time() is used to note the time for calculating the execution of the methods. Function **create_matrixX() is defined to build matrixes of various sizes and random values. Memory is allocated to the matrixes dynamically using the malloc() function [1]. Function free_matrix is defined to release the space allocated to the matrixes using the function free() [1].

Indexing variables used are : i, j, k, ii, jj and kk.

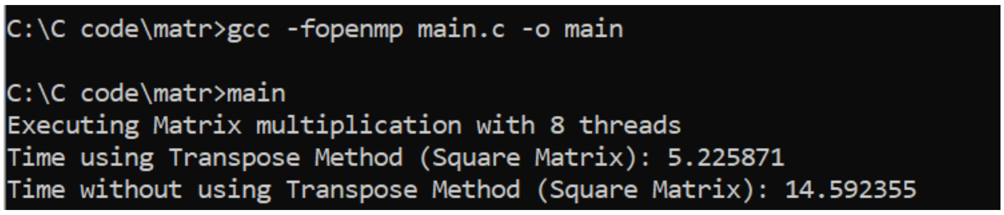
The three methods for matrix multiplication are described below:

- **Method 1:**
  This method produces results only for square matrix multiplication. Non-square matrix multiplication will not produce result using this method. Transpose method for matrix multiplication only produces results when the number of rows and columns of both the matrixes are same [2]. Matrix D and Matrix E are to be multiplied and both of them are square matrix. Matrix E is transposed and the result is saved in Matrix G. Matrix D and G are multiplied using transpose multiplication and the result is stored in Matrix F. Multiplication of D and E produces the same result as multiplication of D and G using transpose method [3].

  The required code for matrix multiplication is parallelized for better efficiency using #pragma omp parallel. Therefore, multiple threads can execute the construct parallelly. In order to avoid race problem, the variables j, k and sum are made private so that each thread has its own private copy of the variable.

Efficiency tested for both the concurrent OpenMP methods - Transpose method and without using Transpose method:

Result with m = 2048:



```
C:\C code\matr>gcc -fopenmp main.c -o main

C:\C code\matr>main
Executing Matrix multiplication with 8 threads
Time using Transpose Method (Square Matrix): 5.225871
Time without using Transpose Method (Square Matrix): 14.592355
```

Method	**Time in seconds**
Transpose	5.22
Without Transpose	14.59

As seen in the results, Transpose Method is proved efficient for large square matrix multiplication and will be considered as Method 1 in this paper.

Note: The code for matrix multiplication without Transpose method is commented out and can be used for verification.

- **Method 2:**
  This method produces results for all kinds of matrix multiplication i.e. square or non-square. Blocking method [4] is applied to improve locality and cache's performance with different variations or patterns of the indexing variables. The most efficient pattern of the indexing variables found after the analyses is used in the program which is jj-kk-i-j-k (bijk) [5].

  The required code for matrix multiplication is parallelized for better efficiency using #pragma omp parallel. In order to avoid race problem, the variables kk, i, j, k and sum are made private so that each thread has its own private copy of the variable. It is not necessary to make the most outer loop indexing variable 'jj' private.

- **Method 3:**
  This method also produces results for all kinds of matrix multiplication i.e. square or non-square. Blocking method [4] is used with the naive pattern of the indexing variables which is ii-jj-kk-i-j-k [5].

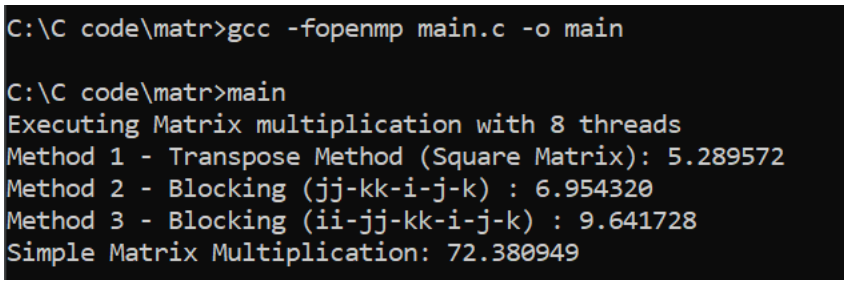
  The required code for matrix multiplication is parallelized for better efficiency using #pragma omp parallel. In order to avoid race problem, the variables jj, kk, i, j, k and sum are made private so that each thread has its own private copy of the variable. It is not necessary to make the most outer loop indexing variable 'ii' private.

Note: Different types of concurrency methods were applied at different places with variation to the loop construct of matrix multiplication like making some variables private, using a reduction variable, tried collapse at different places in the loop, etc. None of them produce greater efficiency than the ones used in the program. But those codes are commented in the program for reference.

Evaluation of the efficiency of the methods:

Block-size determination for Blocking methods: After researching [6] and executing with a lot of variation in block-size, it is found that if the matrix is of the size $2^N * 2^N$, then the efficient block size is $2^{N/2}$. If N is odd, then round-off (N/2).

Result with m = n = o = 2048 and block-size = 64 is shown below:



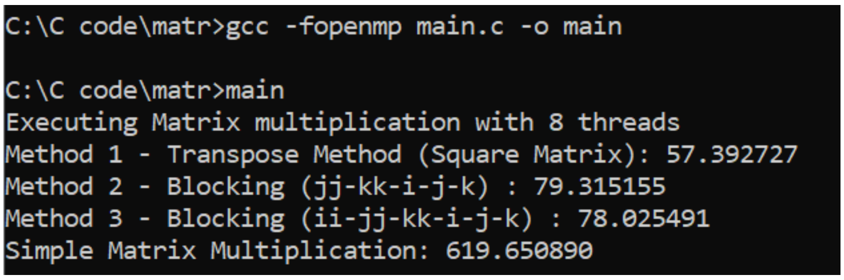
```
C:\C code\matr>gcc -fopenmp main.c -o main

C:\C code\matr>main
Executing Matrix multiplication with 8 threads
Method 1 - Transpose Method (Square Matrix): 5.289572
Method 2 - Blocking (jj-kk-i-j-k) : 6.954320
Method 3 - Blocking (ii-jj-kk-i-j-k) : 9.641728
Simple Matrix Multiplication: 72.380949
```

Method	**Time in seconds**
1.Transpose method (Only for square matrix)	5.29
2.Blocking (jj-kk-i-j-k)	6.95
3.Blocking (ii-jj-kk-i-j-k)	9.64
Simple Matrix Multiplication	72.38

It is observed from the results that if it is Square Matrix Multiplication, then the most efficient method is the Method 1 - Concurrent Transpose method. The results are evaluated with a lot of variation in the matrix size and in all the cases Method 1 showed more efficiency than the other two. But Method 1 is only applicable for Square Matrix Multiplication.

Method 2 and Method 3 produce results for Matrix Multiplication of any size. It is observed from the results that most of the times Method 2 show more efficiency than the Method 3 when the matrix size is $2^{11}*2^{11}$ or small.

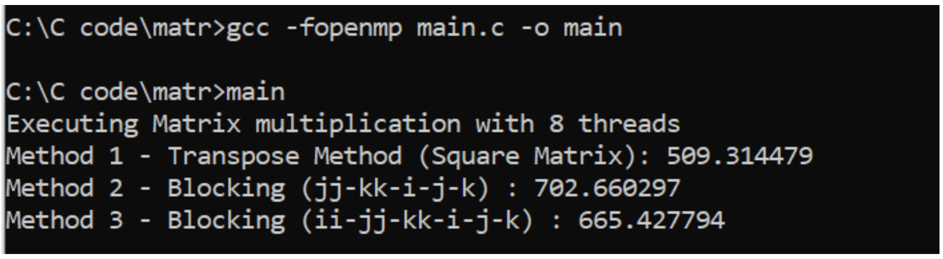
Result for $2^{12}*2^{12}$ i.e m = n = o = 4096 and block-size = 64 is shown below:



```
C:\C code\matr>gcc -fopenmp main.c -o main

C:\C code\matr>main
Executing Matrix multiplication with 8 threads
Method 1 - Transpose Method (Square Matrix): 57.392727
Method 2 - Blocking (jj-kk-i-j-k) : 79.315155
Method 3 - Blocking (ii-jj-kk-i-j-k) : 78.025491
Simple Matrix Multiplication: 619.650890
```

Method	**Time in seconds**
1.Transpose method (Only for square matrix)	57.39
2.Blocking (jj-kk-i-j-k)	79.32
3.Blocking (ii-jj-kk-i-j-k)	78.03
Simple Matrix Multiplication	619.65

Result for $2^{13} * 2^{13}$ i.e. m = n = o = 8192 and block-size = 128 is shown below:



```
C:\C code\matr>gcc -fopenmp main.c -o main

C:\C code\matr>main
Executing Matrix multiplication with 8 threads
Method 1 - Transpose Method (Square Matrix): 509.314479
Method 2 - Blocking (jj-kk-i-j-k) : 702.660297
Method 3 - Blocking (ii-jj-kk-i-j-k) : 665.427794
```

Method	**Time in seconds**
1.Transpose method (Only for square matrix)	509.31
2.Blocking (jj-kk-i-j-k)	702.66
3.Blocking (ii-jj-kk-i-j-k)	665.43

However, it is observed from the results that Method 3 performs better than Method 2 for a matrix multiplication of size $2^N * 2^N$ where N is greater than 11.

Conclusion:

Method 1 performs better than Method 2 and Method 3 if it is a square matrix multiplication. Method 1 is not considered if the matrixes are not square. Method 2 and Method 3 can multiply matrixes of any variation of sizes. Most of the times, Method 2 performs better than Method 3 if N is less than 12 for a $2^N * 2^N$ matrix multiplication. In case N is greater than 11, Method 3 seems to perform much better than Method 2. In comparison with the simple matrix multiplication method, all the three methods described in this paper performed well with excellent efficiency and can be applied accordingly based on the size of the matrixes to achieve utmost efficiency.

Matrix size / type	**Most Efficient Method**
Square Matrix Multiplication	Method 1
Square / Non-square Matrix Multiplication ($N < 12$ for matrixes of size $2^N * 2^N$)	Method 2, Method 3 (Most of the times, Method 2 work faster than Method 3)
Square / Non-square Matrix Multiplication ($N > 11$ for matrixes of size $2^N * 2^N$)	Method 3

Reference:

1. https://www.programiz.com/c-programming/c-dynamic-memory-allocation
2. https://www.sciencedirect.com/topics/computer-science/matrix-multiplication
3. https://www.macs.hw.ac.uk/~hwloidl/Courses/F21DP/srcs/matrix4.c
4. http://www.netlib.org/utk/papers/autoblock/node2.html
5. https://cs61.seas.harvard.edu/wiki/images/0/0f/Lec14-Cache_measurement.pdf
6. https://stackoverflow.com/questions/47685422/best-block-size-value-for-block-matrix-matrix-multiplication