# ELEN4020 LAB 1 - 3D by 3D Multiplication using OpenMP and POSIX Threads technique

*School of Electrical and Information Engineering, University of the Witwatersrand, Johannesburg 2050*

**Ntladi Mohajane**-1599953, **Phetolo Malele**-1412331, **Zwavhudi Mulelu**-1110574

## I. Introduction

Modern computers have the ability to 'multi-task', and this skill is permitted by parallel computing. The parallel computing technique utilizes numerous compute resources to solve computational problems [1]. In parallel computing, a problem is divided into smaller distinct tasks to be solved concurrently. Each task is further broken down into a series of instructions that execute simultaneously, but on different processors.

This lab exercise presents an application of parallel programming through matrix multiplication. The two types of shared memory programming methods - POSIX Threads(PThreads) and OpenMP are used. The two methods are analysed using 2D and 3D matrix multiplication of different sizes. Section 2 below describes how these algorithms were implemented in both 2D and 3D. Results are then analysed in section 3 below.

## II. Algorithm Description

### A. 2D Multiplication

Each of the matrices used in the 2D multiplication algorithm is represented by a one dimensional array as shown in Figure 2 in appendix. Once initialized, each thread is allocated an index to calculate by means of round robin distribution and work concurrently towards the final solution. This means that in theory, the more threads the algorithm has to work with, the less work each thread will do, thus improving performance. By representing the matrices in this manner, it is easier to achieve concurrency and avoid race conditions. This method also has the benefit of requiring very little synchronisation which further improves performance. However, the algorithm is less effective if the number of threads given is higher than N*N, where N is the length of the matrix.

The PThreads and OpenMP parallelization methods are used to generate and manage the threads. The PThreads implementation is a lower level approach to parallel programming that gives finer control over attributes and synchronisation to the developer. On the other hand, the OpenMP implementation is a higher level approach that simplifies paralellization by requiring less initialization on the the developer's part. Both methods have their advantages and disadvantages which will be further explored in section 3.

### B. 3D Multiplication

The 3D multiplication algorithm represents each matrix as a stack of 2D matrices as shown in Figure 1. Each layer of stack is handled one at a time by the 2D multiplication algorithm. Although this implementation allows for the reuse of existing code, it forces the increased use of barrier synchronisation which can be detrimental to performance.

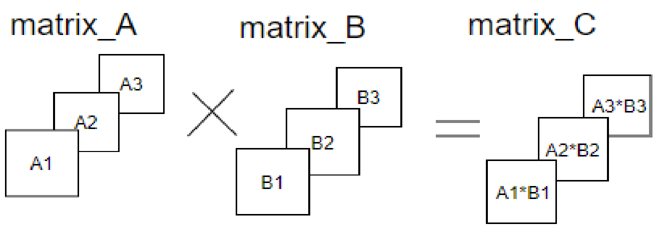

Figure 1. 3-D matrix multiplication illustration

### C. *Matrix Generation*

Each time the algorithm runs, the matrices to be multiplied are populated by random numbers that use time as a seed thereby (in theory) guaranteeing that the same matrix does not appear twice. However, the relatively low range (0-20) of the matrices means that this is not always true in practice. In addition, each 2D matrix is populated using a single thread as well as a single for-loop. 3D matrices are populated by repeatedly using this procedure for each of their layers. Moreover, it is assumed that the time taken to populate a matrix is the same between each test run.

## III. Performance Analysis

Figures 3-12 in the appendix show the performance of PThreads and OpenMP parallelization implementations. They show that on average the more threads that the algorithm uses to solve the problem, the better performance of the algorithm. However, performance gain is limited to the number of processors available on the machine running the algorithm. The specifications of the machine used to gather the results are as follows:

- Intel Core i7-6700 CPU @ 3.4GHz (4 CPU Cores, 8 Logical Threads due to HyperThreading).
- 8.00 GB 2133MHz Dual Channel DDR4 RAM
- Linux Ubuntu 16.04 LTS (64-bit)

Furthermore, the matrix multiplication algorithm is written in C and compiled with the gcc compiler (Version 5.5.0) for Ubuntu. Testing was done using a bash script that was run multiple times and the average result was recorded.

### A. *2D Multiplication*

Results obtained from testing the performance of both the OpenMP and PThreads parallelization implementations on the 2D matrix multiplication algorithm are shown in Figures 3, 4, 5, 6 and 7. It is shown that for small matrices (10x10, 20x20 and 30x30), an increase in the number of threads does not have an effect on the algorithm's performance (See Figure 3 ,4 and 5). In fact, performance decreased after the algorithm is given more threads than physical processors, which implies a hardware bottleneck.

Furthermore, the relatively small time needed to perform the algorithm in this case highlights the difference in overhead for the two parallelization implementations. If it is assumed that the time taken to compute each index of the resultant matrix is the same for both implementations, one can attribute the difference in processing time to initializing the threads needed to perform the computation. The added time is significantly greater for the OpenMP implementation. This is a consequence of it running extra code in the background as a result of it being a higher level implementation than PThreads.

However, for large matrices such as a 100x100 and 1000x1000, the overhead is less pronounced due to the increased number of computations. (See Figures 6 and 7). In addition, the performance scales better with an increase in thread count. However, this is only true up until the thread count equals the number of physical processors thus further implying a hardware bottleneck.

### B. *3D Multiplication*

Results obtained from testing the performance of both the OpenMP and PThreads parallelization implementations on the 3D matrix multiplication algorithm are shown in Figures 8, 9, 10, 11 and 12 in the appendices below. For smaller 3D array sizes (10x10x10, 20x20x20 and 30x30x30), the computation time for each algorithm decreases as the number of threads increase. However, this is only true if the number of threads used in the computation is less than or equal to the logical processor cores. After this point, the increased overhead from initializing the threads and barrier synchronization severely hinders performance.

The scaling of the algorithm improves for the larger matrices (100x100x100 and 250x250x250). In addition, the overhead for initializing the threads is now negligible due to the increased number of computations. A consequence of the way this specific 3D multiplication algorithm is implemented is increased barrier synchronization. In almost every instance, PThreads takes longer to run than OpenMP for 3D matrices. This implies that OpenMP is more efficient at handling synchronization than PThreads for this specific algorithm implementation.

## IV. References

[1] Computing.llnl.gov. (2020). Introduction to Parallel Computing. [online] Available at: https://computing.llnl.gov/tutorials/parallel/comp [Accessed 25 Feb. 2020].

## APPENDIX

### A. *populate2D*

**Algorithm 1:** Populates 2D matrices with random elements

```
Result: 2D matrices(matrix_A, matrix_B) with random numbers
Input  : Pointer to matrix_A, Pointer to matrix_B
Input  : length
1 for i = 0 to length*length - 1 do
2     mat_A[i] = rand()% 21
3     mat_B[i] = rand()% 21
```

### B. *populate3D*

**Algorithm 2:** Populates 3D matrices with random elements

```
Result: 3D matrices(matrix_A, matrix_B) with random numbers
Input  : matrix_A double pointer , matrix_B double pointer
Input  : length, depth
1 for i = 0 to depth - 1 do
2     populate2D(Mat_A_3D[i], Mat_B_3D[i], length)
```

### C. *indexMultiplication*

**Algorithm 3:** multiplies each row of a matrix_A with col of matrix_B

```
Result: matrix_C
Input  : index
1 Declare MAT_LENGTH and MAT_SQUARE as global variables
2 Initialize sum to 0
3 for row = 0, col = 0 to MAT_LENGTH - 1 do
4     row_offset = (index / MAT_LENGTH) * MAT_LENGTH
5     col_offset = (index % MAT_LENGTH)
6 matrix_C[index] = sum
```

### D. *rank2TensorMultPThread*

**Algorithm 4:** 2D matrix multiplication using PThreads

```
Result: 2D matrix_C as a result of multiplying matrix_A and matrix_B
1 Declare matrix_A, matrix_B and matrix_C as pointers globally
2 Declare NUM_THREADS, MAT_LENGTH and MAT_SQUARE as global variables
3 In main:
4 Define number of threads and matrix size
5 Dynamically allocate memory for 2D matrices( matrix_A, matrix_B and matrix_C) of size MAT_LENGTH
6 Populate matrix_A and matrix_B with random numbers
7 for index = 0 to NUM_THREADS - 1 do
8     thread_ids[index] = index
9     pthread_create(&threads[index], NULL, pThreadsPointer, &thread_ids[index])
10 for index = 0 to NUM_THREADS - 1 do
11     pthread_join(threads[index], NULL);
```

### E. *rank2TensorMultOpenMP*

**Algorithm 5:** 2D matrix multiplication using OpenMP

```
Result: 2D matrix_C as a result of multiplying matrix_A and matrix_B
Set number of threads for OpenMP
#pragma omp parallel
thread_id = omp_get_thread_num()
threads_given = omp_get_num_threads()
for index = thread_id to MAT_SQUARE - 1 do
    indexMultiplication(index)
```

### F. *rank3TensorMultPThread*

**Algorithm 6:** 3D matrix multiplication using PThread

```
Result: 3D matrix_C as a result of multiplying matrix_A and matrix_B
Declare MAT_DEPTH
Declare matrix_A, matrix_B and matrix_C as double pointers in the global space
Declare IndexData struct that contains depth and thread_id integers
for index = 0 to MAT_DEPTH - 1 do
    Create NUM_THREADS thread identities
    Create a data variable of type struct
    for index = 0 to NUM_THREADS - 1 do
        data[index].thread_id = index
        data[index].depth = depth
        pthread_create(&threads[index], NULL, pThreadsPointer, &data[index]);
for index = 0 to NUM_THREADS - 1 do
    pthread_join(threads[index], NULL);
```

### G. *rank3TensorMultOpenMP*

**Algorithm 7:** 3D matrix multiplication using OpenMP

```
Result: 2D matrix_C as a result of multiplying matrix_A and matrix_B
Set number of threads for OpenMP
#pragma omp parallel
thread_id = omp_get_thread_num()
threads_given = omp_get_num_threads()
for depth = 0 to MAT_DEPTH - 1 do
    for index = thread_id to MAT_SQUARE - 1 do
        indexMultiplication(index, depth)
```

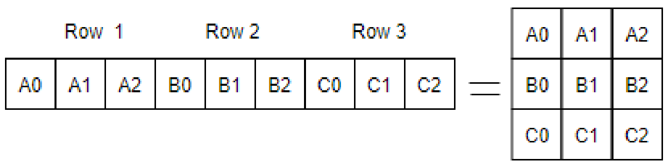

Figure 2. Example 3x3 2-D matrix

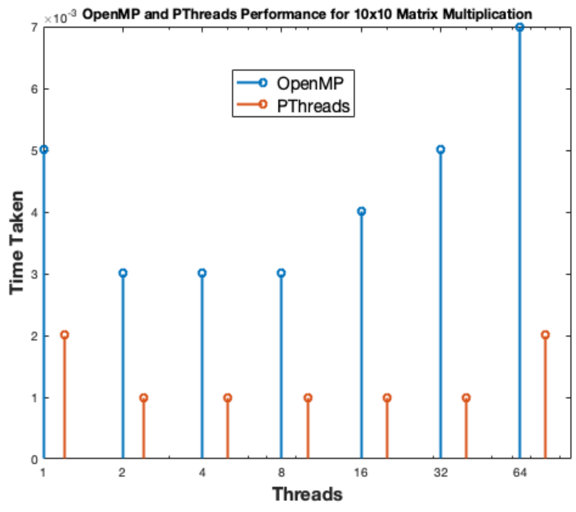

Figure 3. 10x10 matrix multiplication results

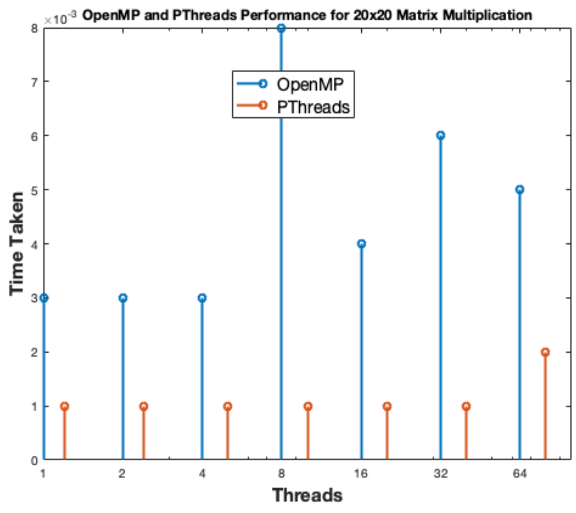

Figure 4. 20x20 matrix multiplication results

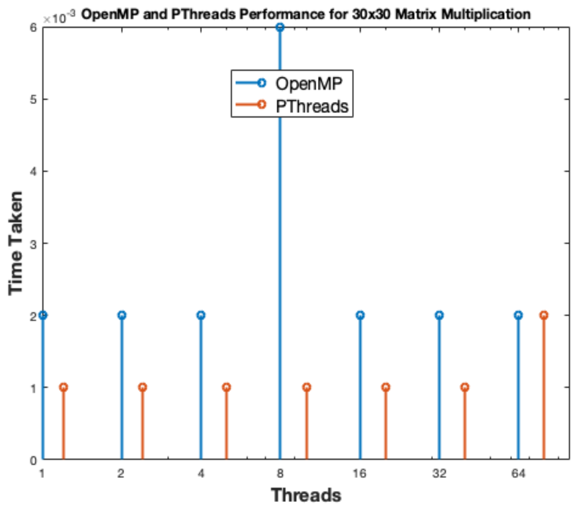

Figure 5. 30x30 matrix multiplication results

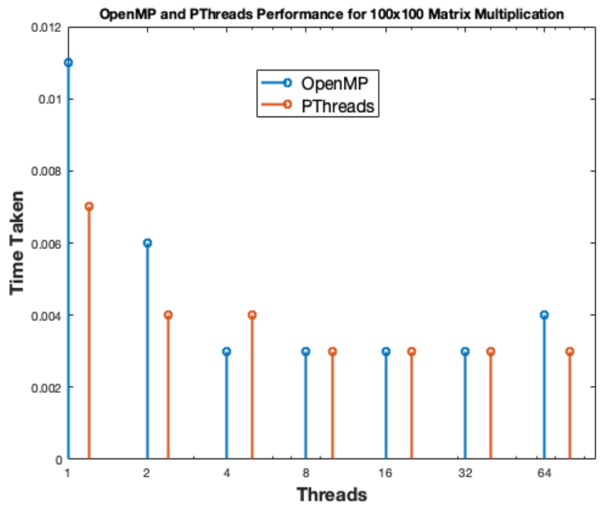

Figure 6. 100x100 matrix multiplication results

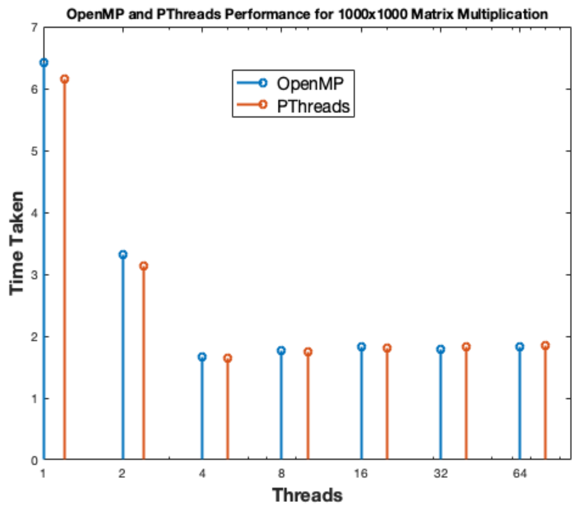

Figure 7. 1000x1000 matrix multiplication results

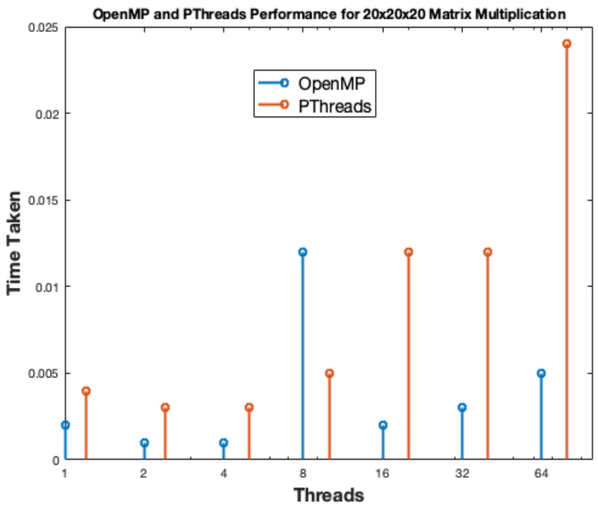

Figure 9. 20x20x20 matrix multiplication results

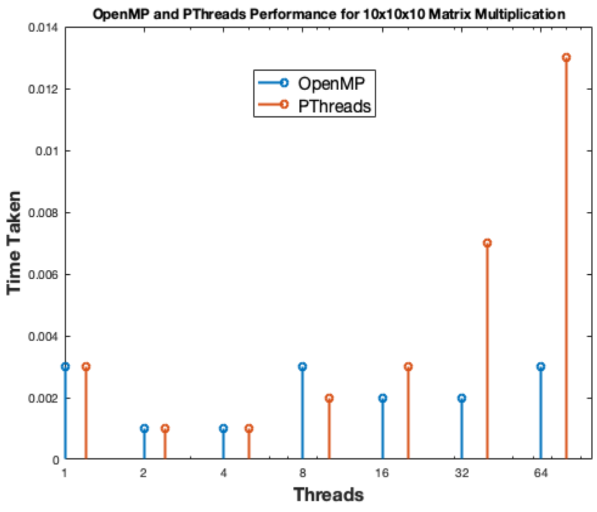

Figure 8. 10x10x10 matrix multiplication results

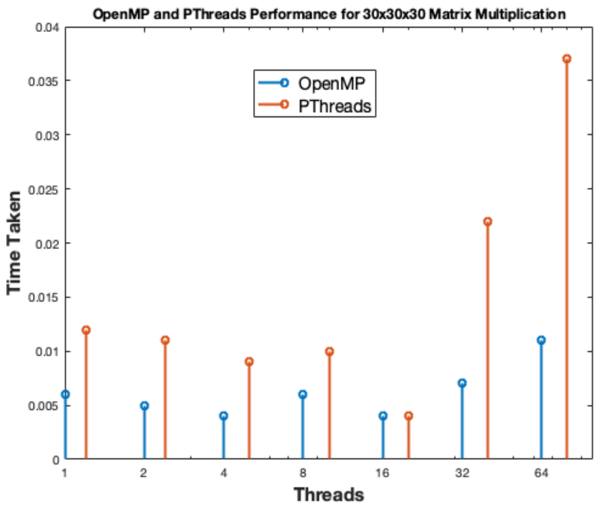

Figure 10. 30x30x30 matrix multiplication results

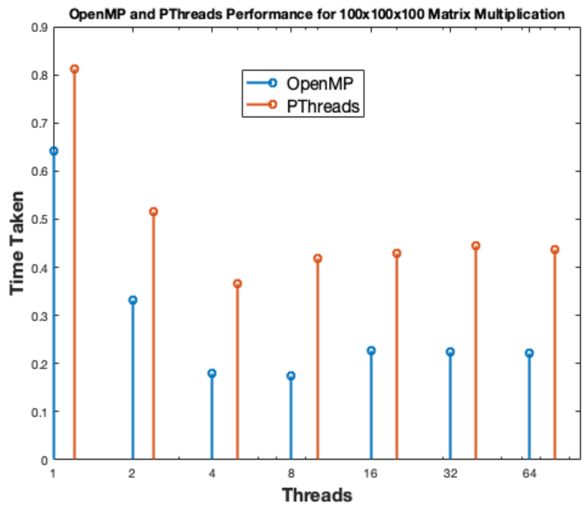

Figure 11. 100x100x100 matrix multiplication results

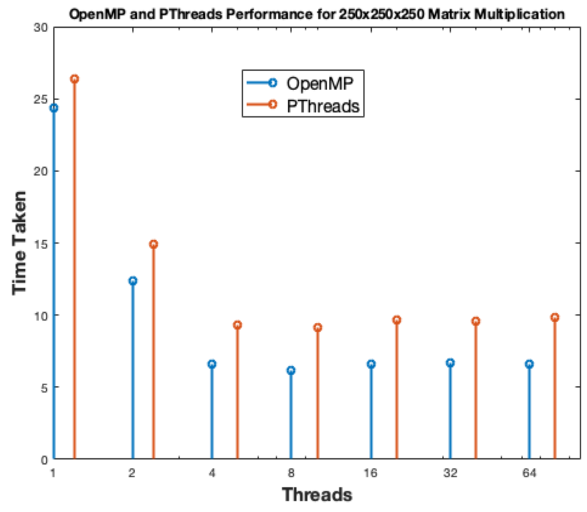

Figure 12. 250x250x250 matrix multiplication results