National University of Computer and Emerging Sciences-FAST Karachi Campus

PDC Project Report

Submitted To:

Miss Nausheen Shoaib

(Department of Computer Science)

Submitted by:

Abu-Bakr Kaleem 18K-1032

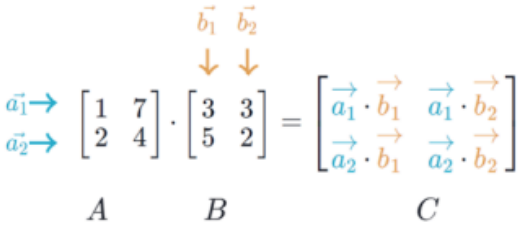
Bilal Asim 18K-1073

Sami 18k-1147

Section: C (PDC)

Introduction:

This project discusses the ability to utilize parallel processing to solve the problem of matrix multiplication, especially multiplying two matrices of the squared size of MxM of randomly generated numbers. Which is simply consisting of applying multiple dot products between the rows and the columns of the two matrices:



Matrix multiplication falls under linear algebra, and fortunately, it is a very suitable operation for parallel processing

Objective:

Parallel processing is a method in computing of running two or more processing units, by dividing the problem into sub-problems, and each processing unit process a portion of the problem. By doing so, will help reduce the amount of time to run a program. Parallel processing is commonly used to perform complex tasks and computations.

The aim of this project is to use the power of parallel processing on applying matrix multiplication in different sizes of matrices and a different number of processing units. Parallel processing can be utilized using multiple schemes, such as MPI, OpenMP, and more. The project follows the OpenMP scheme, since it uses shared memory as a way of communicating between processing units, which makes it easier for the programmer to use parallel processing. On the other hand, MPI uses message passing to communicate among processing units, which leads to more effort at the programmer side.

Project description & Code:

OpenMP API has been employed to perform the parallelism to solve the problem. The OpenMP API is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications on platforms ranging from embedded systems and accelerator devices to multicore systems and shared-memory systems.

The methods used in the program are as follows:

void matrix_multiplication(int l, int m, int n): which initializes three matrices a, b, c of the dimensions l, m, n respectively. The aim of this function is to perform the matrix multiplication operation. The implementation of parallelism has been located in this procedure.

double generate_random(int *seed): It is called by the previous procedure to generate random numbers to help to fill the matrices.

int main(int argc, char *argv[]): which is used to execute the whole program.

OpenMP Statements Used:

1. omp_get_wtime(); to get a value in seconds of the time elapsed from some point. (to get current time in seconds)

2. omp_set_num_threads(NUM_THREADS); to set the number of threads using.

3. omp_get_thread_num(); to get the thread number.

4. omp_get_num_threads(); to get the number of the actual working threads.

5. omp_get_num_procs(); to get the number of processors.

6. omp_get_max_threads(); to get the maximum number of threads.

Code Explanation:

In this block code, the core function is matrix_multiplication(int l, int m, int n), which executes all the parallel code. Firstly, the initialization of the variables and allocation of matrices a, b, and c has been done. Matrix a is the resulting matrix of multiplication of b and c matrices.

```
33
34 void matrix_multiplication(int l, int m, int n){
35     double *A,*B,*C;
36     int i,j,k,ops,seed;
37     double rate;
38     double time_begin;
39     double funcTime;
40     double time_elapsed;
41     double time_stop;
42
43     A = (double *)malloc(l * n * sizeof(double));
44     B = (double *)malloc(l * m * sizeof(double));
45     C = (double *)malloc(m * n * sizeof(double));
46
47
48     seed = 123456789;
49     for (k = 0; k < l * m; k++){
50
51         B[k] = generate_random(&seed);
52     }
53     for (k = 0; k < m * n; k++){
54
55         C[k] = generate_random(&seed);
56     }
57
```

Secondly, where filling the matrices is done. So, what it does is iterating over a matrix and fill out each element with a random number generated by the matrix_multiplication() function. In order to optimize memory allocation, a one-dimensional array is used to store data. What is the purpose of using a one-dimensional array to store 2D matrices? The purpose is because computer memory is much more sequential, even if the program perceives a square matrix as a 2D matrix. It is better at sequential access. When it comes to caching, it prefers related data together. In some caching algorithms, the data required for the next operation will be available in the next memory location. Then it can easily load those data to cache before the CPU requests those data. So, it is better to use the 1D approach, since it is faster, where it offers better memory locality, and less allocation and deallocation overhead. requests those data.

```
59      funcTime = omp_get_wtime();
60      time_begin = omp_get_wtime();
61      omp_set_num_threads(NUM_THREADS);
62      int nthreads, id;
63
64  #pragma omp parallel shared(A, B, C, l, m, n) private(i, j, k)
65  {
66  #pragma omp for
67      for (j = 0; j < n; j++)
68      {
69          for (i = 0; i < l; i++)
70          {
71              A[i + j * l] = 0.0;
72              for (k = 0; k < m; k++)
73              {
74                  A[i + j * l] = A[i + j * l] + B[i + k * l] * C[k + j * m];
75              }
76          }
77      }
78
79
80        // for (j = 0; j < n; j++)
81     // {
82     // printf("%f",a[j]);
83    //  }
84
85  }
86      time_stop = omp_get_wtime();
87
88
89      ops = l * n * (2 * m);
90      time elapsed = time stop - time begin -(time begin - funcTime);
```

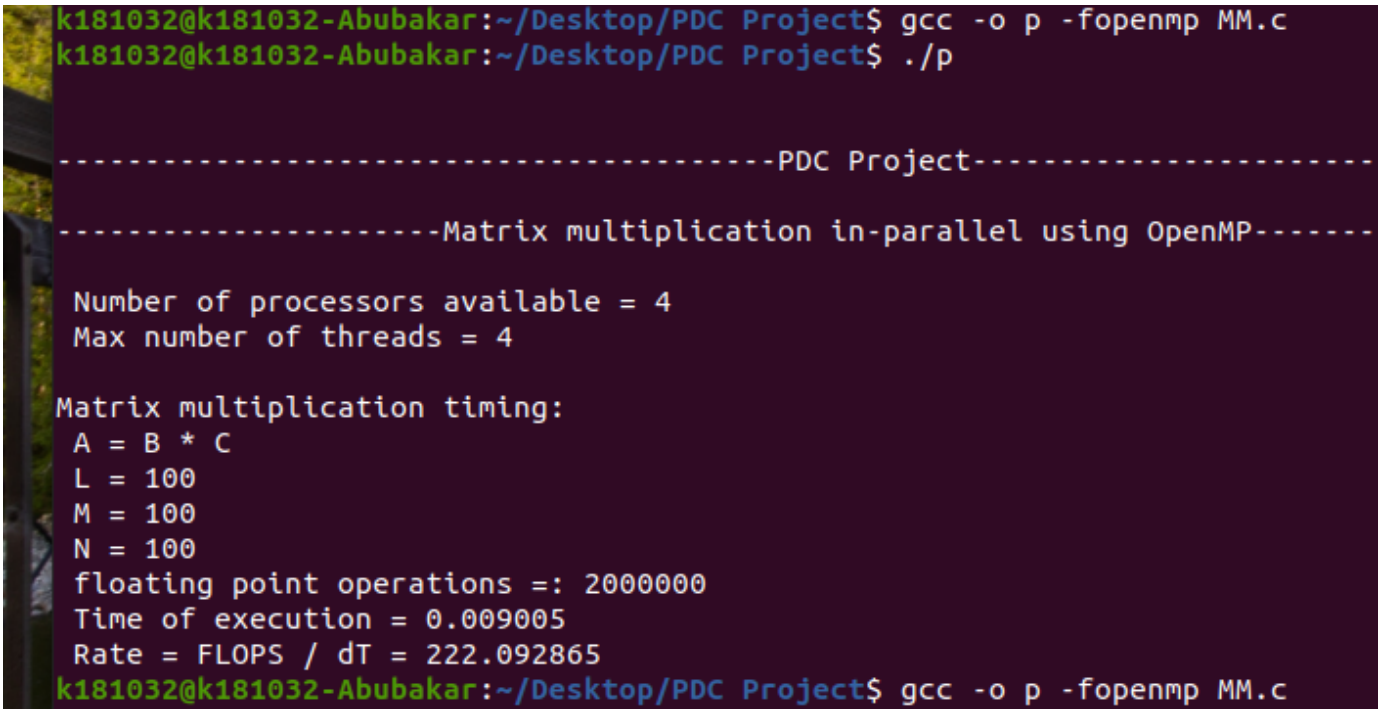
Here, funcTime, time_begin, and time_stop are used for calculating the execution of time for parallel code. Next, is the start of the parallel region. Notice that, shared and private clauses are employed to specify the shared and private variables among the processing units. The statement #pragma omp for has been exploited to parallelize the outermost for loop. Using this statement, it delegates portions of the for loop for different processing units.

Time Measurement:

First, the parallel program run-time has been calculated by taking the average of five runs, after removing outlier results. Each of different matrix sizes and different processing units.

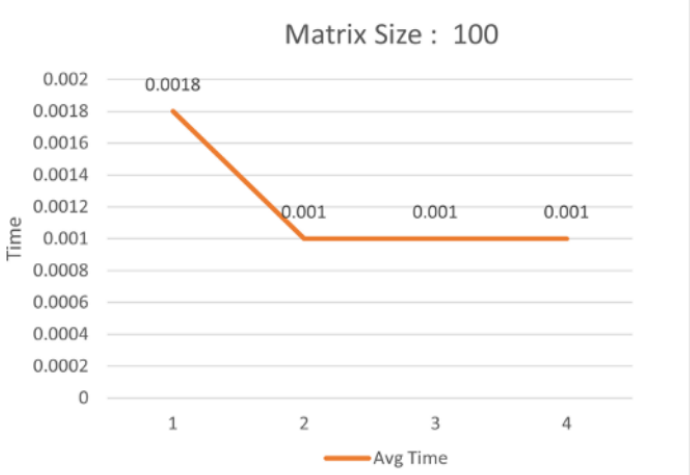
Second, the time of calling the function omp_get_wtime() has been excluded, by calling it twice in a row and subtract that time from the total time. And that captured by the code chunk (time_begin - funcTime)

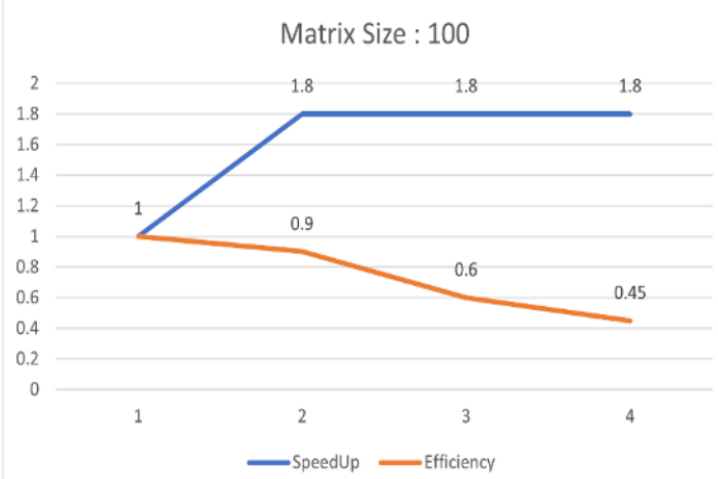
Sample Outputs:

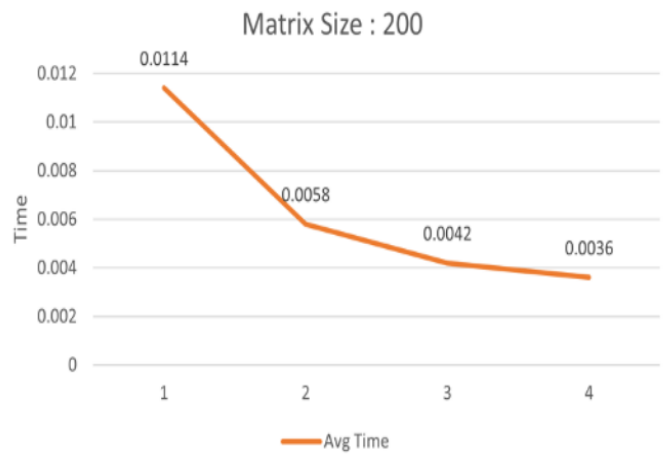


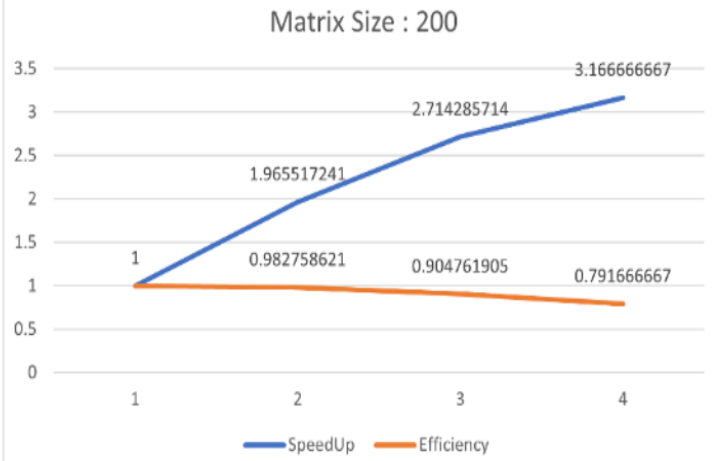
```
k181032@k181032-Abubakar:~/Desktop/PDC Project$ ./p


----------------------------------------PDC Project-------------------------

----------------------Matrix multiplication in-parallel using OpenMP---------

 Number of processors available = 4
 Max number of threads = 4

Matrix multiplication timing:
 A = B * C
 L = 3
 M = 3
 N = 3
 floating point operations =: 54
 Time of execution = 0.000178
 Rate = FLOPS / dT = 0.303832
k181032@k181032-Abubakar:~/Desktop/PDC Project$
```

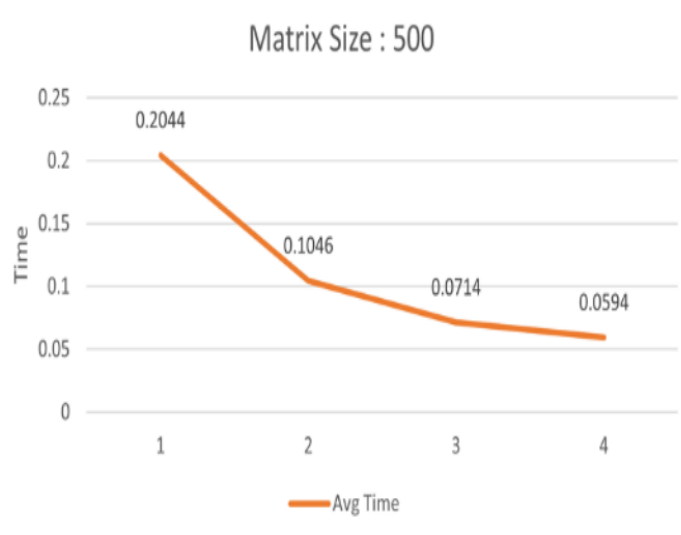
Data Graphs:

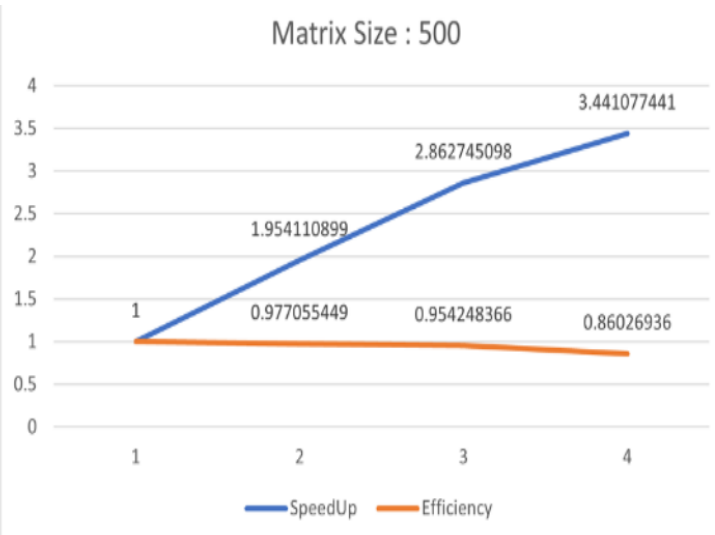


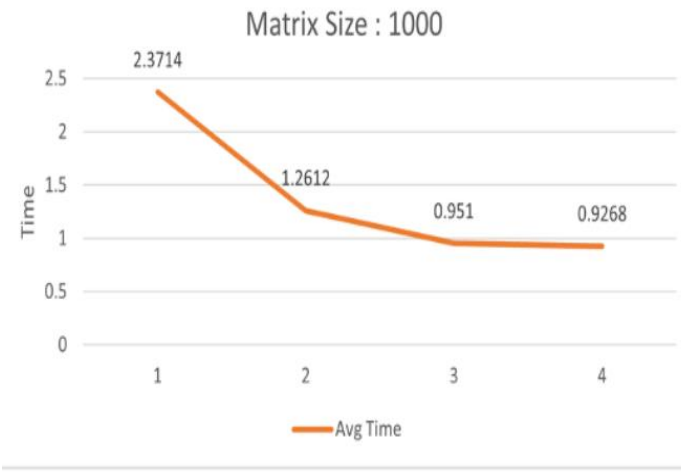


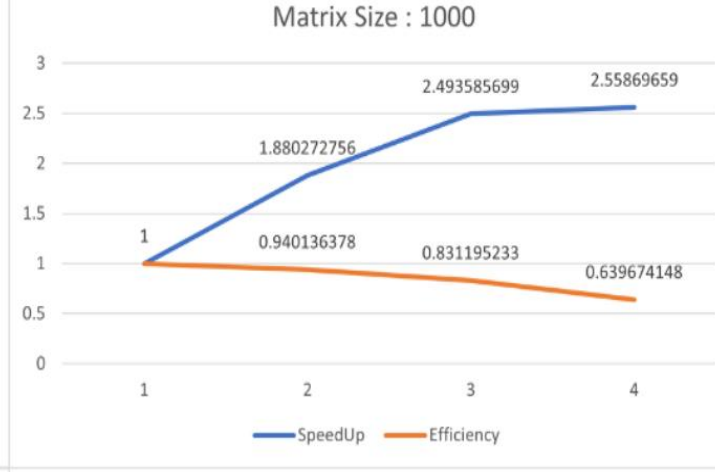












The graphs show that the speedup increases as the number of processing units increases . But looking at the efficiency, it decreases as the number of processing units increases.

Moreover, the average time slows as the number of processing units increases.

And that applied to all the various sizes used in this experiment, except for the matrix of size 100, as it is difficult here to observe the difference between the runs of different processing units. For the reason that with small sizes of a problem, mostly, different processing units will result in time nearly the same.

Conclusion:

After exploring parallel processing on the problem of matrix multiplication, it appears to be that parallel processing can handle complicated problems in less time than serial processing.

In parallel processing, in this problem specifically, the efficiency is reduced as the number of processing units rises. This is due to the lack of utilizing all processing units to their full capability, In the whole run-time of the program.

Average Time:

RUN#	p	M	Time1	Time2	Time3	Time4	Time5	Avg Time
1	1	100	0.002	0.003	0.001	0.002	0.001	0.0018
2	2	100	0.001	0.001	0.001	0.001	0.001	0.001
3	3	100	0.001	0.001	0.001	0.001	0.001	0.001
4	4	100	0.001	0.001	0.001	0.001	0.001	0.001
5	1	200	0.012	0.011	0.011	0.011	0.012	0.0114
6	2	200	0.007	0.006	0.005	0.006	0.005	0.0058
7	3	200	0.004	0.003	0.005	0.005	0.004	0.0042
8	4	200	0.004	0.003	0.003	0.004	0.004	0.0036
9	1	500	0.208	0.207	0.199	0.202	0.206	0.2044
10	2	500	0.101	0.101	0.105	0.105	0.111	0.1046
11	3	500	0.069	0.07	0.073	0.073	0.072	0.0714
12	4	500	0.065	0.057	0.06	0.063	0.052	0.0594
13	1	1000	2.476	2.272	2.383	2.35	2.376	2.3714
14	2	1000	1.159	1.338	1.321	1.316	1.172	1.2612
15	3	1000	0.822	1.031	0.86	1.064	0.978	0.951
16	4	1000	0.934	0.967	0.982	0.818	0.933	0.9268
17	1	2000	37.861	38.372	38.088	38.651	39.035	38.4014
18	2	2000	19.958	19.176	19.587	19.607	19.776	19.6208
19	3	2000	14.081	14.29	14.909	13.984	14.16	14.2848
20	4	2000	12.849	12.803	12.103	12.45	12.837	12.6084