Parallel matrix multiplication using MPI

Lee Kai Yang (23205838)
kai.y.lee@ucdconnect.ie

December 6, 2023

1 Introduction

The aim of this experiment is to discover the difference between serial and parallel algorithm for computing matrix multiplication. The parallel algorithm used in this experiemnt was imlemented using **Message Passing Interface (MPI)** and the benchmark was ran on a machine with 8 x Intel(R) Xeon(R) E5-2620 v3 @ 2.40GHz processors. Hence, all the benchmark for the parallel algorithm is ran with 8 processes meaning they are ran with `mpirun -np 8`.

All the matrix multiplication algorithms used in this experiment were manually written without using any third-party dependencies both for the serial and the parallel algorithm.

2 Dependence of program execution time on matrix size

Since the matrix size n must be divisible by the number of processes, the benchmark was ran with n ranging from 256 to 4096 with a step of 256. Figures below show the execution time for the serial and parallel algorithm with increasing matrix size n.

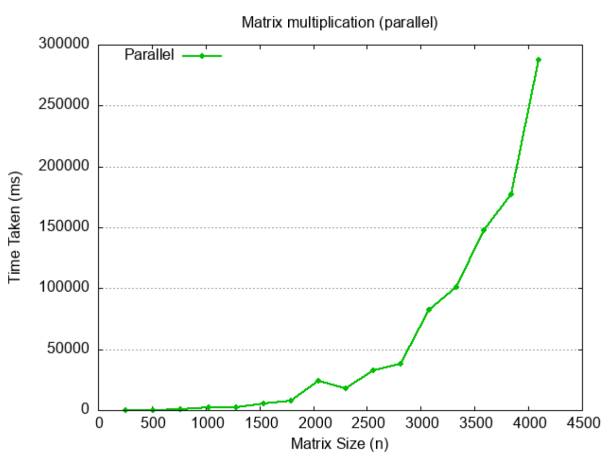

Figure 1: Matrix multiplication with parallel algorithm

From the results above, we can deduce that as matrix size n increases, execution time increases. This is true for both cases because the number of computation needed increases.

3 Speedup of parallel algorithm over serial algorithm

To further investigate on the speedup of the parallel algorithm over the serial algorithm, Figure 3 below shows the plot of the execution time for the serial algorithm and the parallel algorithm on the same graph. Moreover, Table 1 below includes the execution time for both algorithms for different matrix size n.

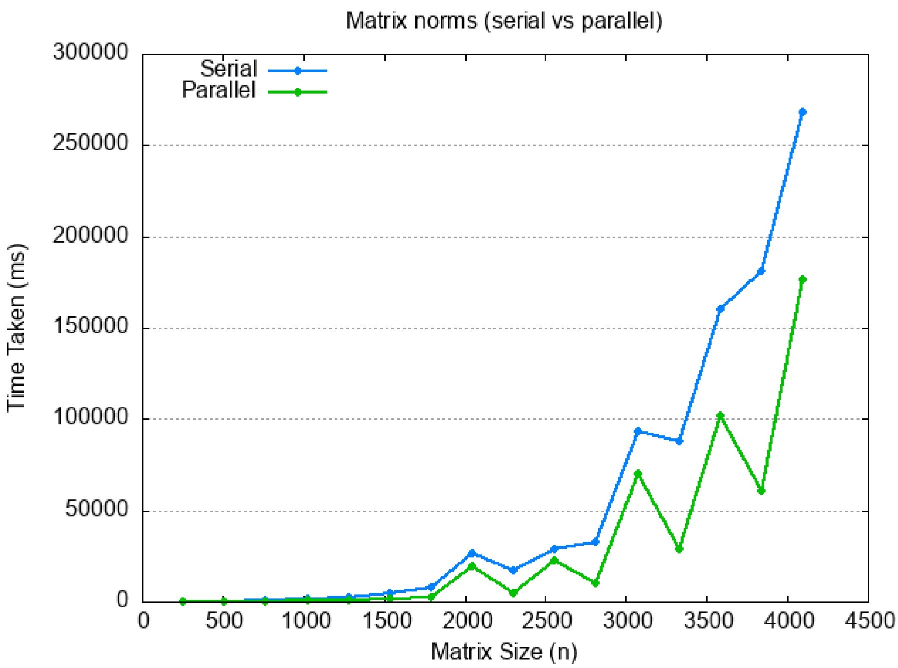

Figure 2: Matrix norm benchmark

Matrix size (n)	Time taken for serial (ms)	Time taken for parallel (ms)
256	48.62	8.54
512	173.62	45.76
758	566.35	107.89
1024	1407.65	948.70
1280	2632.17	528.31
1536	4889.38	1753.68
1792	7963.17	1956.31
2048	26496.56	19573.49
2304	17474.20	4640.64
2560	28872.98	22779.53
2816	32588.44	10419.45
3072	93145.27	70182.88
3328	87606.52	28659.00
3584	160394.68	101793.75
3840	181057.62	60536.71
4096	268178.54	176223.93

Table 1: Time taken for serial and parallel algorithm

From the results shown above, the average time taken for the serial algorithm is $57093.49ms$ and $31259.91ms$ for the parallel algorithm. The speedup is **45.25%** calculated using the formula:

$$\frac{|avg\ time_{parallel} - avg\ time_{serial}|}{avg\ time_{serial}} \cdot 100\%$$

.

That might not seem like a very bizarre speedup from the percentage but it still makes a big difference with an average of extra **25.83s** of overhead while using the serial algorithm. That said, the speedup depends heavily on the matrix size n. For example, when $n = 256$ the parallel algorithm is faster than the serial algorithm by **6** times. However, when $n = 2048$ the parallel algorithm is only faster than the serial algorithm by **1.35** times.

4 Increasing number of threads

To investigate out how different number of threads affect the execution time, I ran the parallel algorithms with different number of threads and plotted the combined graph below. Note that to change the number of threads with OpenMP, just run the binary with an extra `OMP_NUM_THREADS` environment variable.

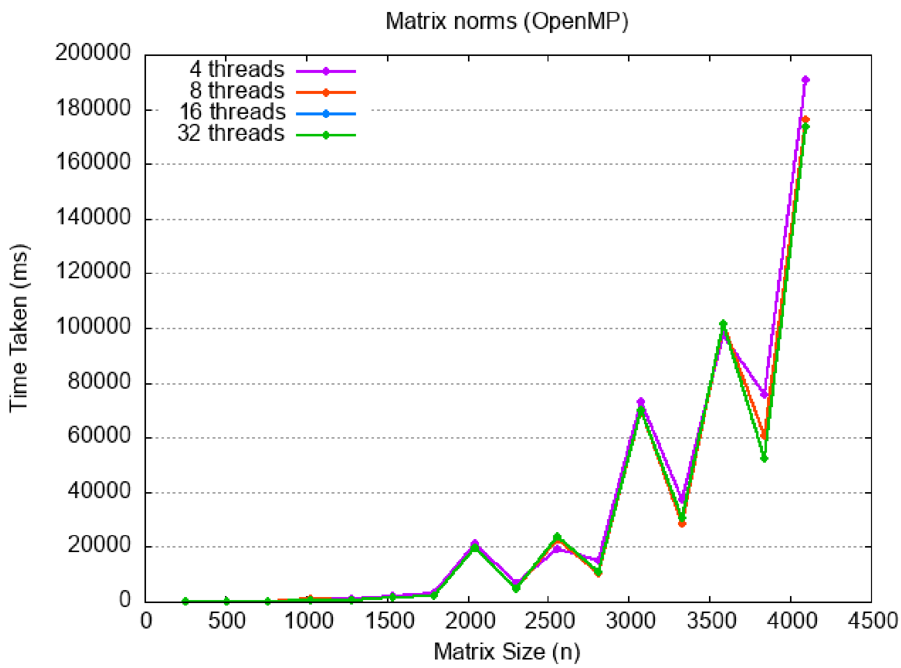

Figure 3: Matrix norm benchmark with different number of threads

From the figure above, we can see that there is still justifiable speedup from increasing the number of threads from 4 to 8 but after 8 threads, the speedup in execution time is basically negligible. One possible cause for this is the bottleneck of thread synchronisation using the `#pragma omp critical` macro in the calculation of matrix norm. The code for this can be found in the source `include/parallel.h` line 95. The reason being this region is mutual exclusive and only accessible by one thread at a time hence no matter how many number of threads we increase, the extra threads will be blocked here waiting for their turn to enter the critical region.