Analysis of Speed-UP Matrix Multiplication using MP

Imad Eddine TIBERMACINE

Abstract:

Matrix multiplication is a concept used in engineering fields such as: image processing, signal processing, graphs….etc. The complexity of matrix multiplication is O(n^3), because of that, huge matrices require a huge computation time. We , as problem solvers, our principal role is to optimise the execution time of those algorithms using different sequential and parallel algorithms. In this research, we used the open MP method of parallel computing to evaluate the execution time under different cases.

1. Introduction:

Parallel programming inside one SMP hub can take advantage of the universally shared address space. Compilers for shared memory structures more often than not back multi-threaded execution of a program. Circle level parallelism can be abused by utilizing compiler mandates such as those characterized within the OpenMP standard (Dongarra et al, 1994; Alpatov et al, 1997).

OpenMP gives a fork-and-join execution show in which a program starts execution as a single string. This string executes consecutively until a parallelization mandate for a parallel locale is found (Alpatov et al, 1997; Anderson et al, 1987). At this time, the string makes a group of strings and gets to be the ace string of the unused group (Chtchelkanova et al, 1995; Barnett et al, 1994; Choi et al, 1992).

All threads execute the statements until the end of the parallel region. Work-sharing directives are provided to divide the execution of the enclosed code region among the threads. All threads need to synchronize at the end of parallel constructs. The advantage of OpenMP (web ref.) is that an existing code can be easily parallelized by placing OpenMP directives around time consuming loops which do not contain data dependencies, leaving the source code unchanged. The disadvantage is that it is not easy for the user to optimize workflow and memory access.

On an SMP cluster the message passing programming paradigm can be employed within and across several nodes.

2. IBN-BADIS Cluster Hardware:

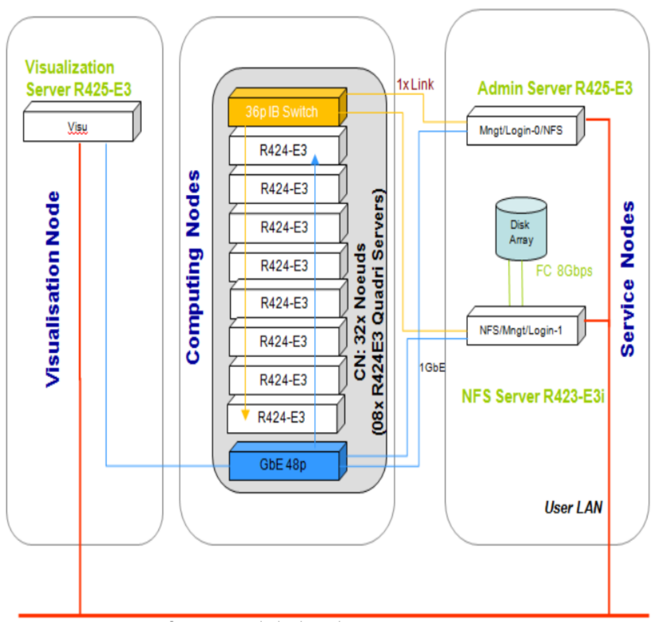
The experiments were done on Cluster of IBN-Badis (HPC of the CERIST researches center), its composed from 32 nodes, each node contains x2 processors Intel(R) Xeon(R) CPU E5-2650 2.00GHz, each processor contains 8 cores which makes 512 cores in total. The theoretical power of the cluster is around 8TFLOPS.

3. Cluster Architecture:

IBNBADIS consists of an ibnbadis0 administration node, an ibm badis10 viewer node and 32 ibnbadis11-ibnbadis42 computing nodes.

The ibnbadis10 visualization node is equipped with an Nvidia Quadro 4000 GPU (6GB, 448 cores) which can be used for calculations. Its equipped with:

- SLURM for job management
- C/C++, Fortran
- MPI and MP



4. Testing Nodes:

According to the following figure, the only nodes 38 and 39 were available at testing time, so all the tests in this sheet have been computed on the node 38 of the cluster.

```
[atibermacine@ibnbadis0 ~]$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
visu         up   infinite      1  alloc ibnbadis10
r424*        up   infinite      6 drain* ibnbadis[12,16,19,22,37,40]
r424*        up   infinite      7  down* ibnbadis[13,18,24-25,31,35,42]
r424*        up   infinite     17  alloc ibnbadis[11,14-15,17,20-21,23,26-30,32-34,36,41]
r424*        up   infinite      2   idle ibnbadis[38-39]
[atibermacine@ibnbadis0 ~]$
```

5. Steps of the analysis:

Master thread forks the outer loop between the slave threads, thus each of these threads implements matrix multiplication using a part of rows from the first matrix, when the threads multiplication are done the master thread joins the total result of matrix multiplication.

6. Sequential algorithm results before using openMP:

Before using the MP, we used the sequential naive algorithm to compare the results with the parallel method later, the naive sequential algorithms is as follows:

```
for i=1 to n
        for j=1 to n
                c(i,j)=0
                for k=1 to n
                        c(i,j)=c(i,j)+a(i,k)*b(k,j)
                end
        end
end
```

I implemented this algorithm using C language and i run it on the cluster with different matrix sizes, the results of this experiments are in the following table:

Matrix Size	Execution Time in seconds
50*50	0.000001
100*100	0.000001
300*300	1.000000
1000*1000	11.0000
2000*2000	109.0000
5000*5000	2256.0000
10000*10000	25320.00000

Tab.1: Execution time of the sequential algorithm for matrix multiplication with dynamic size

I tried to design the plot that defines the relation between different matrix sizes and the execution time:

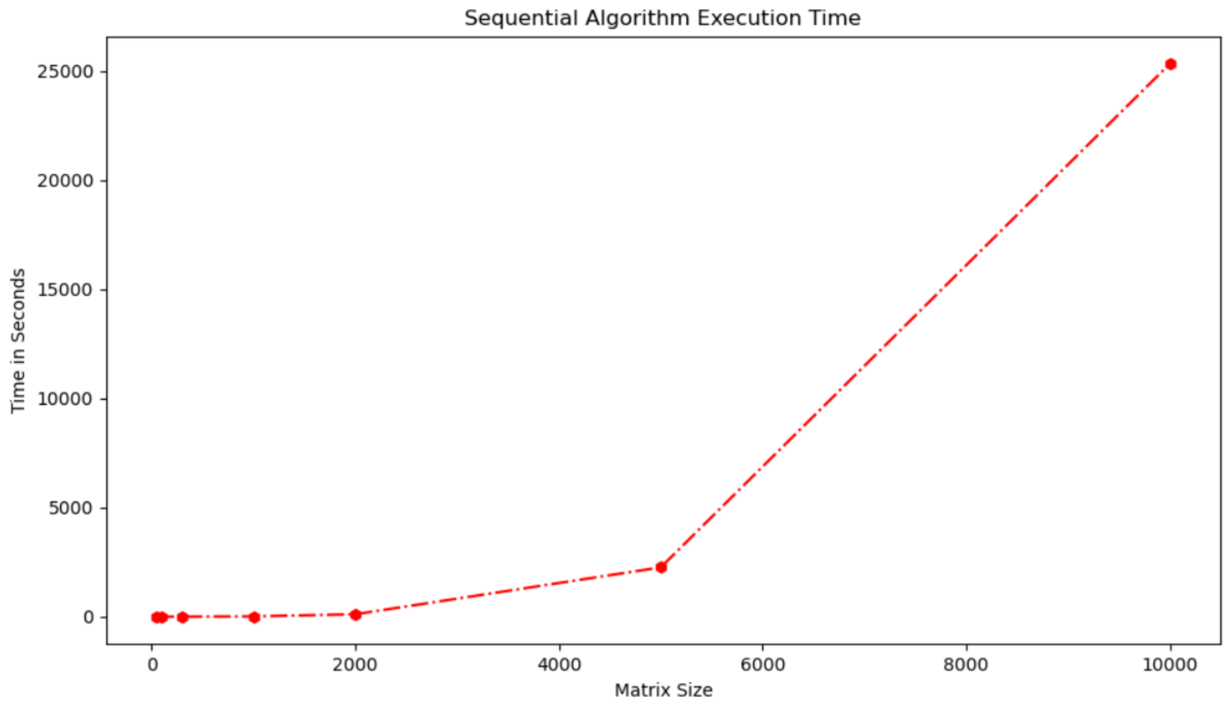


Fig.1: Execution time of different matrix sizes using sequential algorithm

7. MP parallel algorithm results:

The following result show the execution time of different matrices sizes with different number of Threads:

Threads	1000*1000	2000*2000	5000*5000	10000*10000
1	11.473568	105.182661	2402.201203	21379.59070
2	5.762523	52.618817	1178.661500	10772.96611
3	3.830655	35.268822	786.494686	7185.415451
4	2.881317	26.215736	580.940709	5245.894602
5	2.303854	20.947459	454.559860	4191.041909
6	1.926191	17.419138	384.962949	3572.071203
7	1.647273	14.884884	327.616296	2968.203605
8	1.442446	13.326681	288.256110	2640.425967
10	1.153247	10.555159	225.385298	2060.021623
12	0.972328	8.775925	194.371441	1776.173688
14	0.846106	7.523306	163.820344	1520.100034

Tab.2: Execution time of the parallel MP algorithm for matrix multiplication using different sizes with different number of Threads

The figure below defines the graph of the previous results:

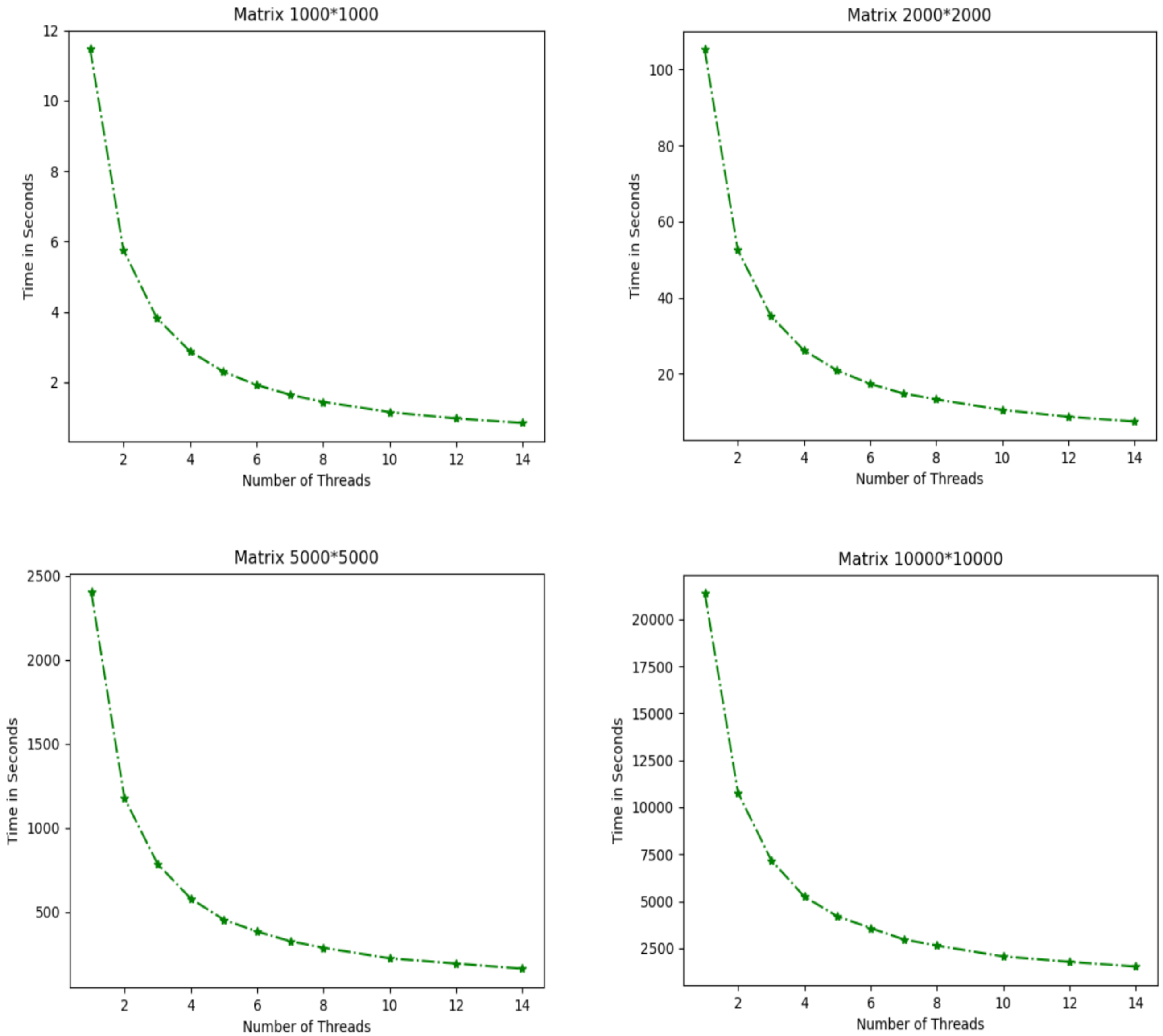


Fig.2: The execution time of different matrix sizes according to the number of Treads

8. Speed UP and Efficiency:

Using the previous results, in this section I tried to calculate both of speed up and efficiency, and plot their graph. We can calculate the speed up and Efficiency using the following rules:

SpeedUP (s)= Execution time with 1 Thread/ Parallel Time

Efficiency = SpeedUP / Number of Threads

Thread	SpeedUP 1000*1000	Efficiency	SpeedUP 2000*2000	Efficiency
1	1	1	1	1
2	1.991066	0.995533	1.998955	0.999477
3	2.995197	0.998399	2.982312	0.994104
4	3.982056	0.995514	4.012195	1.003048
5	4.980162	0.996032	5.021261	1.004252
6	5.956609	0.992768	6.038339	1.006389
7	6.965189	0.995027	7.066407	1.009486
8	7.954244	0.994280	7.892637	0.986579
10	9.948925	0.99489	9.965047	0.996504
12	11.800100	0.983341	11.985364	0.998780
14	13.560438	0.968602	13.980909	0.998636

Tab.3.1: Speed up and efficiency calculated for the previous results

Thread	SpeedUP 5000*5000	Efficiency	SpeedUP 10000*10000	Efficiency
1	1	1	1	1
2	2.038075	1.019037	1.984559	0.992279
3	3.054313	1.018104	2.975414	0.991804
4	4.135019	1.033754	4.075489	1.018872
5	5.284675	1.056935	5.101259	1.020251
6	6.240084	1.040014	5.985208	0.997534
7	7.332361	1.047480	7.202872	1.028981
8	8.333565	1.041695	8.097023	1.012127
10	10.658198	1.065819	10.378333	1.037833
12	12.358817	1.029901	12.036880	1.003073
14	14.663631	1.047402	14.064594	1.004613

Tab.3.2: Speed up and efficiency calculated for the previous results

The figure below defines the graph of the speedUP and Efficiency changes during the tests:

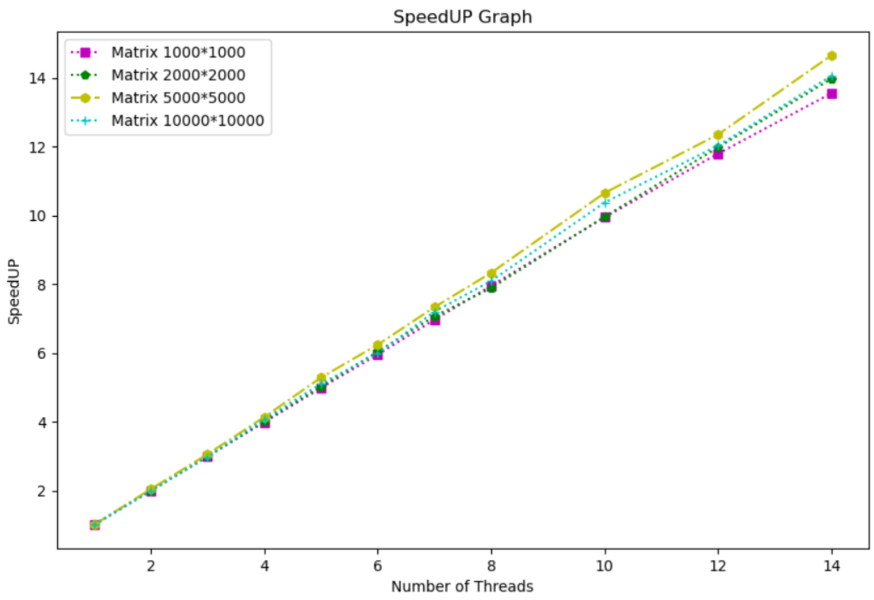


Fig.3: Graph speedUP of the experiments

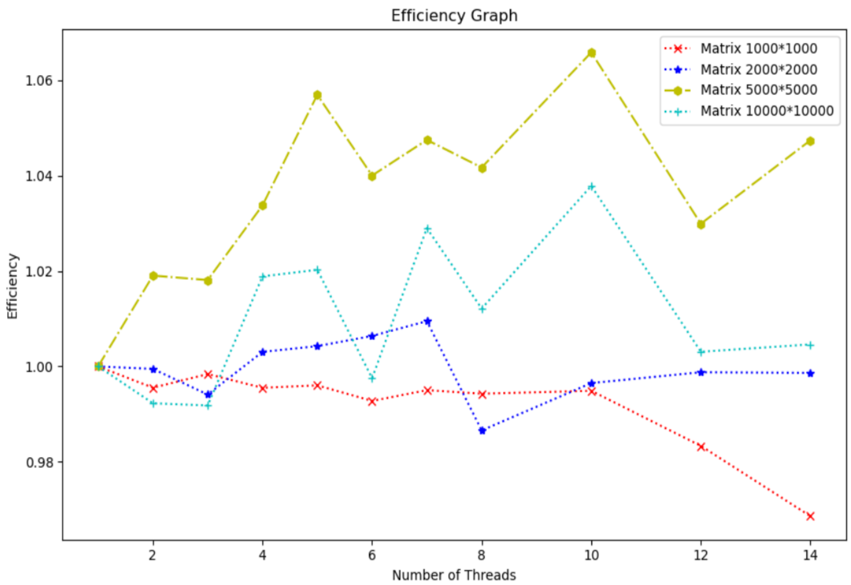


Fig.4: Graphs of Efficiency of the experiments

9. Conclusion:

Based on the previous obtained results, and the plots shown above , the conclusion can be drawn is that MP is a good method to use as an environment for parallel matrix multiplication with huge sizes, here we can increase the speedup but negatively affect the system efficiency. The second thing I can suggest is to use a hybrid parallel system to manipulate matrices of huge sizes.