HPC Project: Optimize matrix-matrix multiplication

Arthur Picault Brieuc Horard

June 7, 2020

All of our code and results can be found in at: https://github.com/AeroPyk/HPC_project

Contents

1 Introduction

In High Performance Computing one of the main challenges is the matrix matrix multiplication. Particularly at large scale, it gets very slow to perform multiplication the naive way. Say we want to perform:

$$C = A * B$$

with A, B and C being matrices of size [i,j], [j,i] and [i,i].

A first step is to trick the cache memory by arranging the loops: (i,k,j) instead of (i,j,k). A step further is to use threads to parallel the upper loop with framework like OpenMP. However, how could we get one step even further ?

One solution is provided by the Matrix algorithms on a hyper-cube I: Matrix multiplication by G.C Fox and S.W Otto and A.J.G Hey [1]. This paper describe an algorithm to exploit several processes at once to parallel the multiplication.

Based on this paper we implemented an algorithm that calculates a sub result of the final result per process. To do so we virtually separated the matrix A and B into equally sized sub-matrices mapped to a squared number of processes. Each process mapped to a sub-matrix (see Fig.1) would then calculate the corresponding sub-matrix of C.

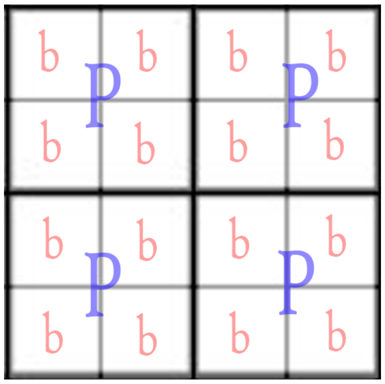


Figure 1: Mapping sub-matrices (b's) to processes (P)

In this paper 2 main steps are described in the implementation, the first step implementing the algorithm with square sub-matrices and the second step extending this algorithm to rectangular sub-matrices.

We chose to focus on implementing the algorithm using square sub-matrices.

2 Methodology

We worked in the following conditions:

- Matrices are square
- Tile size depends on the number of processes
- Number of processes must be square

2.1 Topology

The very first goal was to set up the mesh topology. That said, from the world communicator, we made up a square topology depending on the number of processes detected with

```
MPI_Comm_size(MPI_COMM_WORLD, &wsize);
```

If the number of processes isn't square, then the program ends with a warning. Along the way we made a few debugging function mainly to print the topology to ensure we had the topology we were aiming for, reusing some MPI knowledge.

Sub topology were then created according to each line and each column making several more topologies helped by the documentation of mpich [2].

2.2 Read / write A and B

The next move was to figure out how to load A and B since the paper didn't mention anything about it. Therefore we decided to load from files on disk. This provide a common source to ensure every process works on the very same matrices.

The file format would be as:

```
row column value value value ... [row*column times]
```

Instead of writing these A and B matrices by hand, at each beginning of the program, one process (and only one!) write A and B with random double values before a barrier instruction. Indeed writing a same file by several processes can't be of any good. This provide a very good efficiency when it comes to test several size / number of processes.

At this point any process can write and read a matrix from file. But wouldn't it better for each one to know which part of each matrix one process should load instead of a whole matrix? Therefore with some math we provided a formula that allows us to extract the sub-matrix from the file.

Another function can extract the sub matrix from the mere matrix, however that supposes it is already loaded therefore slower.

2.3 Matrices

When it comes to matrices in C we always wonder at first whether we should use dynamic allocation or just simple 2D arrays. After a shy start with static 2D arrays, we realized that dynamic memory allocation was more suited to our project whether talking about passing it through functions or between processes.

Our allocation is such that our memory is allocated consecutively [3] to avoid any struggles passing it with MPI. However to keep the old fashion way of programming with 2D array, an allocation is made along of point referring to the start of each line letting use the A[i][j] notation. With such allocation an appropriate free function had to be made.

This way our matrices are all set for the Intel multiplication process.

2.4 The algorithm itself

The explanation of the algorithm is well described in the original paper [1].

Three steps occur:

2.4.1 A broadcast

By turns the diagonal of A + an offset is broadcasted across line sub-topologies. This is mainly done using

```
MPI_Bcast()
```

called by every single process, only the root changing according to the turn. However we need to be careful and not erase the original sub-matrix used by the process to let him broadcast it when it's its turn. That's why we have a back up of sub matrix for A. The full details of the function have been found on [4].

2.4.2 C multiply

Once A broadcasted, B being already in place from the loading step, we can perform the multiplication.

2.4.3 B rolled

Once the multiplication done, B is "rolled" in such way that each process send its sub-B to the upper process in the column sub-topology receive. Here we need to be clear with the direction we give to

```
MPI_Cart_shift()
```

Followed by

```
MPI_Sendrecv_replace()
```

Then we get on the next turn.

2.5 Multiplication within a process

At first, only the cache optimized version of matrix multiplication was used. Once done with the main algorithm, enabling the multi-threading with MPI_THREAD_FUNNELED allowed us to use OpenMP within a process to optimize the upper loop of the matrix multiplication.

We also wanted to use the Intel multiplication method (DGEMM - BLAS).

2.6 Matching back

To gather the results (i.e. each C sub matrix) from all processes we use

```
MPI_Gather()
```

which will unfortunately gather the sub-matrices as lines [5]. We therefore made a function that map each line to the resulting matrix. This is done by one process.

Finally we write this full C matrix into a file. This file can be used for further matrix multiplication since it has the same format than A and B.

2.7 File structure

The project fit into 3 files: main.c, ext.h, ext.c with this last holding all the functions made.

2.8 Timers

As in our paper we didn't take in account the loading time and the set up like the topologies, but only the multiplication time.

2.9 Validity

Since the multiplication is performed in 2 ways (naive - 1 process - and multiprocess) we compare the 2 results with a function equalMat() which answer "Va tutto bene" if both matrices are equal (with an error lower than 1e-8 - since we use doubles).

3 Experimental Setup

3.1 Laptop

Item	Value
Model	Surface Pro 4
Processor	Intel(R) Core(TM) i7-6650U CPU 2.20GHz, 2 Cores, 4 Logical Processors
Physical Memory	16.0GB
Compiler	TDM-GCC-64 (MinGW)
Flags	-fopen ; target_link_libraries ... (CMake file)
IDE	CLion
MPI version	Microsoft MPI
Execution	mpiexec.exe -n 16 ”main.exe”

3.2 Beskow

We experimented with different setup for several matrix size.

3.2.1 Compilation

We've discovered that we could reuse the CMake file used in CLion (IDE) with the cmake command on Beskow. We therefore used it and then used *make* to compile the file from project.

Therefore, *-fopen* is one of the flags applied and the MPI library is linked and used the default compiler, Cray.

3.2.2 Nodes

We used different amount of nodes / process to test our algorithm always being careful to get square sub-matrices. We used exclusively Haswell processors.

3.2.3 Queuing

We were used to allocate manually nodes and then run our program with srun. However here we chose to explore another way of asking Beskow to run our program.

We used the *sbatch* command to queue the job we were giving thanks to a script that precise the way it should run the algorithm (see file queuing.sh in repository).

We think this is a better and more efficient way to run our code by scheduling them.

4 Results

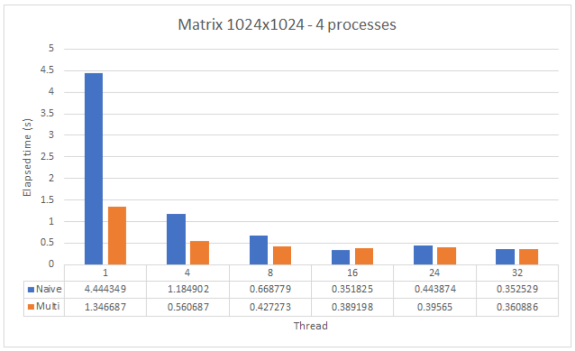
We've always been doing the calculation with the two versions: Naive being with one process and Multi referring to the method with several processes.

4.1 Laptop

Results aren't very much relevant as on laptop processes don't quite run parallel. Therefore it always took longer with several processes than using the naive way. This is due to the overhead generated by the use of several processes.

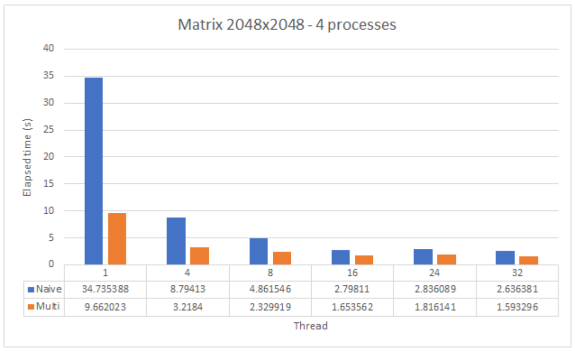
4.2 Beskow

4.2.1 4 processes over 4 nodes



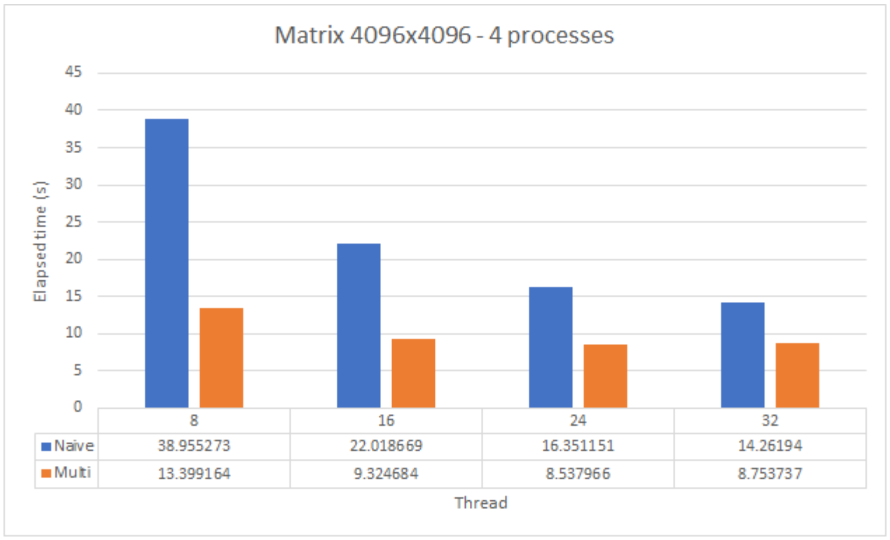
	1	4	8	16	24	32
🟦 Naive	4.444349	1.184902	0.668779	0.351825	0.443874	0.352529
🟧 Multi	1.346687	0.560687	0.427273	0.389198	0.39565	0.360886

(a) 1024*1024



	1	4	8	16	24	32
🟦 Naive	34.735388	8.79413	4.861546	2.79811	2.836089	2.636381
🟧 Multi	9.662023	3.2184	2.329919	1.653562	1.816141	1.593296

(b) 2048*2048



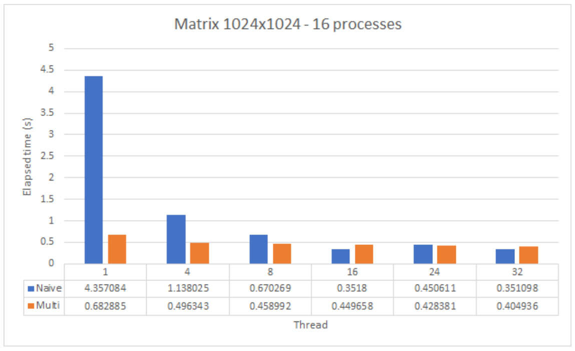
	8	16	24	32
🟦 Naive	38.955273	22.018669	16.351151	14.26194
🟧 Multi	13.399164	9.324684	8.537966	8.753737

(c) 4096*4096

Figure 2: Multiplication performance with 4 processes

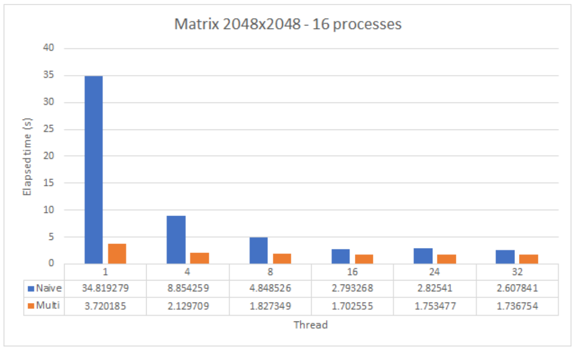
Overall, the multiprocessing (orange) version is more efficient. However on small matrices it tends to be as fast as the naive version most likely because of the overhead generated by the use of several processes.

4.2.2 16 processes over 16 nodes



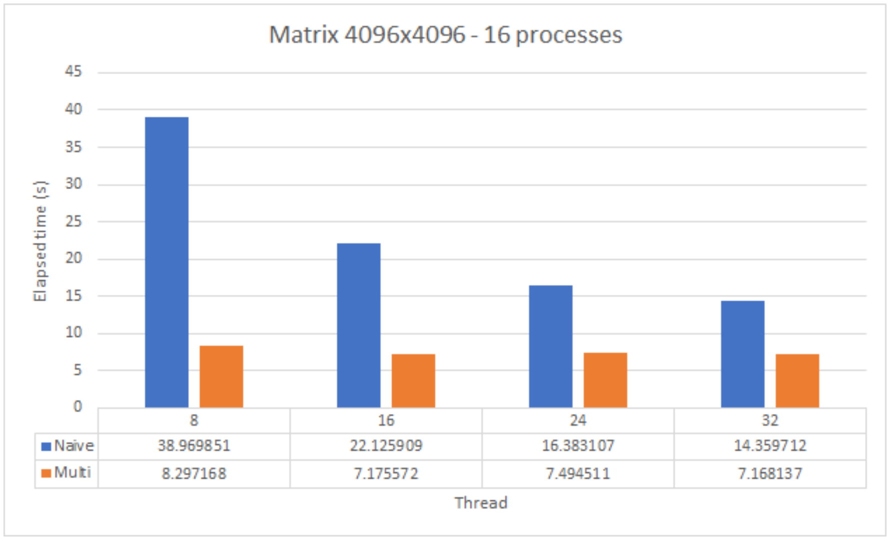
	1	4	8	16	24	32
■ Naive	4.357084	1.138025	0.670269	0.3518	0.450611	0.351098
■ Multi	0.682885	0.496343	0.458992	0.449658	0.428381	0.404936

(a) 1024*1024



	1	4	8	16	24	32
■ Naive	34.819279	8.854259	4.848526	2.793268	2.82541	2.607841
■ Multi	3.720185	2.129709	1.827349	1.702555	1.753477	1.736754

(b) 2048*2048



	8	16	24	32
■ Naive	38.969851	22.125909	16.383107	14.359712
■ Multi	8.297168	7.175572	7.494511	7.168137

(c) 4096*4096

Figure 3: Multiplication performance with 16 processes

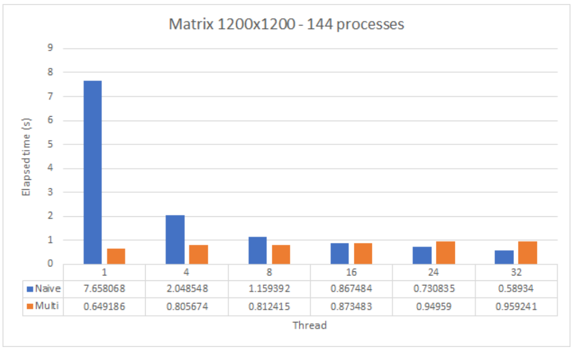
This overhead is confirmed here if we look at the 32 threads version of 1024*1024 operation. Naive version is faster but applied to larger matrices, multi version is obviously more effective.

Further more, multi-threading is always improving the time computation as more threads are used. This for both versions.

On the larger matrices we see an improvement as we use more processes, as expected.

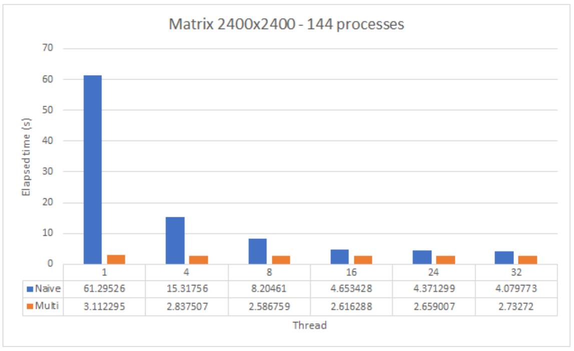
4.2.3 144 processes over 36 nodes

For our 4200*4200 matrix only with 8 thread has been executed (the job ran out of time) getting a Naive time of 46.643537 s and a Multi time of 8.757849 s.



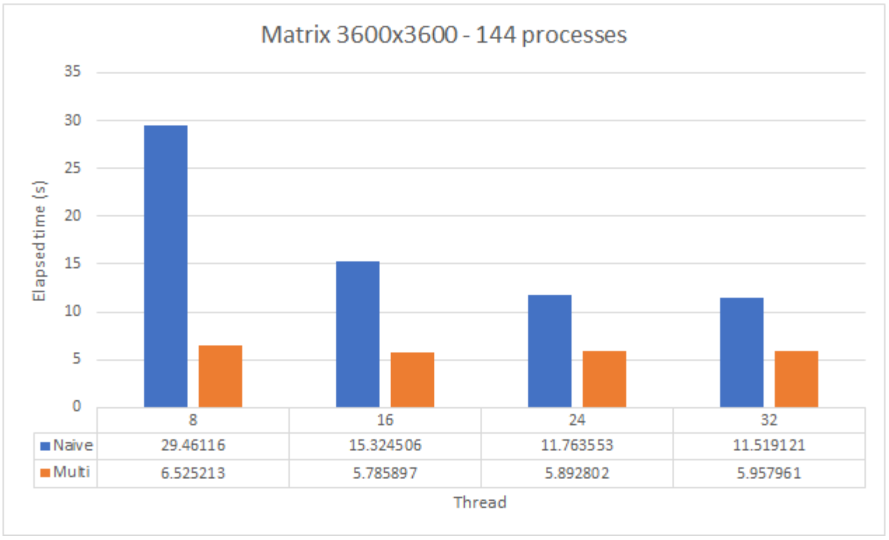
Thread	1	4	8	16	24	32
Naive	7.658068	2.048548	1.159392	0.867484	0.730835	0.58934
Multi	0.649186	0.805674	0.812415	0.873483	0.94959	0.959241

(a) 1200*1200



Thread	1	4	8	16	24	32
Naive	61.29526	15.31756	8.20461	4.653428	4.371299	4.079773
Multi	3.112295	2.837507	2.586759	2.616288	2.659007	2.73272

(b) 2400*2400



Thread	8	16	24	32
Naive	29.46116	15.324506	11.763553	11.519121
Multi	6.525213	5.785897	5.892802	5.957961

(c) 3600*3600

Figure 4: Multiplication performance with 144 processes

With these last results we suspect that at high amount of processes, sub-matrices are small enough to let the overhead being more a waste of time than the actual gain with several threads.

5 Discussion and Conclusion

Before running our program on Beskow we were doubting slightly about this algorithm and also wondered whether we had made mistakes in our code. Then we ran it on one node on Beskow. Still, the multi version was slower but of course when using more nodes this changed radically. The naive version became largely outperformed.

We intended to use Intel MKL multiplication process as well (DGEMM - Double precision GEneral Matrix Multiply) and compare. However with the end of the semester we ran out of time and had to move on to the summer job. However, we took time to analyse how it would work and it appears that our matrices format would have perfectly fit in the callable function:

```
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, subsideMat, subsideMat, subsideMat,
    1.0, &(A[0][0]), subsideMat, &(B[0][0]), subsideMat, 1.0, &(C[0][0]), subsideMat);
```

That's the way we would have call the function.

Indeed for passing matrices through MPI, it is way easier to allocate contiguous memory and so it is for *cblas_dgemm*.

We think this project was perfect to end our class over the subject. We learned how to put together MPI and OMP, used Beskow yet another way and overcome a classic performance optimizing matrix-matrix multiplication. For that we're aiming for a B if it's not an A.

6 Further optimizations

Our algorithm isn't very much flexible in terms of amount of processes versus matrix size. That's why we would be eager to implement the second part of our base paper in which we would allow multiplication between non square sub matrices.

References

[1] G.C. FOX, S.W. OTTO, and A.J.G. HEY. Parallel computing 4 - matrix algorithms on a hypercube i: Matrix multiplication *. pages p. 17 – 31, 1987.

[2] Mpi documentation. `https://www.mpich.org/static/docs/latest/www3/`, 2019.

[3] Sending and receiving 2d array over mpi. `https://stackoverflow.com/questions/5901476/sending-and-receiving-2d-array-over-mpi`, 2013.

[4] Mpi_bcast. `https://mpi.deino.net/mpi_functions/MPI_Bcast.html`, 2009.

[5] Wes Kendall. Mpi scatter, gather, and allgather. `https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/`.