Parallelizing `AX=B`

Source code from this homework can be find in the archive directory, along with a README file.

Introduction

The goal of this project is to study the parallelization of the resolution of `AX=B` equation (where $A \in M_N(\mathbb{R})$). This equation has been resolved using Gaussian elimination algorithm without pivoting. Several solutions have been implemented, using two parallelization methods: MPI and multithreading (using `pthreads` library). Those solutions have been compared to a traditional resolution.

About Gaussian elimination

Gaussian elimination algorithm hands in two steps: elimination and back substitution. The first step involves $O(N^3)$ operations when the second only involves $O(N^2)$ operations. That is why the parallelization only aims to accelerate the first step.

This step is a triple `for` loop which consists in diagonalizing matrix `A`. To achieve this, the algorithm operates column after column: working on smaller and smaller sub-matrices of `A`, it puts zeros in the lower part of it.

Thus, we can notice three important things about this algorithm:

(1) Iterations of the first for loop will work on the current sub-matrix but not the previous computed rows
(2) Iterations of the second for loop will only work on the current row of the sub-matrix based on itself and its first row
(3) Iterations of the third for loop will only work on the current cell of the row based on itself and the first row

These three points will have implications on the parallelizing methods and will limit them.

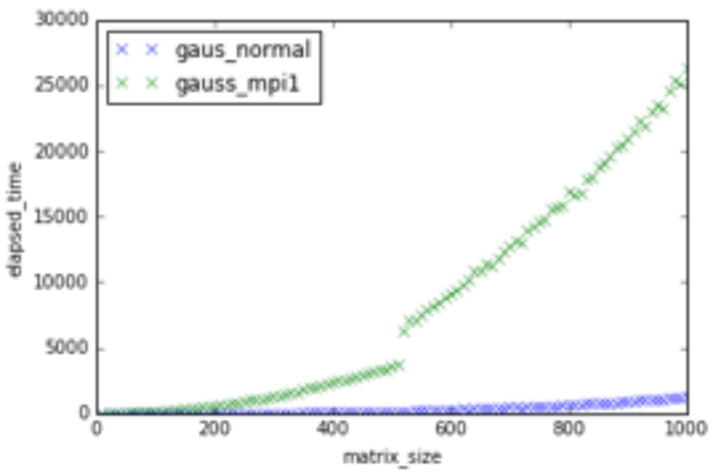
First method: using MPI

The first parallelization was done using MPI. MPI consists in sharing the work among several processes. In this implementation I decided to provide every process the whole matrices. The algorithm is as follows (given a world of `n` processes):

```
process 0 broadcasts A and B to the world
for each sub-matrix of A
    process i computes rows i, i+n, i+2n, ... of A and B
    process i broadcasts rows i, i+n, i+2n, ... of A and B to the world
    process i waits until the world stopped broadcasting
```

This method has one major problem: the gain of shared work is ruined by the heavy messages. If we exclude the the broadcasts from process `0`, each process will make $O(N^3/n)$. However, there will be `O(N2/n)` messages broadcasted (≡ sent to `n` processes) containing each time $O(N)$ floats. Thus, as long as the work given to a process will not exceed the length of messages, the gain will not be consequent.

We can observe in graph 1 that this method is slower than the traditional one.



Graph 1

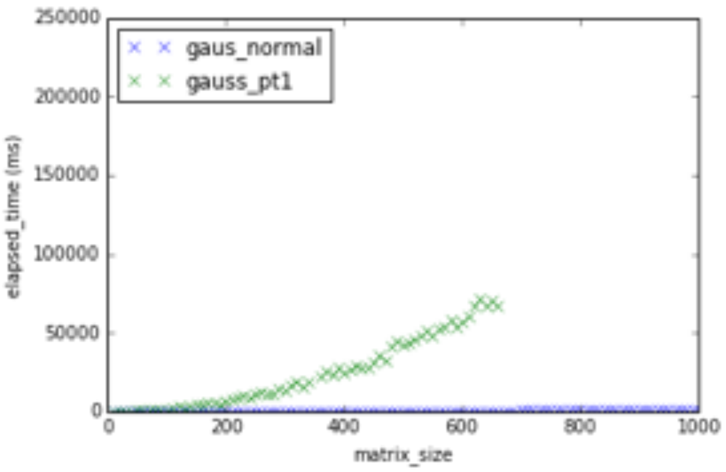
In this algorithm, there is a self-dependance in the calculation of each row. At each sub-matrix iteration, a process needs to know the row how it has been computed at last.

I could not think of any significantly better solution using MPI. However, it would be possible to save a few messages shortening broadcasts (using `MPI_Send()` to send computed rows to the process that will need it).

Second method: using multi-threading

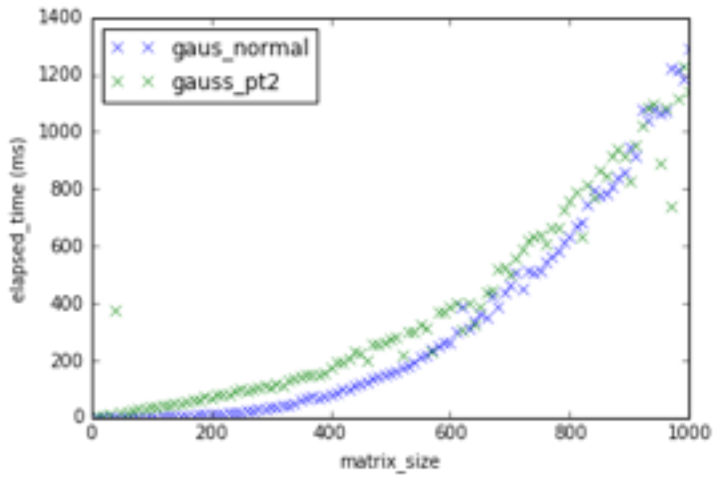
The main parallelization was done using `pthreads` library. With threads, compared to MPI, we save messages transmission, given that threads share memory. Three implementations have been implemented.

The first one consists in sharing the work in the third loop. It means that given a sub-matrix and a row, each thread will work on designated cells. It accelerated the computation of a new row, but implies the creation of too many threads, and finally slows the algorithm. We can see that in graph 2.



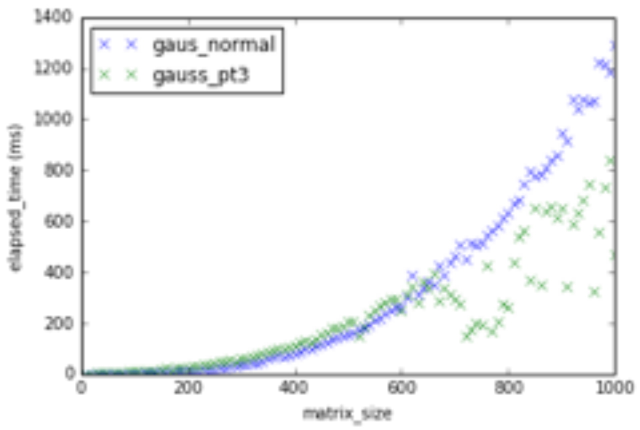
Graph 2

A direct improvement was to share the work not in the third but in the second loop: given a sub-matrix, each thread will work on designated cells. If we ignore the time of threads creation, it accelerates the algorithm, going from O(N^3) operations to O(N^2), as we see in graph 3. However, we see that this method does not seem to outrun the traditional one. That is because we cannot ignore the time needed to create and join threads.

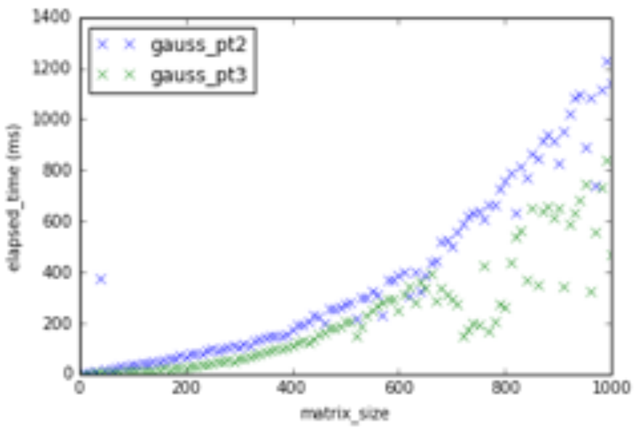


Graph 3

Thus, a final improvement of this algorithm was to process such as threads are only created once. The workload is then lightened, as we can see in graph 4 and 5 but it implied to overcome the rows self-dependance problem described in MPI part.

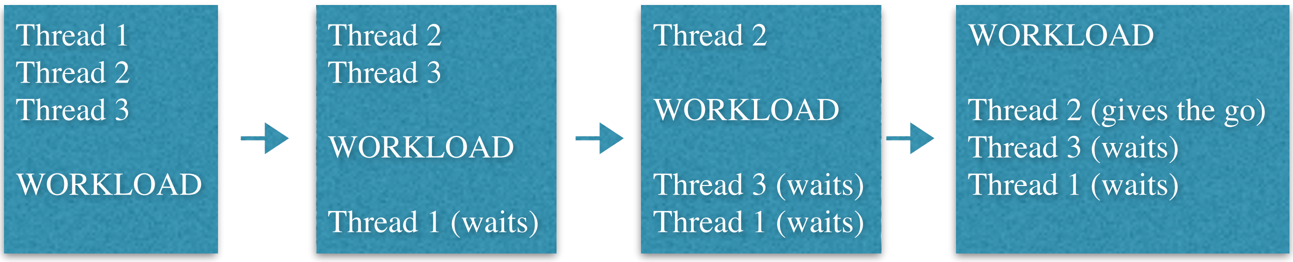


Graph 4



Graph 5

I had to simulated the limits naturally given by the first loop so that no thread will work on a sub-matrix if there is still a thread working on the previous one (as illustrated in the next illustration).



Threads behavior during one iteration of the first `for` loop

Conclusion

In conclusion, although both methods MPI and pthreads can be very powerful, in the case of Gaussian elimination MPI seems to be difficult to implement. The best way to implement this algorithm for now is with pthread library. However, in other algorithms, it may be much more profitable.