INFO 0939: High Performance Scientific Computing

Project 2

Teachers: Geuzaine Christophe

PAQUAY Joachim, BONHOMME Louis

1^{st} Master in engineering

Academic year 2019-2020

1 Introduction

Many physical problems are represented by partial differential equations. In such cases, the exact solution can be rarely found analytically and needs to be approximated by using different methods. In the framework of this project, a wave propagating at the surface of the ocean needs to be modelled using the so-called "shallow water" equations in two dimensions given by:

$$\begin{aligned} \frac{\partial \eta}{\partial t} &= -\nabla \cdot (h\mathbf{u}), \\ \frac{\partial \mathbf{u}}{\partial t} &= -g\nabla\eta - \gamma\mathbf{u}, \end{aligned} \tag{1}$$

where the unknowns fields $\eta(t,x,y)$ and $\mathbf{u}(t,x,y) = (u(t,x,y), v(t,x,y), 0)$ are respectively the free-surface elevation and the depth averaged velocity, and where

- $h(x,y)$ is the depth at rest;
- $g = 9.81\text{m/s}^2$ is the gravitational acceleration;
- γ is the dissipation coefficient.

To do so, finite differences method is used. The domain is discretized in a rectangle of dimensions $[a,0] \times [0,b]$ as shown in Fig.1

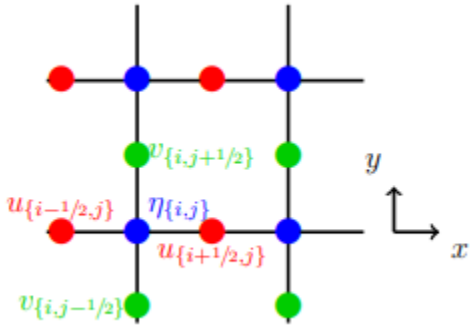

Figure 1: Spatial grid.

The solutions of the problem are found at these points of interest by using two different methods of approximation:

- The explicit Euler method given by:

$$\begin{aligned} \frac{\eta_{ij}^{n+1} - \eta_{ij}^{n}}{\Delta t} &= -\frac{h_{i+1/2,j}u_{i+1/2,j}^{n+1/2} - h_{i-1/2,j}u_{i-1/2,j}^{n+1/2}}{\Delta x} - \frac{h_{i,j+1/2}v_{i,j+1/2}^{n+1/2} - h_{i,j-1/2}v_{i,j-1/2}^{n+1/2}}{\Delta y}, \\ \frac{u_{i+1/2,j}^{n+1/2} - u_{i+1/2,j}^{n+1/2}}{\Delta t} &= -g\frac{\eta_{i+1,j}^{n} - \eta_{i,j}^{n}}{\Delta x} - \gamma u_{i+1/2,j}^{n+1/2}, \\ \frac{v_{i,j+1/2}^{n+1/2} - v_{i,j+1/2}^{n+1/2}}{\Delta t} &= -g\frac{\eta_{i,j+1}^{n} - \eta_{i,j}^{n}}{\Delta y} - \gamma v_{i,j+1/2}^{n+1/2}. \end{aligned} \tag{2}$$

- The implicit Euler method given by:

$$
\begin{aligned}
\frac{\eta_{ij}^{n+1}-\eta_{ij}^{n}}{\Delta t} &= -\frac{h_{i+1/2,j}u_{i+1/2,j}^{n+1+1/2}-h_{i-1/2,j}u_{i-1/2,j}^{n+1+1/2}}{\Delta x}-\frac{h_{i,j+1/2}v_{i,j+1/2}^{n+1+1/2}-h_{i,j-1/2}v_{i,j-1/2}^{n+1+1/2}}{\Delta y},\\
\frac{u_{i+1/2,j}^{n+1+1/2}-u_{i+1/2,j}^{n+1/2}}{\Delta t} &= -g\frac{\eta_{i+1,j}^{n+1}-\eta_{i,j}^{n+1}}{\Delta x}-\gamma u_{i+1/2,j}^{n+1+1/2},\\
\frac{v_{i,j+1/2}^{n+1+1/2}-v_{i,j+1/2}^{n+1/2}}{\Delta t} &= -g\frac{\eta_{i,j+1}^{n+1}-\eta_{i,j}^{n+1}}{\Delta y}-\gamma v_{i,j+1/2}^{n+1+1/2},
\end{aligned}
\tag{3}
$$

where every parameters are given in a "file.txt" and h is a matrix also given in a "file.dat". For both methods at each time step n, the solution is found by using the conjugate gradient method given by:

Algorithm 1: Conjugate gradient method.

$\mathbf{x}_0 = \mathbf{0}$
$\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$
$\mathbf{p}_0 = \mathbf{r}_0$
i = 0
while $\frac{\|\mathbf{r}_i\|_2}{\|\mathbf{r}_0\|_2} \geq r_{threshold}$ **do**
  $\alpha = \frac{\mathbf{r}_i^T\mathbf{r}_i}{\mathbf{p}_i^T A\mathbf{p}_i}$,
  $\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha\mathbf{p}_i$
  $\mathbf{r}_{i+1} = \mathbf{r}_i - \alpha\mathbf{A}\mathbf{p}_i$
  $\beta = \frac{\mathbf{r}_{i+1}^T\mathbf{r}_{i+1}}{\mathbf{r}_i^T\mathbf{r}_i}$
  $\mathbf{p}_{i+1} = \mathbf{r}_{i+1} + \beta\mathbf{p}_i$
  i = i + 1
end
return $\mathbf{x}_i$.

Figure 2: Conjugate gradient method.

The goal of this project is to analyze the stability, the scalability and the performance for the two methods of approximation.

2 Description of the implementation

2.1 Explicit method

Let us consider the discretized space domain as shown in Fig. 1. The particularity of the Euler explicit method is that the calculation for computing a variable at time $n+1$ needs only the values of the variables at time n which are known. As a consequence, the variables at time $n+1$ are all independent and can be computed separately. Moreover, the computation of a variable at time $n+1$ needs only the values of the variables at time n that surround it. This characteristic will be used in the implementation of the communication between the processes. Knowing that, the domain is divided into the different processes as shown in Fig.3.

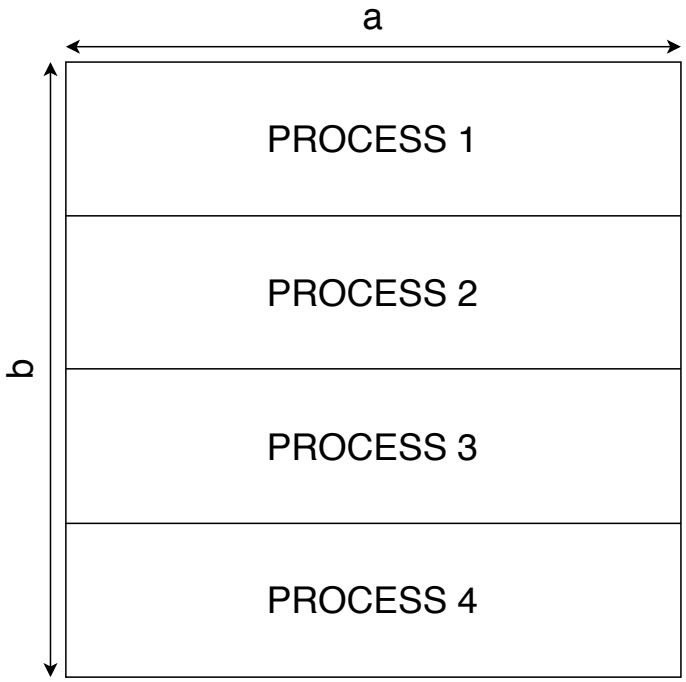

Figure 3: Distribution of the space between the different processes

This means that, if the variables η, u, and v are separated in three vectors, every process would have a portion of the three vectors. This also means that some communications between the processes will be required. Indeed every process needs to receive the values at time n at the edges from their neighbours in order to compute their next values at their edges at time $n+1$. Fig.4, represent this property.

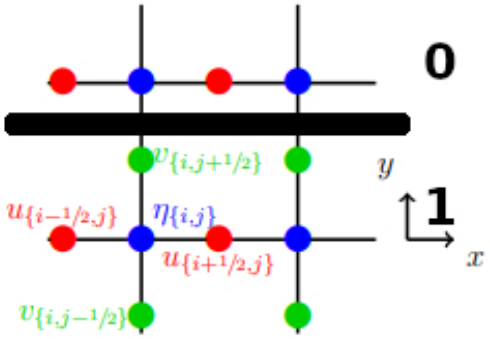

Figure 4: Spatial grid divided between two process, using *refraction.dat* map

On this figure, the process 0 have to communicate his values of η as well as the process 1 have to communicate his values of v that are the closest to the boundary. Then, every process has all the needed values in order to compute theirs. To do so, the process can be divided in several threads using *omp* because as there is no dependence for a value at a time $n+1$ on an other value at the same time, every operation can be done independently from the others. The final values for each time step are then sent to the process 0. Depending on S, the results at each time step or every two time steps. Then they are saved in a file.

2.2 Implicit method

At the beginning the barymetric map, h, was interpolated. Certain number of lines (depending of the parameters) were given to each process in order to divide the barymetric map and to interpolate it. Therefore one has two sub-matrix H_i and H_j, where the term "sub-matrix" stands for a matrix where the number of lines corresponds to the number of lines given to the process. Then a *Allgatherv* method was used to regroup all the H_i and the H_j on every process, where H_i corresponds to the h value of red dots and H_j corresponds to the h value of the green dots in figure 1. This can be represented in the Fig.5

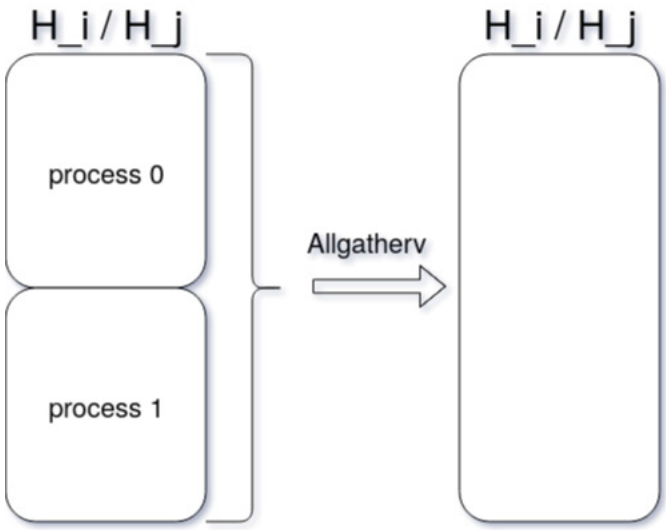

Figure 5: MPI on H_i and H_j.

After gathering all the H_i and the H_j, the time loop was started. By looking at the figure 2 one can better understand the following explanations.
The first step was to address a certain number of lines to x and therefore A, as one did for H_i and H_j but here the boundary conditions were removed from x and A and added to b, on a given process then a *Allgatherv* method was used in order to have the whole x on every process. This is represented in the Fig.6.
By doing so, one can compute the sub-vector Ax on each process which is represented in the Fig.7. It can be noticed that the sparse matrix A is not stored in the memory and the value of its coefficients can be found by giving a line and a row index. An other important detail is that the vector x, b,... corresponds to the concatenation of the three vectors eta, u and v.

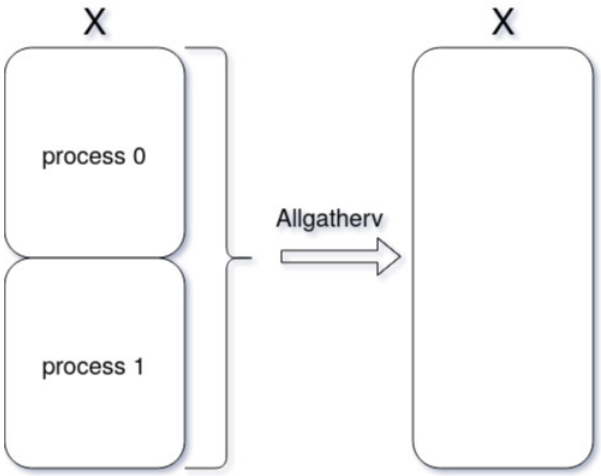

Figure 6: MPI on x

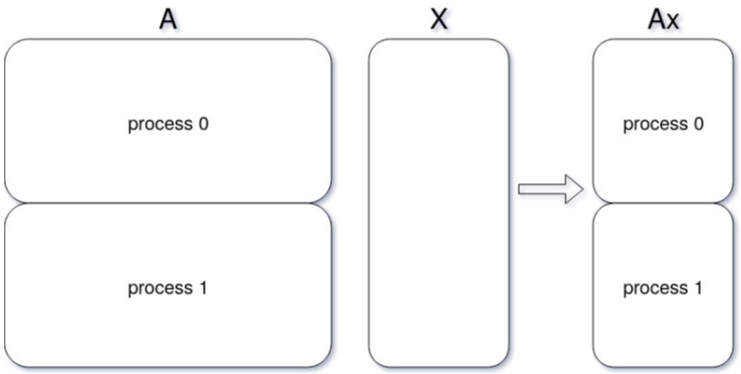

Figure 7: MPI on Ax

Then, because of the fact that the matrix A was not symmetric positive-definite one has to multiply b and Ax by A^t. A *Allgatherv* method was also used in order to gather all the Ax divided over the process, this is represented in the Fig.8. And then the sub-vector A^tAx on each process could be compute, this is represented in the Fig.9.

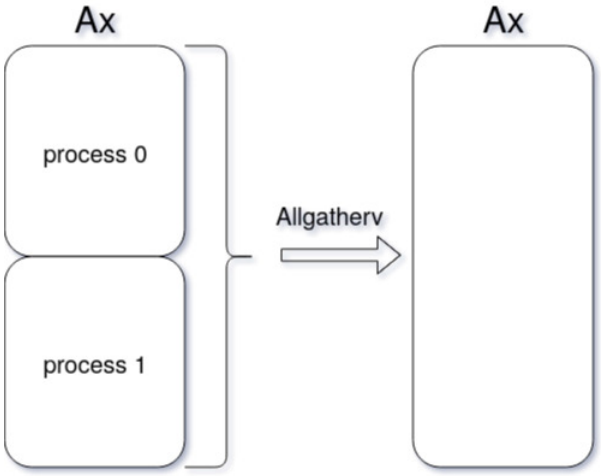

Figure 8: MPI on Ax

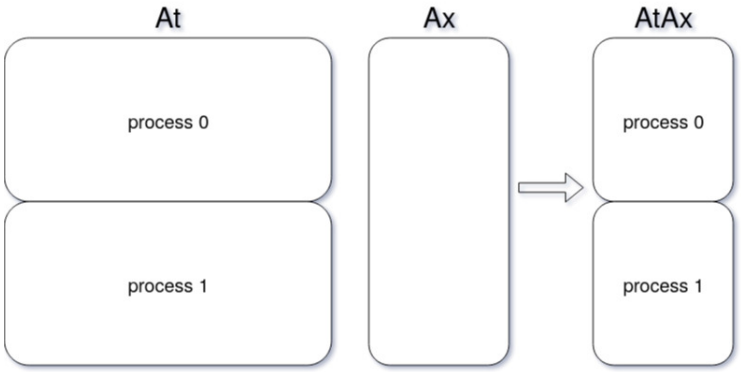

Figure 9: MPI on A^tAx

As x, a *Allgatherv* method was used to get all b's in order compute the sub-vector A^tb, this is represented in the Fig.10 and Fig.11.

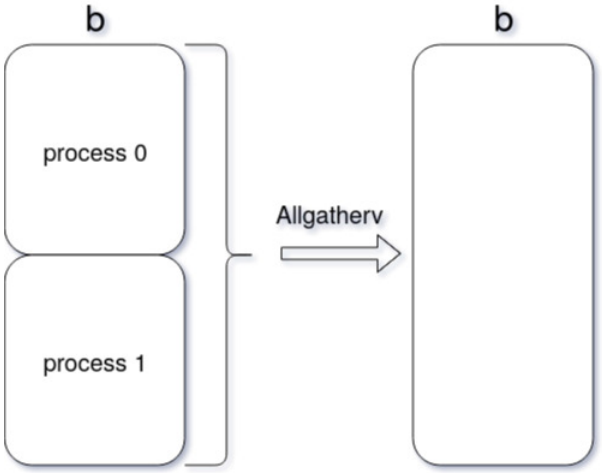

Figure 10: MPI on b

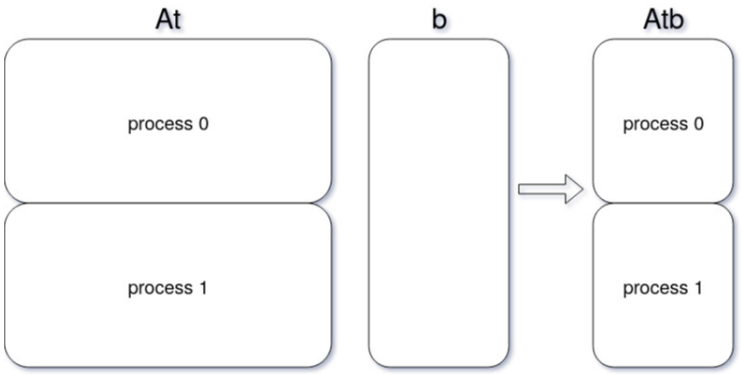

Figure 11: MPI on $A^t b$

At this point one has a sub-vector $A^t b$ and a sub-vector $A^t Ax$, so the sub-vectors r and p can be compute. In order to compute the guardian of the while loop, the first step was to compute $r * r$ on each process and then a *Allreduce* method were used in order to get the whole $r * r$ on every process, this is represented in the Fig.12.

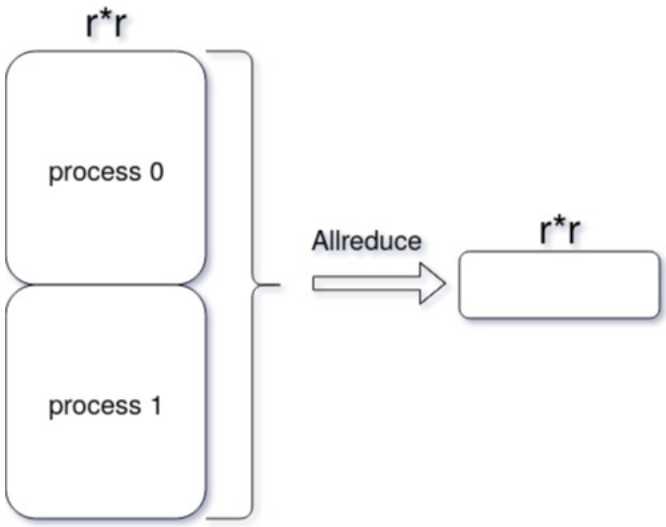

Figure 12: MPI on $r * r$.

After that, the *while* loop of the conjugate gradient method starts. One needs to compute the sub-vector $A^t Ap$, to do that, this will be the exact same principle as $A^t Ax$ (this is represented in Fig.9), indeed one starts by using *Allgatherv* in order to get the whole vector p, then one can compute the sub-vectpr Ap. One also needs to use *Allgatherv* in order to get the whole vector Ap and finally one can compute the sub-vector $A^t Ap$.

One also needs to compute the product $r * r$ in the exact same way as described before (in Fig.12).

It was also done, with the exact same principle as represented in Fig.12, in order to obtain $p * A^t Ap$.

Finally the last new MPI communication before the next iteration of the *while* loop was to do, again, the exact same thing as described in the Fig.12 but this time with the new r updated by the conjugate gradient algorithm.

3 Stability of the Explicit method

In this section, the stability of the explicit method will be studied as a function of the different parameters. The goal of this is to find a law that tells if the code will converge or not as a function of the parameters. According to *Von Neumann stability analysis*, the stability of the numerical code should follow a law given by:

$$\Delta t \sqrt{\frac{gH}{\Delta x^2} + \frac{gH}{\Delta y^2}} \leqslant 1, \tag{4}$$

where $\sqrt{gH}$ is the phase speed. By taking $\Delta x = \Delta y$ and knowing that $\sqrt{gH}$ is constant, the law reduces to:

$$\frac{\Delta x}{\Delta t} \geqslant C, \tag{5}$$

where C is a constant. This law means that the *numerical speed* given by $\frac{\Delta t}{\Delta x}$ should be at least equal to a constant given by the phase speed.
However, the simulations showed that the stability of the system is totally independent of the space step. The tests have been done with the *reflection.dat* file, for different time and space steps, the results are represented in Fig.13.

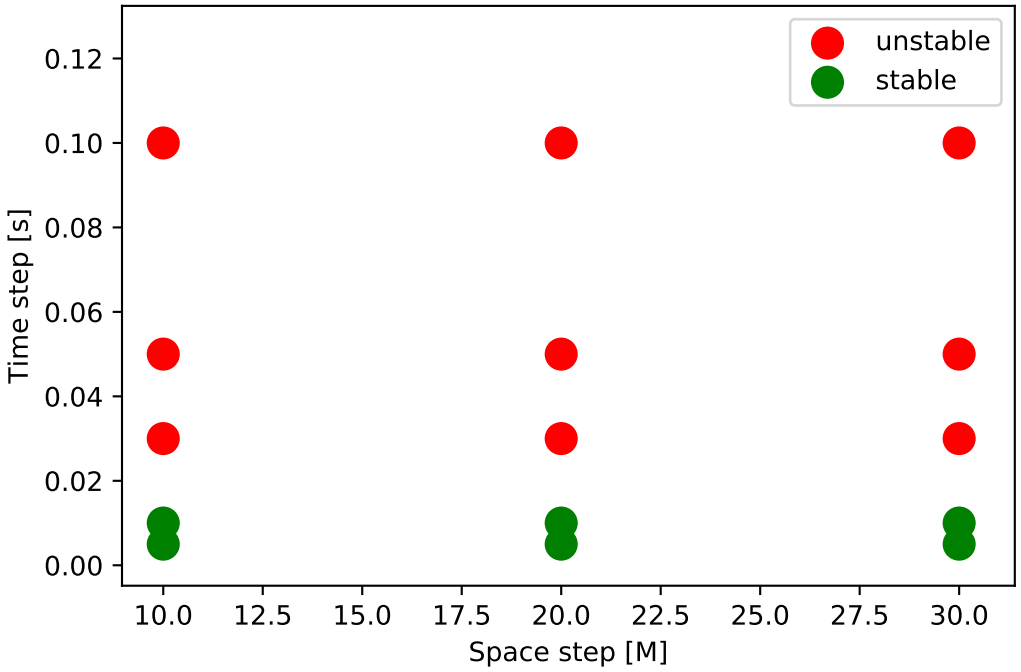

Figure 13: Study of the stability as a function of the steps

Unfortunately, the results are not those that were expected. Indeed the stability of the simulations is totally independent of the space step. The expected curve was a right with a slope proportional to the phase speed.

4 Scaling

In this section the effect of the weak scaling will be studied, this how the execution time varies with a constant problem size for each process/thread.

An other point that will be analysed is the strong scaling, this is how the execution time varies when the global size of the problem remains the same but the number of processes/threads varies.

4.1 Weak scaling

In order to perform the weak scaling, the size of the problem on every process needs to be the same. In order to determine this size, the following formula can be used :

$$\frac{nb\_lines\_eta}{nb\_process} = C \Leftrightarrow nb\_line\_eta \quad = C * nb\_process. \tag{6}$$

Here the "reference" will be the number of lines in η.
By doing so, in order to study this scaling, the results will be based on the $refraction.dat$ map with the parameters $g = 9.81$, $\gamma = 2e-5$, $\Delta x = 10$, $\Delta t = 0.01$, $A = 1$, $f = 0.1$, $S = 0$, $s = 0$ and $r_{tresh} = 1.e-2$. It can be noticed that the parameter T_{max} was not used but rather a certain number of iteration of the time loop in order to have a significant measurement.
The only parameter that will be modified at every measurement is Δy because one wants to have the same number of lines of eta on every process whatever the number of processes and therefore, for this test the number of lines is 100. So, C = 100.

Number of process	Number of lines in total	Δy
1	100	20
2	200	10
4	400	5
8	800	2.5
16	1600	1.25

Table 1: Evolution of Δy with respect to the number of process/number of lines of eta

4.1.1 Explicit method

In order to study the scaling for the explicit method, 2000 iterations of the time loop were made and Fig.14 represents the weak scaling efficiency for this method.

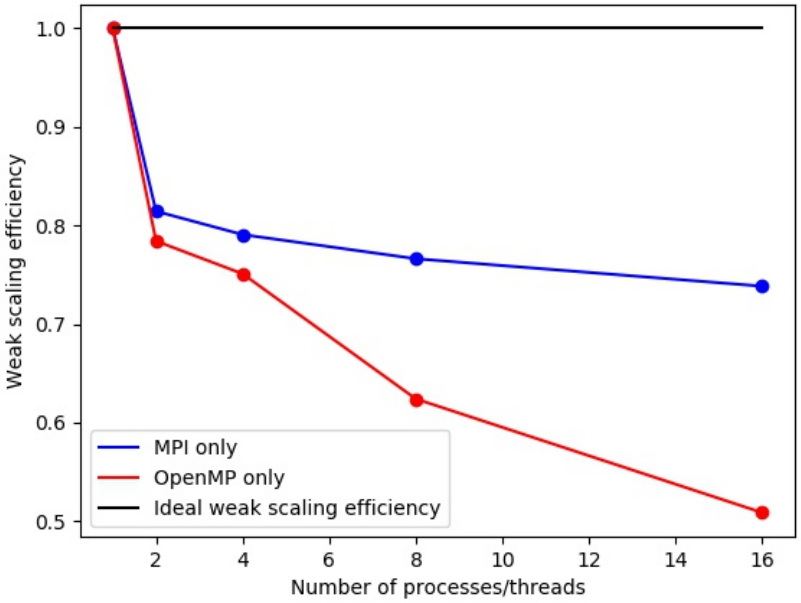

Figure 14: Weak scaling efficiency of the explicit method by using MPI or $OpenMP$ with a different number of processes/threads

It can be seen from the Fig.14, that the black line represents the ideal weak scaling efficiency, i.e. the execution time remains constant when the number of processes/threads grows up, if the processes/threads keep for instance a $nb\_lines\_eta = 100$ as said before.
For MPI, the efficiency decreases when the number of processes grows up. This can be explained by the fact that when the number of processes grows up, there will be more communication between the processes, because for this project the processes need to communicate between them. So, the bigger the number of processes, the bigger the number of communications and the bigger the number of communications, the bigger the time loss.
For $OpenMP$, the efficiency also decreases when the number of threads grows up. This can be explained by the fact that threads' creation and destruction has a cost, so when the number of threads is increased, this cost increases too and it can be seen that here, $OpenMp$ has a weak scaling efficiency lower than MPI.
It also can be noticed that the huge gap a when passing from 1 to 2 processes/threads could be explained by the fact that there will be some part of the code that won't be executed if there is only one process and there will be no creation or destruction of threads.

4.1.2 Implicit method

In order to study the scaling of the implicit method by using MPI and $OpenMP$, 100 iterations of the time loop were made and it can be seen from the Fig.15, that the graph has the same behavior has the explicit one (Fig.14) and this could be explained by the same hypothesis.

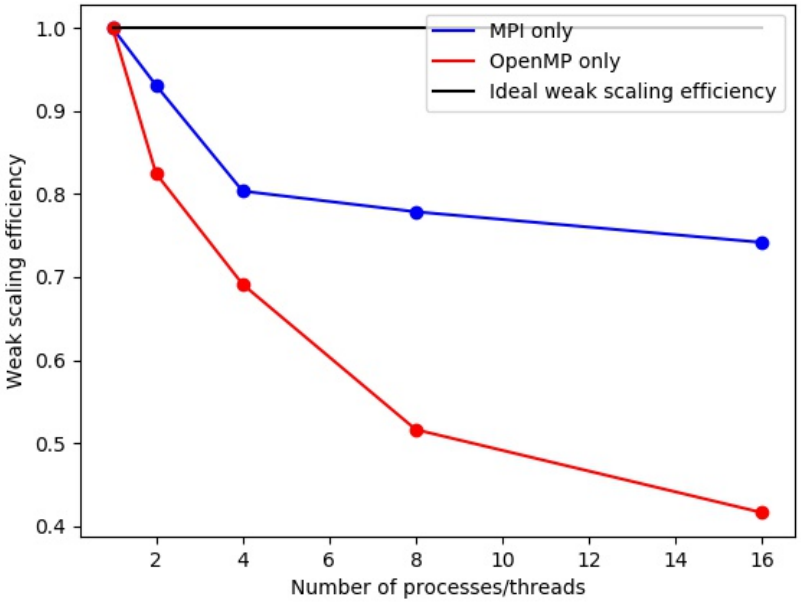

Figure 15: Weak scaling efficiency of the implicit method by using MPI or OpenMP with a different number of processes/threads

Even if there is more MPI communication between the processes, for the implicit part the communications were made in a smarter way. Indeed, for the explicit method the communications were only some *send* and *rcv* which could be ineffective. On the opposite, for the implicit method, the communications between the processes were *Allgatherv*, *Allreduce*,... which is more effective (or at least easier to use/implement).
The efficiency of the threads is lower and that could be explained by the fact that there is one *atomic* close and one *reduction* close which could affect the performance of the threads and thus decrease the weak scaling efficiency.

4.2 Strong scaling

In order to perform the strong scaling, the size of the problem will remain constant while the number of threads and the number of processes will grow up.
One should pay attention to the fact that the time measured in order to plot the Fig.16 and Fig.17 was only the time of the time loop with enough number of iterations in order to see a scaling on the measurements and with only one write a the end for the explicit method and with no write for the implicit method. This choice was made because the writing and the "loading" part can't be parallelized so it will bias the scaling.
The parameters used were $g = 9.81$, $\gamma = 2e-5$, $\Delta x = 2000$, $\Delta y = 2000$, $\Delta t = 0.05$, $A = 1$, $f = 0.001$, $S = 2000$, $s = 0$ and $r_{tresh} = 1.e-2$ and the map was *sriLanka.dat*.

4.2.1 Explicit method

In order to study the scaling for the explicit method, 2000 iterations of the time loop were made and the Fig.16 represents the scaling for this method.

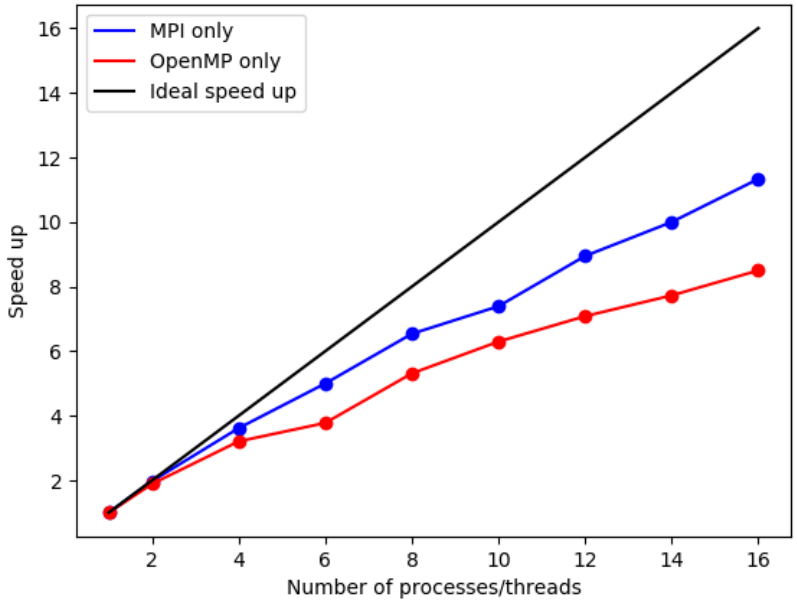

Figure 16: Speed up of the explicit method by using MPI or OpenMP with a different number of processes/threads

As it can be seen on the Fig.16, the black line represents the ideal speed up, i.e. the execution time is directly proportional to the number of processes/threads.
For MPI, the speed up follows quite good the ideal one until 8 processes, after that the speed up evolution begins to be less effective. This could be explained by the fact that in this project the processes need to send some communications to others, even in the time loop, so the bigger the number the number of processes, the bigger the number of communications and the bigger the number of communications, the bigger the time loss as already mention in the weak scaling section.
For $OpenMP$, the speed up follows quite good the ideal one until 4 threads, after that the speed up begins to be less effective. This can be explained by the fact that the threads' creation and destruction have a cost, so as for MPI, the bigger the number of threads, the bigger the time loss as already mention in the weak scaling section.
It must also be noticed that even in the time loop everything wasn't parallelized, for instance the *memcpy* operation or the writing so it mays be a part of the scaling loss.

4.2.2 Implicit method

The strong scaling results obtained with the implicit method is represented in the Fig.17.

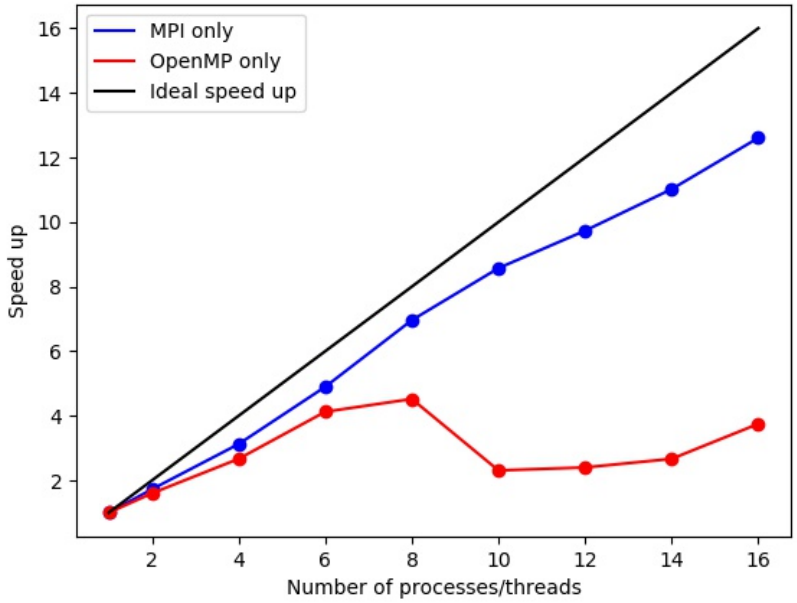

Figure 17: Speed up of the implicit method by using MPI or OpenMP with a different number of processes/threads

For MPI, it can be seen that the speed up follows quite good the ideal one until 10 processes and after that the speed up begins to be less effective. This could be caused by the same reasons as the explicit method and a possible explanation was said in the weak scaling section.
For $OpenMP$, the figure shows that the speed up is quite good until 4 or 6 threads and after it begins to have a strange shape, indeed when the number of threads grows up, the time needed to run the time loop decreases which should normally not append. A possible explanation, as said before, could be the *atomic* and *reduction* close but this is not sure.
Be aware that there is some operations that couldn't be done in parallel and this could also influence the performance.

5 Comparison of the performances

5.1 Comparison of the stability

As seen in the section 3, the explicit method is not stable for any parameters values, for example if a too large time step is taken, the system will probably explode which will lead to some very high or very small values.
On the opposite, the implicit method, as seen in the theoretical course, is unconditionally stable which means that too large time step for the explicit method could be choose.
One should be careful that the implicit method is the exact solution but rather an approximation of what the wave should be.
The Fig.18 and Fig.19 show the difference of stability between the explicit and the implicit method when using parameters that are unstable for the explicit method, i.e. $g = 9.81$, $\gamma = 2e-5$, $\Delta x = 2000$, $\Delta y = 2000$, $\Delta t = 1$, $T_{max} = 2000$, $A = 1$, $f = 0.001$, $S = 2000$, $s = 0$ and $r_{tresh} = 1.e-2$ and the map was $sriLanka.dat$.

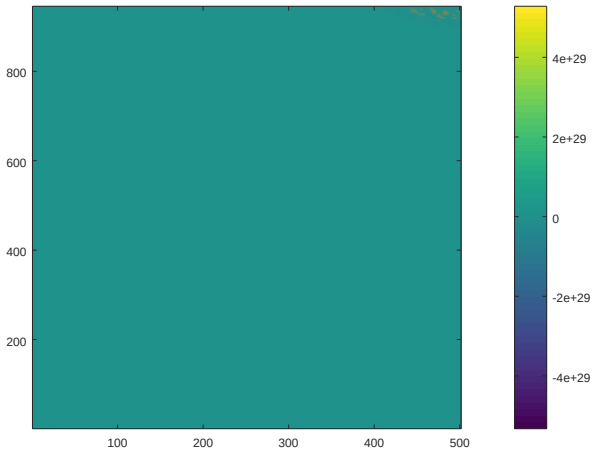

Figure 18: Results of the explicit method for the η

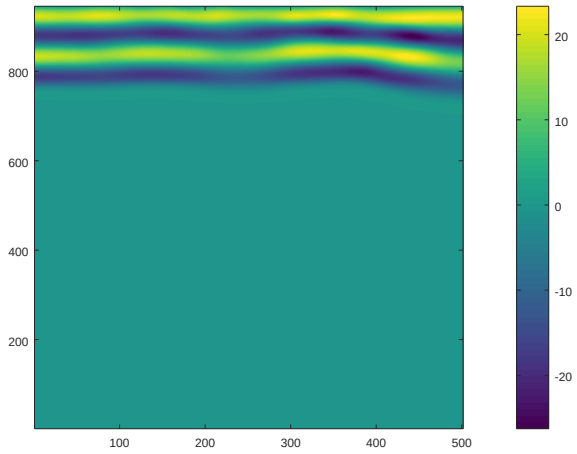

Figure 19: Results of the implicit method for the η

As it can be seen on the Fig.18 and Fig.19, the Fig.18 is totally unstable, indeed it has results between $4e-29$ and $-4e-29$ and there are some strange pixels at top right of the figure while the Fig.19 is a quite good estimation of the real wave.

5.2 Results and efficiency

As observed during the testing phase, the implicit method is a lot slower than the explicit one and it is even slower because the matrix A is not symmetric positive-definite (because as said previously, one has to multiply everything by At in order to assure the convergence of the conjugate gradient). For instance, for the parameters used in the sub-section above, the time needed with 16 processes and 1 thread to reach 2000 iterations of the time loop was 17.86s for the explicit and 1 hour for the implicit. One should pay attention to the fact that the number of iterations of the conjugate gradient increased with these parameters which translates an increase of the time required to compute an iteration of the time loop.

The utility of the implicit method may be not seen directly but, because it is unconditionally stable, a larger time step can be used which should lead to faster results but here it's not the case, the explicit method is still faster than the implicit one.
An other advantage of the implicit method is based on the physical sense, indeed in the real world

the components will not necessarily depend on the components of the previous time and thus an implicit method should be used and as it can be seen from Fig.20 and Fig.21 the implicit method gives a very good result.

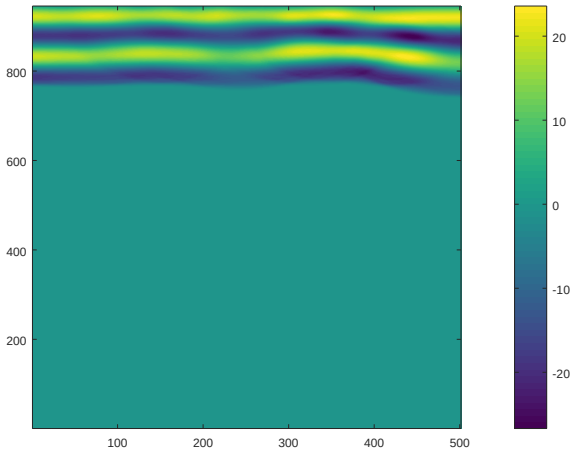

Figure 20: Results of the explicit method for the η

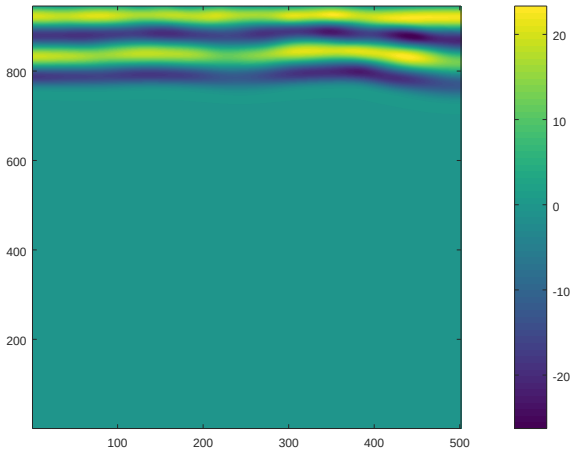

Figure 21: Results of the implicit method for the η

The parameters used are the same as for the comparison of the stability with $\Delta t = 0.05$ for the explicit method and $\Delta t = 1$ for the implicit one.

6 Conclusion

There are many ways to solve a PDE problem using a numerical code. Each method has its advantages and drawbacks. An explicit method provides equations that can be solved directly without the use of an iterative method. This advantage avoids the numerical code of an additional loop that would increase the time of computation. However, an explicit method is not always stable. As a consequence, the user needs to choose carefully the parameters that are used in order to have physically admissible results. By contrast, an implicit method is always stable but requires an additional loop in order to solve the system. The parallelization method is also a problem that needs to be taken into account. Indeed, the most efficient way to parallelize a computer code is without any interaction between the processes/threads. However most of the time, some communication is needed as well as the different threads need sometimes to interact.

As a consequence, a computer code needs to be paralellized in a way that minimize the amount of interaction between the processes and threads. The different results that have been obtained showed also that the analysis of a computer code can not be so trivial as many features of the code can influence the results. A more in-depth analysis of the computer code could answer to these strange results.