University of Liège

High performance Scientific computing

Project 2 - Report

Master in Data Science & Engineering

Authors:
Arnaud Delaunoy
Tom Crasset

Professor :
C. Geuzaine

Academic year 2019-2020

1 Introduction

The aim of this project is to familiarise the student to high performance computing and to parallel programming. This report is going to study the wave propagation on the surface of the ocean. The "shallow water" equations in (1) will be used with a finite difference method to model the phenomenon. Moreover, this report will also study the differences between an implicit and an explicit finite difference scheme. The implementation will make use of the OpenMP library and more specifically the MPI library in C. Both these libraries will be used to parallelize the computation and the effects on the computation time and memory consumption will be studied too.

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

where the unknown 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. Moreover, we have

- $h(x, y)$, the depth at rest
- $g = 9.81 m/s^2$, the gravitational constant
- γ, the dissipation coefficient

2 Implementation

In this section, we're going to focus on the high level concepts of the implementation, not the little details like the indices of the formulas.

2.1 Explicit Euler

The values for each element of η, v and v were stored in two dimensional arrays for simplicity. In this setting, we traded simplicity for efficiency as one dimensional arrays only require one pointer evaluation to reach a given value, while our method requires 2.

However, for the subdivision of the domain space between processes, we were able to mitigate this loss in speed by deciding to split it along the X-axis . This choice was done because the C programming language is row major, meaning that consecutive elements of a row reside next to each other in memory. Thus accessing `eta[i][j]` placed the whole row `eta[i]` in cache, and thus accessing `eta[i][j + 1]` was nearly instantaneous.

The general case is depicted on Figure (1). It shows that, every iteration, processes included in the interval $[0..N-2]$ send the last column of their η matrix to the process of the rank above and how each process in the interval $[1..N-1]$ sends their first column of u to the process with the rank below.

There is an edge case when there are only 2 processes because only process 0 sends η (and receives u) and only process 1 sends u (and receives η).

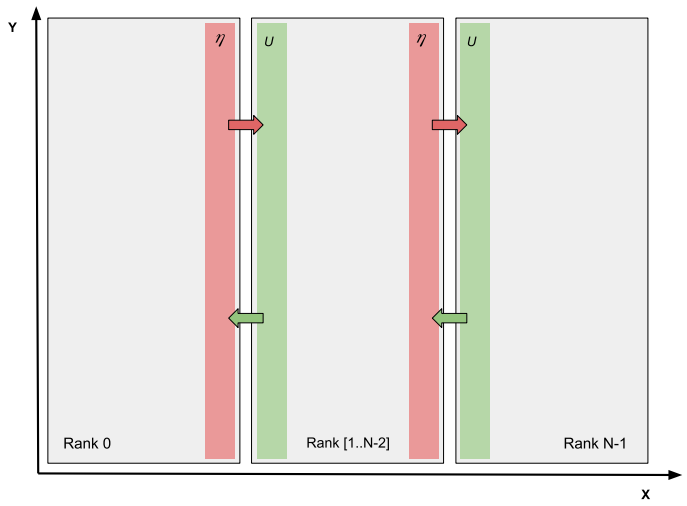


Figure 1: Subdivision of the domain space using N processes. The arrows show the direction of the inter-process communication of the variable of the same colour.

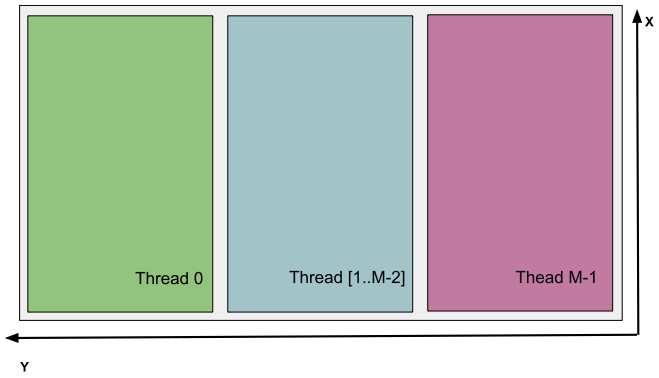


Figure 2: Subdivision of one process into M threads.

2.2 Implicit Euler

The main difference between explicit and implicit Euler scheme is that a Linear system must be solved in the implicit scheme. For solving this system we have chosen to work with the conjugate gradient method. The main reason for that is that the most expensive operation in this algorithm a Matrix-vector multiplication that can be implemented efficiently when the matrix is sparse which is the case for our system.

Process Parallelism In the implicit setting, the parallelism is done at the conjugate gradient level. Consider the system to solve

$$\mathbf{Ax} = \mathbf{b}$$

Each process possesses the full vector $\mathbf{b}$ and intermediate vectors used in the conjugate gradient method. The matrix $\mathbf{A}$ is however split between the process, each possesses some lines of $\mathbf{A}$, each line corresponding to a constraint.

When performing a vector-vector dot product, each process compute a part of the multiplications. Their results are then added with a reduction clause. When performing a matrix-vector multiplication, each process computes the multiplications corresponding to its part of the matrix $\mathbf{A}$. The results are then gathered between all the processes.

Conjugate gradient In order to use the conjugate gradient method The matrix $\mathbf{A}$ must be symmetric semi-definite positive. We therefore multiply both side of the equation $\mathbf{Ax} = \mathbf{b}$ by $\mathbf{A^T}$. In order to do that, we first compute in each process the lines of $\mathbf{A^T}$ this process is responsible for and save it. After this, we compute all the lines of A^T one by one saving only the current line. For each line, we compute the vector-vector dot product of the elements corresponding to the current vector with each line in the saved $\mathbf{A^T}$. We also compute the vector-vector dot product of the current vector and b. At the end of the process has the full vector $\mathbf{A^Tb}$ and the lines it is responsible for of $\mathbf{A^TA}$.

Sparse matrix representation As there will always be at least one non zero element per line, we decided to represent a sparse matrix by an array of sparse vectors. Each sparse vector is represented by two array, each of the size of the number of non zero elements. The first array contain the indices if the elements and the second array the values of those elements.
In addition to efficient storage, this representation of vectors allows efficient dot product operation. Indeed, one just need to loop over the array of indices instead of all the possible indices as it would have been done with non sparse representation.
However for this efficient dot product being possible, indices must be sorted. Therefore operations need to be performed on insertion in order to keep the array sorted. For that we loop over the indices to find the right insertion place starting from the back. In the rest of the program, we try as much as possible to insert the indices in order such that to reduce this insertion overhead. Note that for large vectors, inserting all the elements without caring about the order and sorting with quicksort at the end would have been more efficient. However, as we work with vectors that contains very few elements, this is not worth.

Thread parallelism Thread parallelism can be performed at several levels.
First, we can parallelize the construction of the matrix $\mathbf{A}$. Each element of $\mathbf{A}$ can be compute by different threads
Second, we can parallelize the dot product of dense vectors. Each thread compute the element-by-element multiplications of part of the vector. Note that sparse vector dot product is not worth parallelizing as those contain very few elements.
Finally, Sparse matrix-vector multiplication can be parallelized. Each thread is responsible for part of the lines of the matrix.

3 Stability of the explicit numerical scheme

In this section, we are going to analyze the stability of the explicit scheme. We used the sinusoidal excitation so that the waves are generated forever and 20000 iterations. There are two factors that may cause instability: A too high time step and a too low discretization. In order to analyse the point where instability appears we therefore fix one of the parameters and make the other vary.

First we analyse the instability when varying the timestep. A first broad search has been performed and is shown at figure (3). We see that instability appears for $\Delta T \in [0.1, 1]$ for $\Delta X = 2000$ and $\Delta Y = 2000$.

Then we analyse the instability when varying the spatial step. Again, a first broad search was performed and it is shown on Figure (4). We see that instability appears for ΔX and $\in [500, 1000]$ for $\Delta T = 0.05$.

Looking more closely at these threshold, we see from Figures (5) and (6) that these values are actually between $[575, 600]$ for ΔX and and $[0.2, 0.4]$ for ΔT.

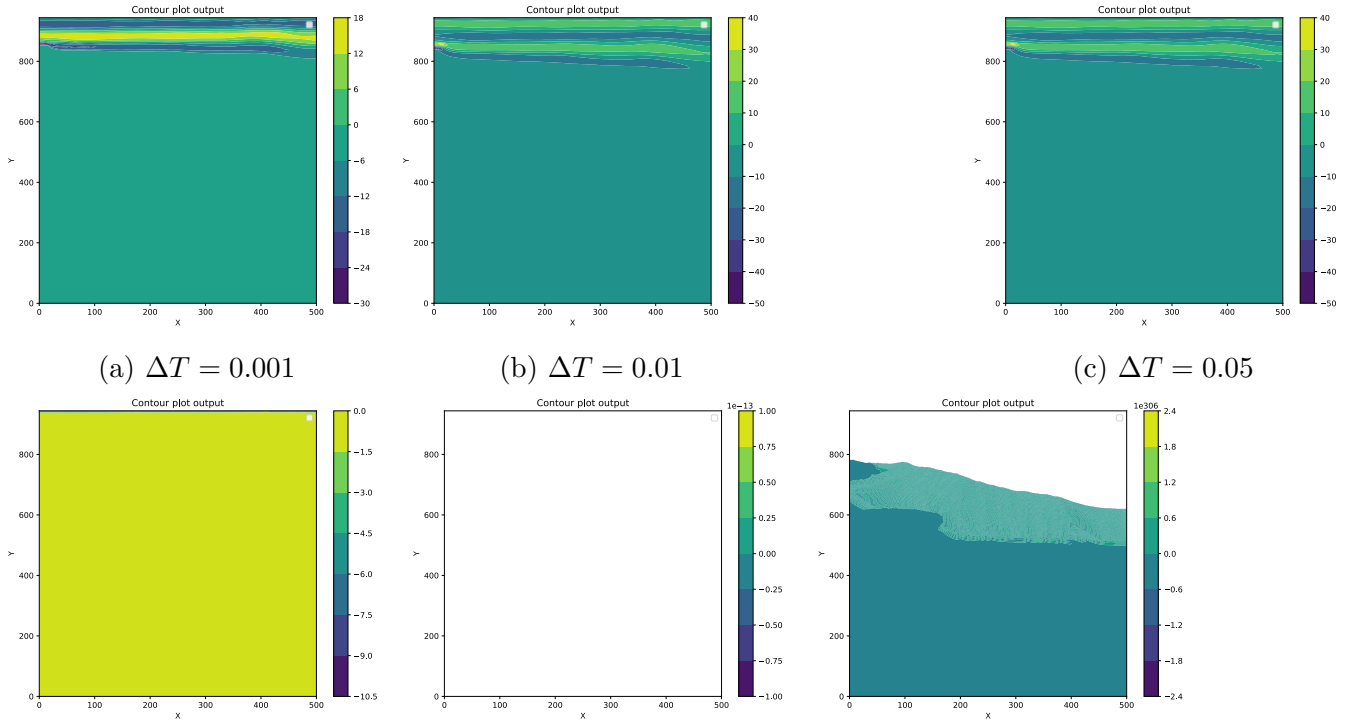
From the third chapter of [sha], who use the same model as ours except that they don't disregard the vertical velocity, we find that the stability of the explicit scheme depends on equation (2) and if the spatial steps are equidistant, we have the formula at equation (3).

$$\Delta T \sqrt{\frac{gH}{\Delta X^2} + \frac{gH}{\Delta Y^2}} \leq 1 \text{ with } H = 3414\text{m being the average height of the water.} \tag{2}$$

$$\frac{\Delta T}{\Delta X}\sqrt{gH} \leq \sqrt{\frac{1}{2}} \tag{3}$$

However, using this formula with our values, we find that the threshold is at $\frac{\Delta T}{\Delta X} \leq \sqrt{\frac{1}{2gH}} = \frac{1}{2*9.81*3414} = 0.00386$.

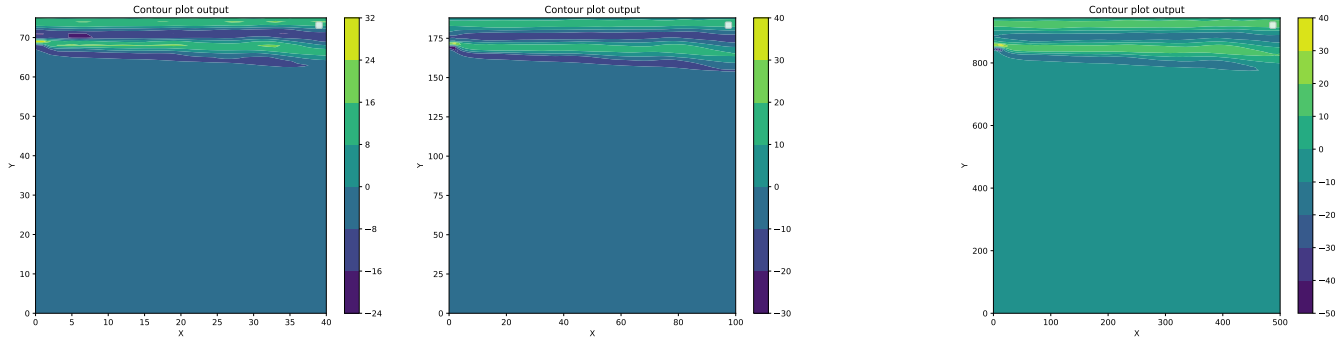
Experimentally, we don't have the same threshold $C = 0.00386$. When looking at a fixed $\Delta X = \Delta Y = 2000$ we find that instability appears for $\Delta T \in [0.2, 0.4]$, therefore that $C \in [10^{-4}, 2 \times 10^{-4}]$. When looking at a fixed $\Delta T = 0.5$, we find that instability appears for $\Delta X \in 575, 600$, therefore $C \in 8.3 \times 10^{-4}, 8.7 \times 10^{-4}$. We can therefore conclude that C is of the order of magnitude 10^{-4}.



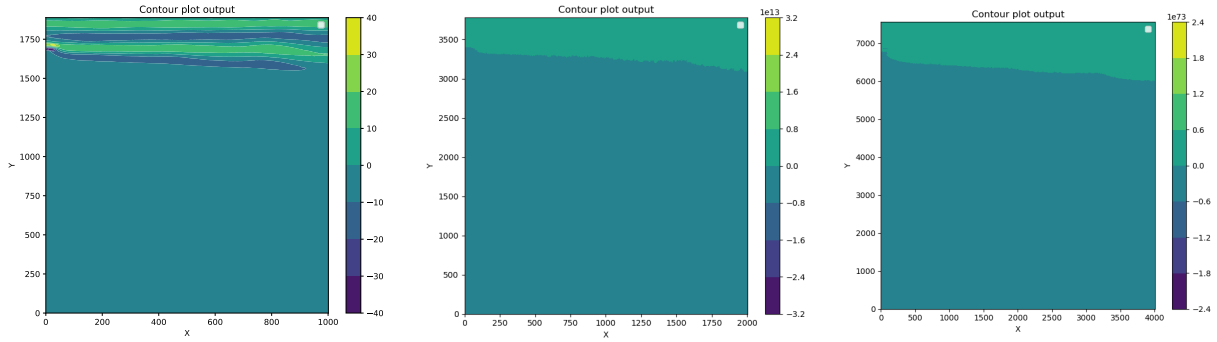
(a) $\Delta T = 0.001$ (b) $\Delta T = 0.01$ (c) $\Delta T = 0.05$

(d) $\Delta T = 0.1$ (e) $\Delta T = 1$ (f) $\Delta T = 10$

Figure 3: Varying parameter ΔT while keeping the spatial step ΔX and ΔY constant at 2000

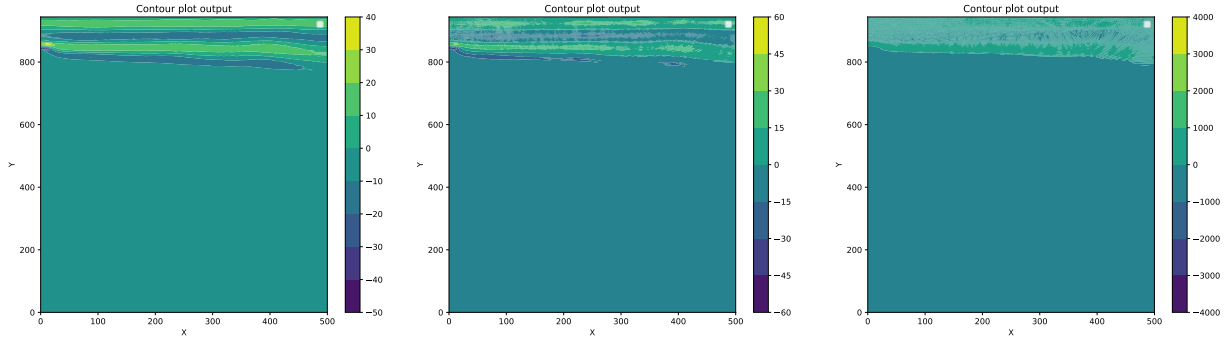


(a) $\Delta X, \Delta Y = 25000$ (b) $\Delta X, \Delta Y = 10000$ (c) $\Delta X, \Delta Y = 2000$



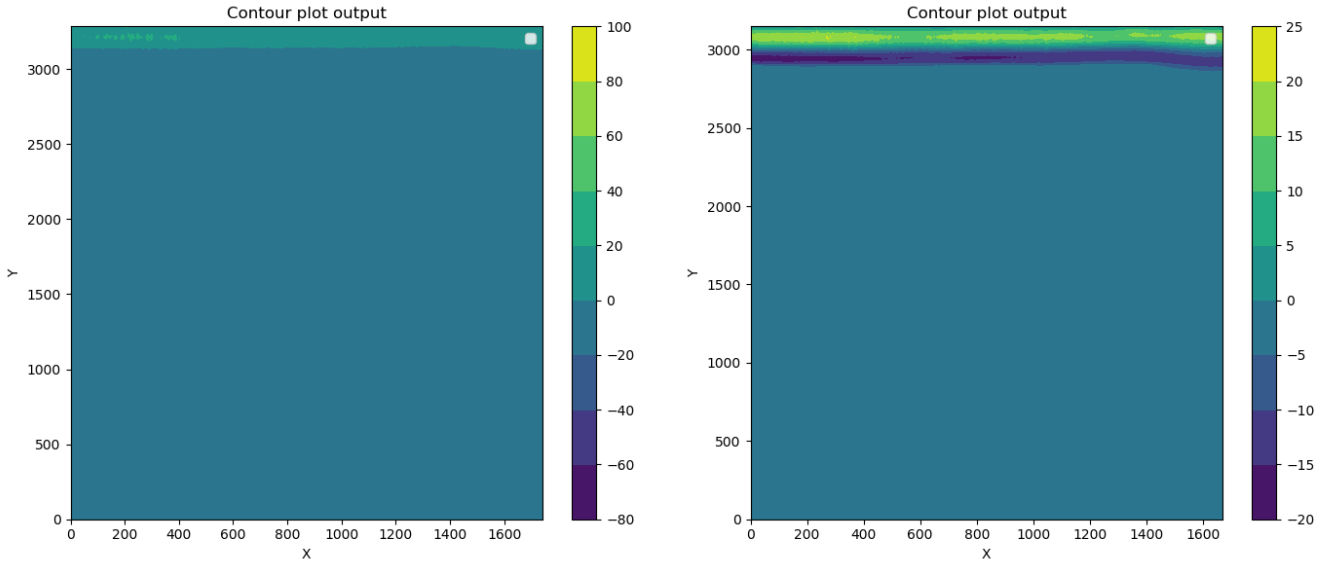
(d) $\Delta X, \Delta Y = 1000$ (e) $\Delta X, \Delta Y = 500$ (f) $\Delta X, \Delta Y = 250$

Figure 4: Varying spacial steps ΔX and ΔY while keeping the temportal step ΔT constant at 0.05



(a) $\Delta T = 0.2$ (b) $\Delta T = 0.3$ (c) $\Delta T = 0.4$

Figure 5: This is the spot when the instability appears with $\Delta X; \Delta Y = 2000$



(a) $\Delta X; \Delta Y = 575$ (b) $\Delta X; \Delta Y = 600$

Figure 6: This is the spot when the instability appears with $\Delta T = 0.05$

4 Scalability analysis of the explicit and implicit numerical scheme

Investigating the strong scaling of an implementation is measuring how efficient an application is when using increasing numbers of parallel processing elements.

The measurements were done on the NIC4 cluster of the CECI(Consortium des Équipements de Calcul Intensif). The cluster has the following characteristics [cec] :

"Hosted at the University of Liège (SEGI facility), it features 128 compute nodes with two 8-cores Intel E5-2650 processors at 2.0 GHz and 64 GB of RAM (4 GB/core), interconnected with a QDR Infiniband network, and having exclusive access to a fast 144 TB FHGFS parallel filesystem."

4.1 Explicit strong scaling

In this section, we are tasked with computing the strong scaling of the explicit scheme. We measured the strong scaling separetely between threads and processes. That is, while we increase one of them, the other was kept at 1 so as not to have an influence on the other.

Using the aforementioned supercomputer, we measured the values shown on Table (1) for the strongscaling of processes and on Table (2) for the strongscaling of threads.

Visually, one can see the results on Figures (7) and (8). These show the results of the strong scaling measurements of the application alongside a perfect scaling, for processes and threads respectively. On the left, the actual average running time for 1000 iterations is plotted and on the right plot, the speedup factor with respect to a single process/single threaded application is depicted.

On Figure (7), we can clearly see that as the number of processes increase, the runtime decreases nearly linearly. Using the formula (4) for strong scaling, we get a strong scaling efficiency of 95.5 for 2+ processes.%. This is an excellent strong scaling and is very useful to accelerate the computation of problems of such a scale.

On Figure (8) however, the speedup is not as great anymore. For the first 2 threads, we see a good scaling but at 4 threads there is an an inflection point and the performance is not scaling linearly anymore and stagnates. Using the formula (4) for strong scaling, we get a strong scaling efficiency of 46.7 for 8 threads and 25% for 16 threads.%.

One of our hypotheses as to why that is, is the simple fact that the operations that we parallelize using threads in the explicit scheme are rather few and arriving at higher threading values, the speedup is not good enough compared to the increased communication.

$$\frac{t_1}{nt_n} * 100 \tag{4}$$

Number of processes	1	2	4	8	10	12	14	16
Average runtime	359	179	90	46	37	31	26	23

Table 1: Runtime (in seconds) of 1000 iterations of the explicit scheme using various numbers of processes with 1 thread each, averaged over 3 runs

Number of threads	1	2	4	8	10	12	14	16
Average runtime	359	217	143	95	97	95	95	89

Table 2: Runtime (in seconds) of 1000 iterations of the explicit scheme using one process and various numbers of threads, averaged over 3 runs

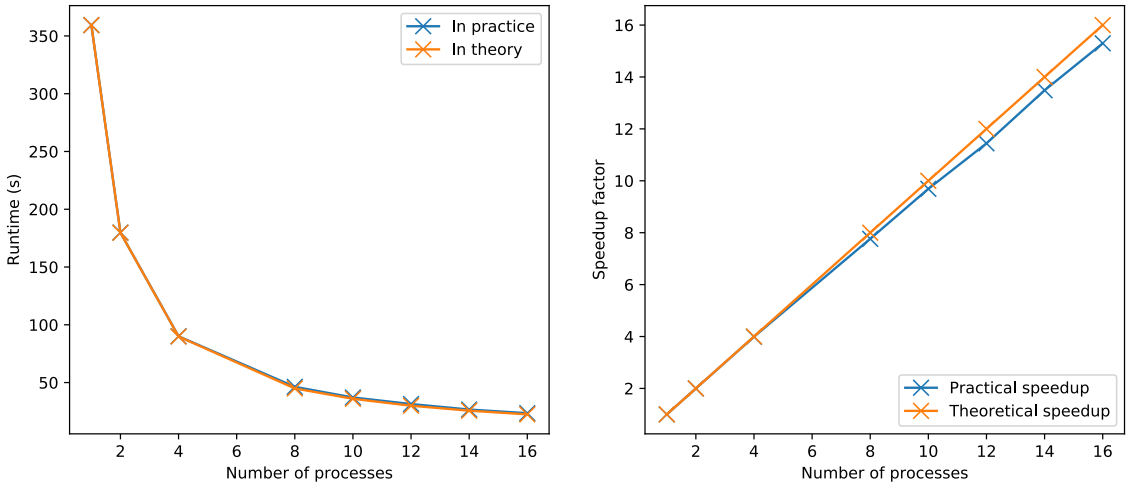


Figure 7: This Figure shows the strong scaling efficiency with processes of the explicit finite difference scheme. Left) Average runtime with respect to the number of processes, both in theory and applied to our problem. Right) Average speedup with respect to the number of processes, both in theory and applied on our problem.

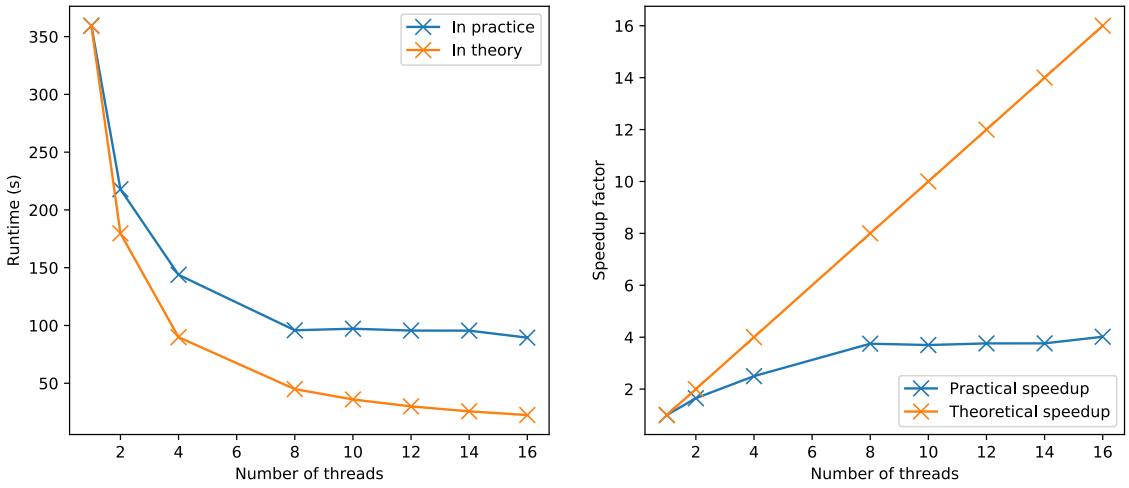


Figure 8: This Figure shows the strong scaling efficiency with threads of the explicit finite difference scheme. Left) Average runtime with respect to the number of threads, both in theory and applied to our problem. Right) Average speedup with respect to the number of threads, both in theory and applied on our problem.

4.2 Explicit weak scaling

Another measure of efficiency is weak scaling. During this test, one increases the number of processing elements without decreasing the load of each processing element. In this case, we maintain the number of cells that each process/thread has to compute constant by increasing the global computation space.

During the measurement, we increased the number of threads and the number of processes separetely. For measuring the weak scaling of processes, we used 1 thread per process to really focus on the scaling of the threads without outside interference. For measuring the weak scaling of threads, we did an analogous experiment and kept the number of processes at 1.

First, we measured the weak scaling efficiency with regards to processes. That is, we varied the problem size along the x-axis while maintaining the size along the y-axis constant. The different parameters can be seen in Table (3) and the results can be seen in Table (5) and in Figure (9).

Again, on the left of each figure, we plotted the average runtime and on the left, we plotted the speedup. In this case, the speedup corresponds to the weak scaling efficiency, as we made sure that there would be no speedup by keeping the per process problem size constant. Thus, an implementation that has a good weakscaling efficiency will not take more time to compute the whole grid.

And as we can see on Figure (9), the weak scaling efficiency doesn't drop below 95% for any number of process. This is a very good weak scaling effiency.

Then, we measured the weak scaling efficiency with regards to threads. This time, we varied the problem size along the y-axis while maintaining the size along the x-axis constant. This time around, the parameters used are in Table (4) and the results can be seen in Table (6) and in Figure (10).

Looking at the left of Figure (10), one can see that the weak scaling efficiency is not as great for the threads as it is the processes. However, the graph paints a darker picture than what is actually happening, because it is not on the same scale. By looking at the actual weak scaling efficiency, we see that it remains above 80%. However, note that there is a linear decline in the efficiency as the number of threads increases.

Number of processes	ΔX	ΔY
1	4000	1890
2	2000	1890
4	1000	1890
8	500	1890
10	400	1890
12	333	1890
14	285	1890
16	250	1890

Table 3: Parameters for different processes with each process having an effective size of the domain remaining constant at 250×1000

Number of threads	ΔX	ΔY
1	2000	1890
2	2000	945
4	2000	472
8	2000	236
10	2000	188
12	2000	157
14	2000	134
16	2000	118

Table 4: Parameters for different threads with each thread having an effective size of the domain remaining constant at 500×1000

Number of processes	1	2	4	8	10	12	14	16
Average runtime	167	169	172	177	171	173	171	175

Table 5: Runtime (in seconds) of the explicit scheme using various numbers of processes with 1 threads each, averaged over 3 runs

Number of threads	1	2	4	8	10	12	14	16
Average runtime	332	344	359	362	382	422	406	402

Table 6: Runtime (in seconds) of the explicit scheme using one process and various numbers of threads, averaged over 3 runs

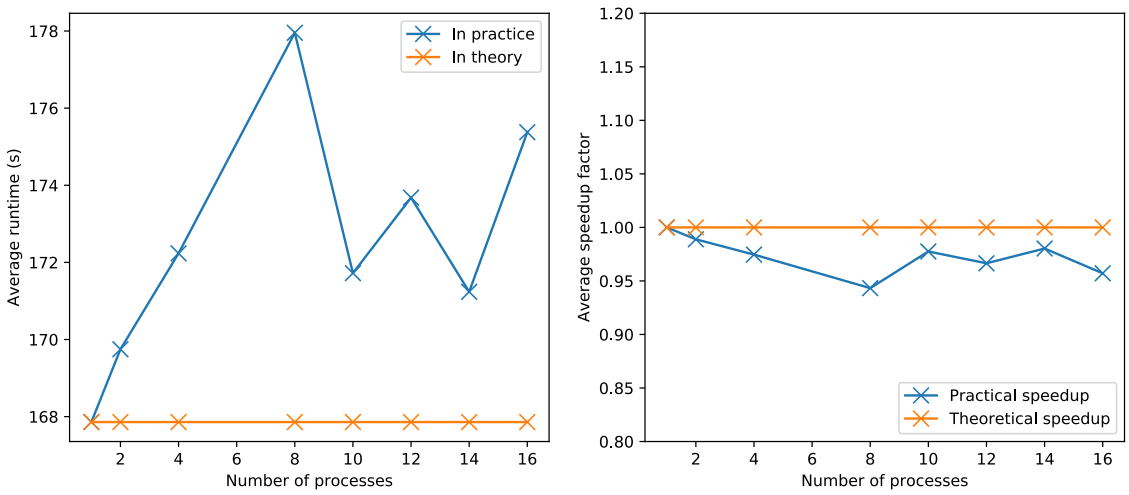


Figure 9: In this figure, we measure the weak scaling efficiency with regards to processes of the explicit finite difference scheme. The per process problem size was maintained constant. Left) Average runtime with respect to the number of process of the explicit scheme. Right) Average speedup with respect to the number of processes of the explicit scheme.

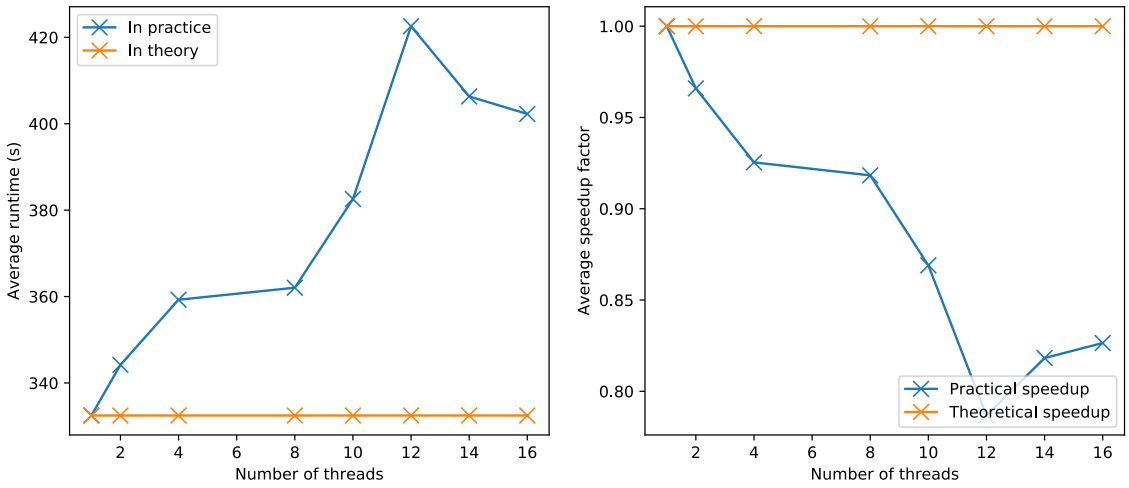


Figure 10: In this figure, we measure the weak scaling efficiency with regards to threads of the explicit finite difference scheme. The per process problem size was maintained constant.Left) Average runtime with respect to the number of threads of the explicit scheme. Right) Average speedup with respect to the number of processes of the explicit scheme.

4.3 Implicit strong scaling

In this section, we are tasked with computing the strong scaling of the implicit finite difference scheme. Although the implementation is vastly different, we would expect a similar scaling as for the explicit finite difference scheme.

We measured the strong scaling separately between threads and processes. That is, while we increase one of them, the other was kept at 1 so as not to have an influence on the other.

Using the supercomputer, we measured the values shown on Table (7) for the strongscaling of processes and on Table (8) for the strongscaling of threads.

Visually, one can see the results on Figures (11) and (12).

As expected, the strong scaling efficiency with regards to processes for the implicit finite difference scheme on Figure (11) is as good as the one for the explicit finite difference scheme.

However, despite what would be expected, the strong scaling efficiency for threads is better than it was for the explicit scheme. Indeed, by looking at Figure (12), one can see that the strong scaling efficiency for threads is as good as the one for threads. One of the possible reasons is that during the implicit implementation, vector-vector or matrix-vector multiplications are done at every loop, and thus using more threads speeds up these lengthy computations. The fast computations in the explicit scheme are done faster and thus, using more threads will likely add too much overhead and nullify the speedup of using more threads.

Number of processes	1	2	4	8	10	12	14	16
Average runtime	507	264	139	70	55	46	48	45

Table 7: Runtime (in seconds) of the implicit scheme using various numbers of processes with 1 thread each, averaged over 3 runs

Number of threads	1	2	4	8	10	12	14	16
Average runtime	513	261	140	72	59	49	43	37

Table 8: Runtime (in seconds) of the implicit scheme using one process and various numbers of threads, averaged over 3 runs

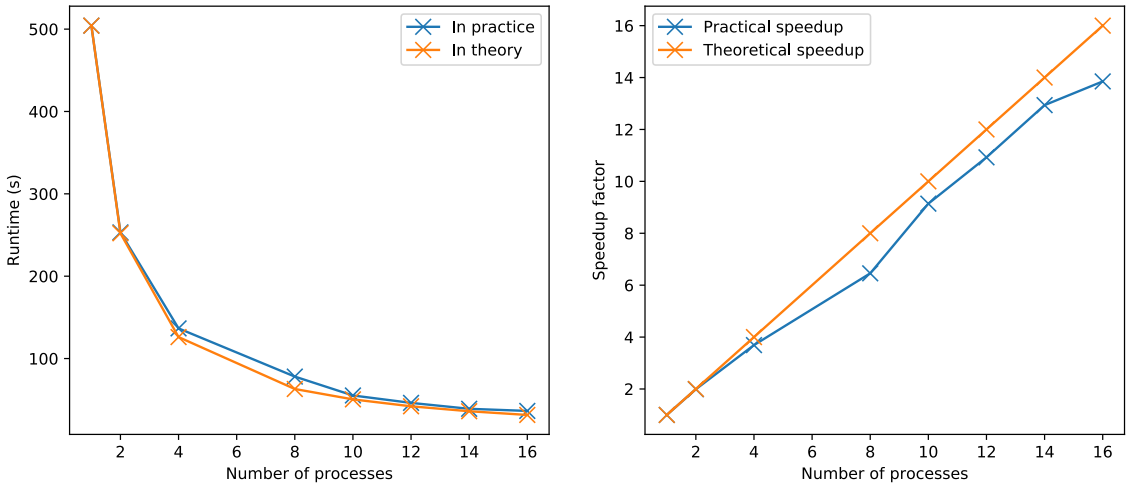


Figure 11: Left) Average runtime with respect to the number of processes, both in theory and applied to our problem. Right) Average speedup with respect to the number of processes, both in theory and applied on our problem.

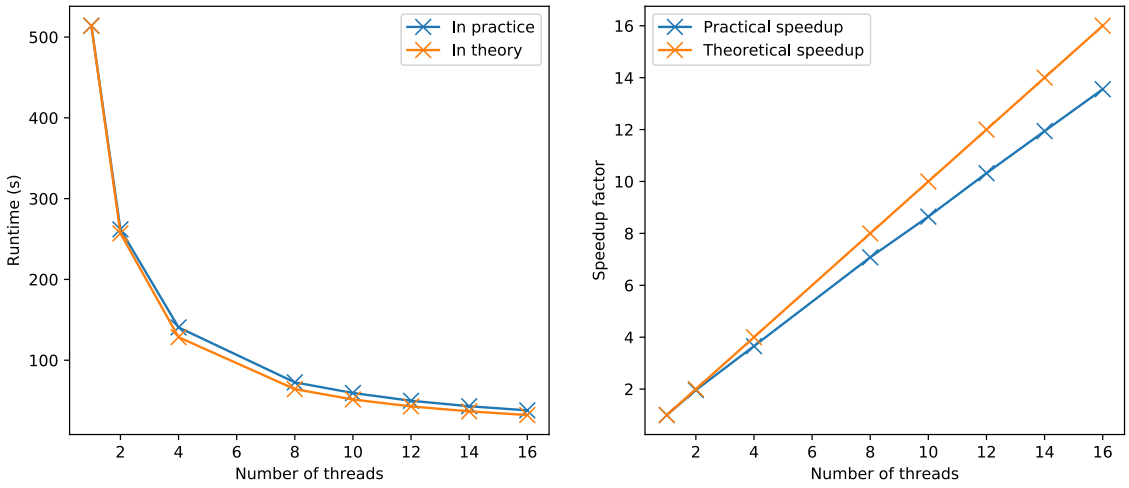


Figure 12: Left) Average runtime with respect to the number of threads, both in theory and applied to our problem. Right) Average speedup with respect to the number of threads, both in theory and applied on our problem.

4.4 Implicit weak scaling

During the measurement, we increased the number of threads and the number of processes separetely. For measuring the weak scaling of processes, we used 1 thread per process to really focus on the scaling of the threads without outside interference. For measuring the weak scaling of threads, we did an analogous experiment and kept the number of processes at 1.

Again, we first measured the weak scaling efficiency with regards to processes. That is, we varied the problem size along the x-axis while maintaining the size along the y-axis constant. The different parameters can be seen in Table (9) and the results can be seen in Table (11) and in Figure (13).

Then, we measured the weak scaling efficiency with regards to threads. This time, we varied the problem size along the y-axis while maintaining the size along the x-axis constant. This time around, the parameters used are in Table (10) and the results can be seen in Table (12) and in Figure (14).

However, looking at the Figures (13) and (14), we see that the weak scaling efficiency is 0%. The computation time seems to depend on the total problem size and, as we increase the total problem size but keep the per-processling-element problem size constant, we have a slowdown.

On the left Figure (13) and (14), we see the line depicting 'No speedup' and the measured values seems to be coherent with that approach.

Number of processes	ΔX	ΔY
1	80000	25000
2	40000	25000
4	20000	25000
8	10000	25000
10	8000	25000
12	6666	25000
14	5714	25000
16	5000	25000

Table 9: Parameters for different processes with each process having an effective size of the domain remaining constant at 12×75

Number of threads	ΔX	ΔY
1	25000	80000
2	25000	40000
4	25000	20000
8	25000	10000
10	25000	8000
12	25000	6666
14	25000	5714
16	25000	5000

Table 10: Parameters for different threads with each thread having an effective size of the domain remaining constant at 40×23

Number of processes	1	2	4	8	10	12	14	16
Average runtime	52	102	209	594	509	622	721	818

Table 11: Runtime (in seconds) of the implicit scheme using various numbers of processes with 1 threads each, averaged over 3 runs

Number of threads	1	2	4	8	10	12	14	16
Average runtime	52	105	218	444	561	675	784	894

Table 12: Runtime (in seconds) of the implicit scheme using one process and various numbers of threads, averaged over 2 runs

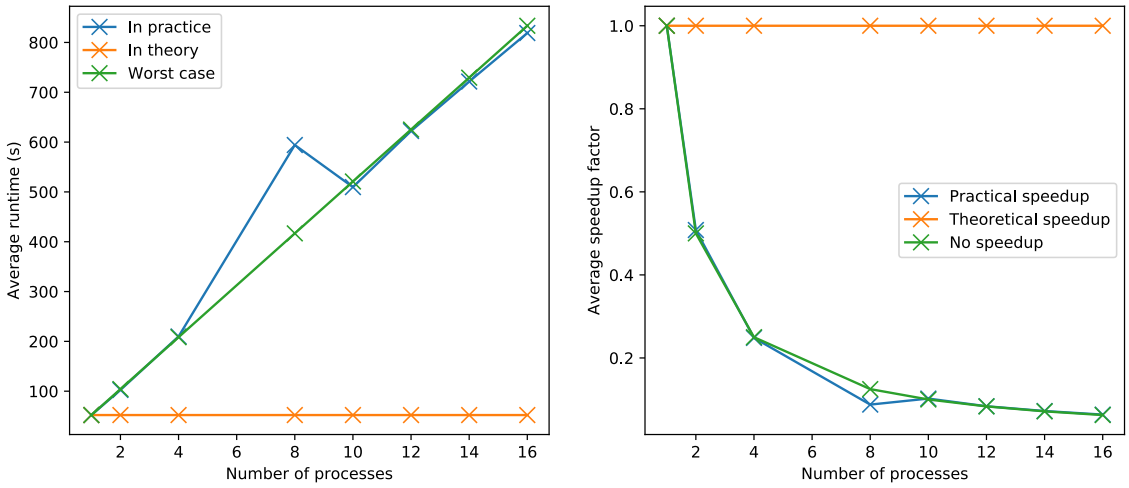


Figure 13: Left) Average runtime with respect to the number of processes, both in theory and applied to our problem. Right) Average speedup with respect to the number of processes, both in theory and applied on our problem.

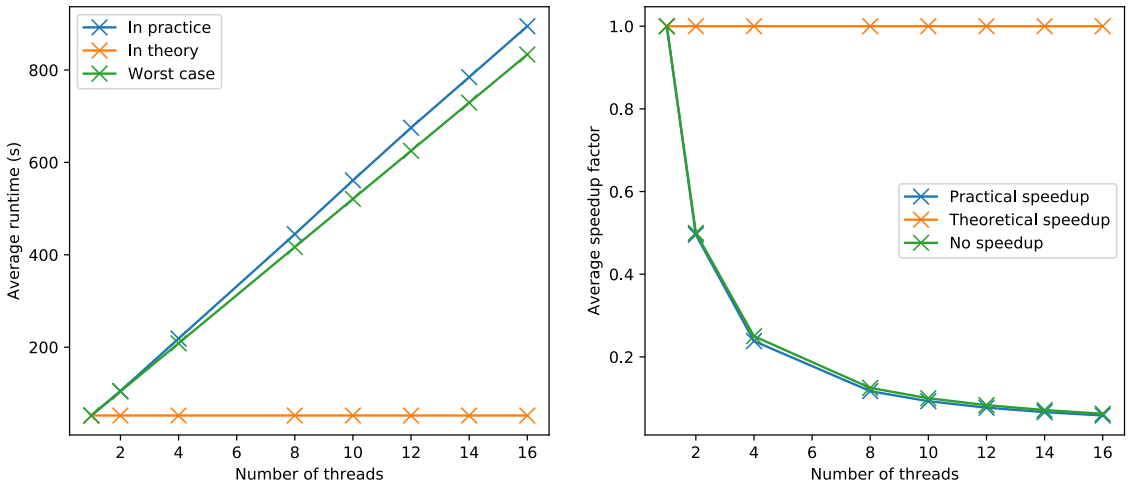


Figure 14: Left) Average runtime with respect to the number of threads, both in theory and applied to our problem. Right) Average speedup with respect to the number of threads, both in theory and applied on our problem.

5 Performance comparison of explicit and implicit

In order to compare the performances of the explicit and the implicit Euler scheme, we will compare the results of the execution of both schemes on two different set of parameters. We will focus on 3 points: the quality of the output, the execution time and the stability of the methods.

We first compare the results on the same setting. First results are shown in the Figure (15). We see that the implicit result is less clean. This may due to the fact that in the implicit scheme a

system needs to be solved and it comes with an approximation. One must however note that this result is dependant on the $r_{threshold}$ value which is here set to 0.01.

In a second time we compare the execution time. For Figure (15), the explicit scheme has run in 2 seconds while the implicit one has run in 2 minutes 18 seconds. We can therefore conclude that solving the linear system with an implicit scheme induces a big overhead in the computational time. However, even-though each iteration of the implicit scheme requires solving an extra set of equations and it being much harder to implement, the stability of the implicit scheme makes it so that one can use a much bigger ΔT, and in the end, the computation is faster than the explicit finite difference scheme. Furthermore, implicit Euler shows better scaling with threads which may be of use when having lots of threads at disposal.

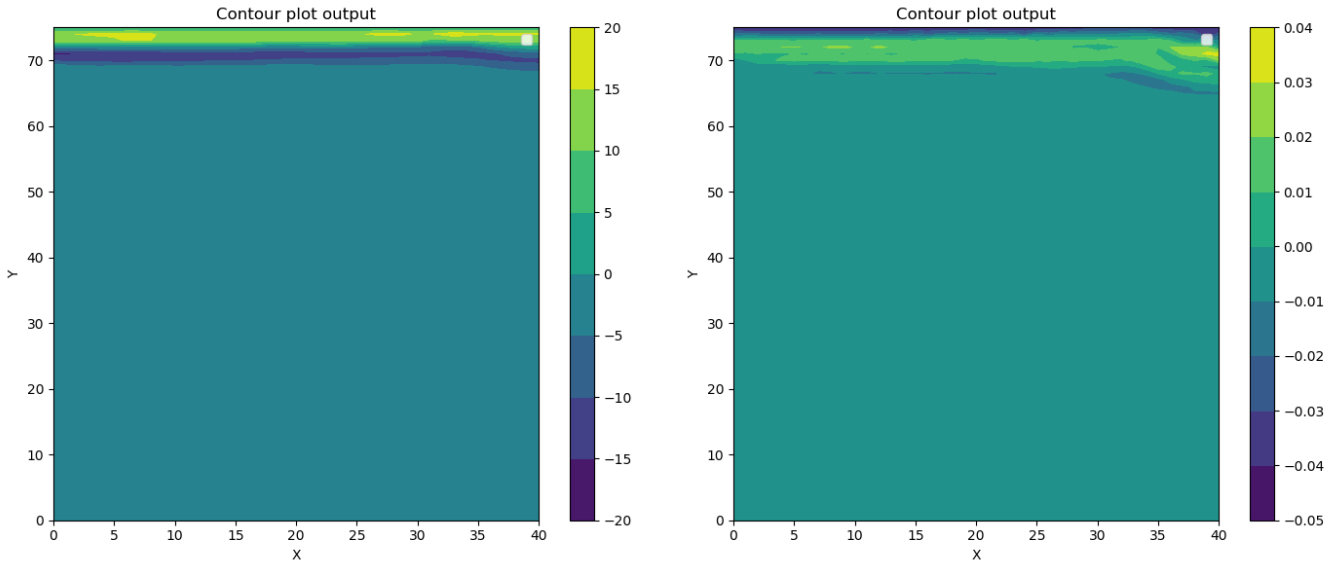


Figure 15: The values for the explicit scheme on the left and the implicit scheme on the right, at the iteration 1050 with parameters $\Delta T = 1$, $\Delta X, \Delta Y = 25000$

6 Conclusion

To conclude, if one would want to speed up the computation of our implemention of the shallow water equations using the explicit finite difference method, we would advise them to use as many processes as they can but limiting the number of threads at 2. For the implicit finite difference scheme, increasing both the processes or the threads yields a near linear improvement in speed with respect to the number of processing elements. Thus, the user is invited to choose the type of element that is most available to them. However, we give no guarantee to how the scaling is for a combination of multiple threads and multiple processes.

References

[cec] Ceci - consortium des Équipements de calcul intensif.

[sha] Discretisation of shallow water equations.