High Performance Calculation

HPC Project : shallow water model

Romain Mekarni
Clément Chouteau

Supervisors:
Wilfried Kirschenmann
Samia Zaidi

May 20, 2018

Contents

Introduction **3**

1 MPI parallelization **3**

1.1 Dependencies 3

1.2 Splitting into tasks 4

1.2.1 Stripes 4

1.2.2 Blocs 5

1.3 Recovering of communication 6

1.4 Exporting results in a file 7

1.4.1 Stripes 7

1.4.2 Blocs 7

1.4.3 Export step 7

1.5 Verification and tests 8

1.6 Performances results 8

2 Hybrid programming with OpenMP **9**

2.1 Performances results 10

2.2 Improvements 10

3 Processor optimization **10**

3.1 Memory access 11

3.2 SIMD with OpenMP 12

3.3 SIMD with compiler GCC 12

Conclusion **13**

List of Figures **14**

Acronyms **14**

Bibliography **14**

Introduction

The shallow water equations are partial differential equations that describe the flow below a pressure surface in a fluid, like a pebble launched in a river. With a lot of calculations, we can determine the physical state at a t instant. In this project we are going to improve performances a lot by parallelizing calculations in order to treat bigger problems in less time.

We start by having the program code for a simple sequential execution that is able to save frames of the successive physical states in a file so we can visualize the shallow water effect in time (figure 1). The parallelized program also has to be able to save results in output files.

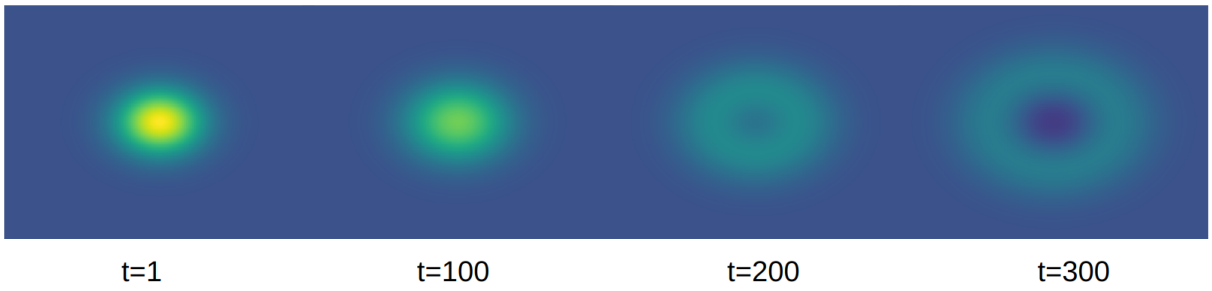

Figure 1: Shallow water simulation example

The project goal is to parallelize theses calculations at different levels : multiple machines, threads, and instructions in processors. We are using standard technologies like MPI, OpenMP and SIMD by mixing them.

1 MPI parallelization

The Message Passing Interface (MPI) library provides functions that help us make different processes executing on different machines by exchanging messages, like data or results after calculations. The first step is to **determine dependencies between each calculations** so each process can have the right data and return the right results to right processes.

1.1 Dependencies

There are six matrices of data : HFIL, UFIL, VFIL and HPHY, UPHY, VPHY. **The problem has spacial and temporal dependencies** represented in the figure 2 :

- $t-1$ results are needed to calculate t state
- each (H/U/V)FIL(t) depends on the correspondents (H/U/V)PHY(t)
- but HPHY(t), UPHY(t), VPHY(t) are independents each other

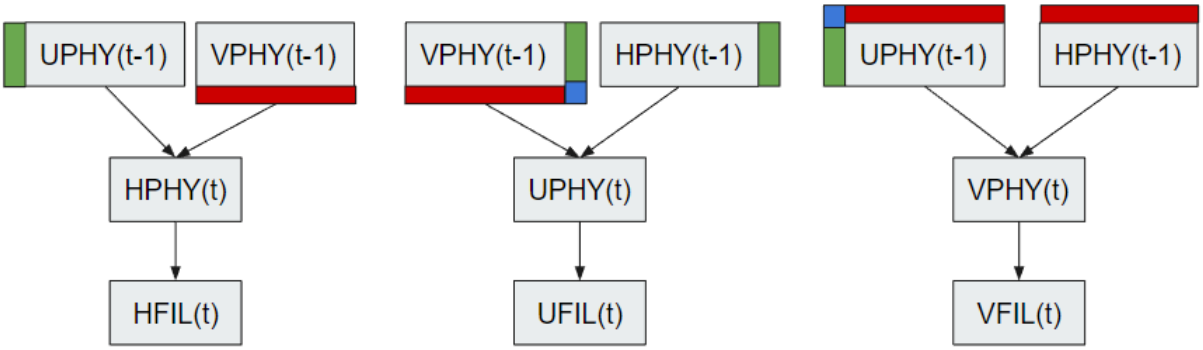

Figure 2: Dependencies between matrices

Because of the temporal dependency, **it requires a double buffering memory** management so we can save new t results without corrupting $t-1$ data needed in computations.

1.2 Splitting into tasks

In order to allocate work to processes, we need to split into smaller tasks. Basically, there are two possible ways to distribute the grid :

- a distribution by stripes, simple.
- a distribution by blocs, more complex.

Each processes will allocate memory only for the task he is willing to do which permits to treat instances of the problem impossible to address on a single computer. We choose to allocate memory by taking into account the extra lines and columns that will be exchanged. This way we don't have to adjust calculations functions : data is stored the same way as it is in the sequential program. We assume that it **is better avoiding to touch the scientific part of the code** where we may lack of knowledge and make mistakes.

1.2.1 Stripes

A breakdown by stripes has the advantage to be simple to implement. In figure 3 we see that we only need to exchange first and last lines of matrices with last and next processes.

Exchanges cost per process, where X and Y are the size of the considered problem :

- data : $X\dfrac{Y}{p} + 2X$
- communications : 6
- data transfer : $6X$

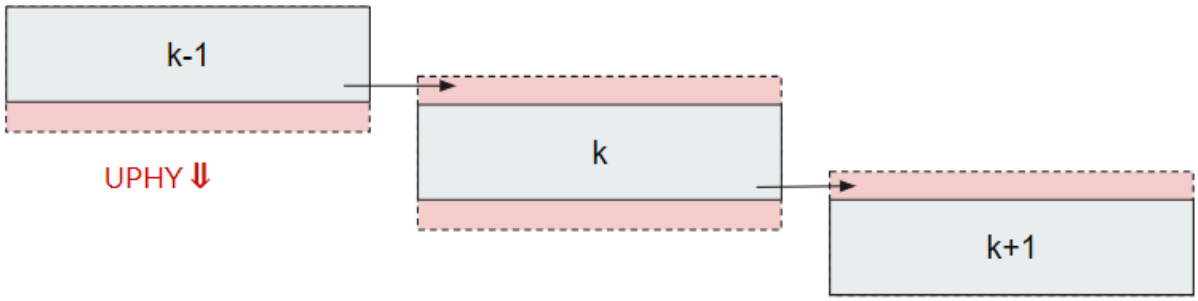

Figure 3: Splitting into stripes

Data transfer $6X$ are constant with the number of processors p. **This inherently sequential fraction f limits the maximal acceleration** which is called Amdahl law

$$S(n,p) \leq \frac{1}{f + (1-f)/p}$$

1.2.2 Blocs

We have the next data flow as shown in the figure 4 :

- lines : HPHY ⇑, UPHY ⇓, VPHY ⇑
- columns : HPHY ⇐, UPHY ⇒, VPHY ⇐
- corners : UPHY ↘, VPHY ↖

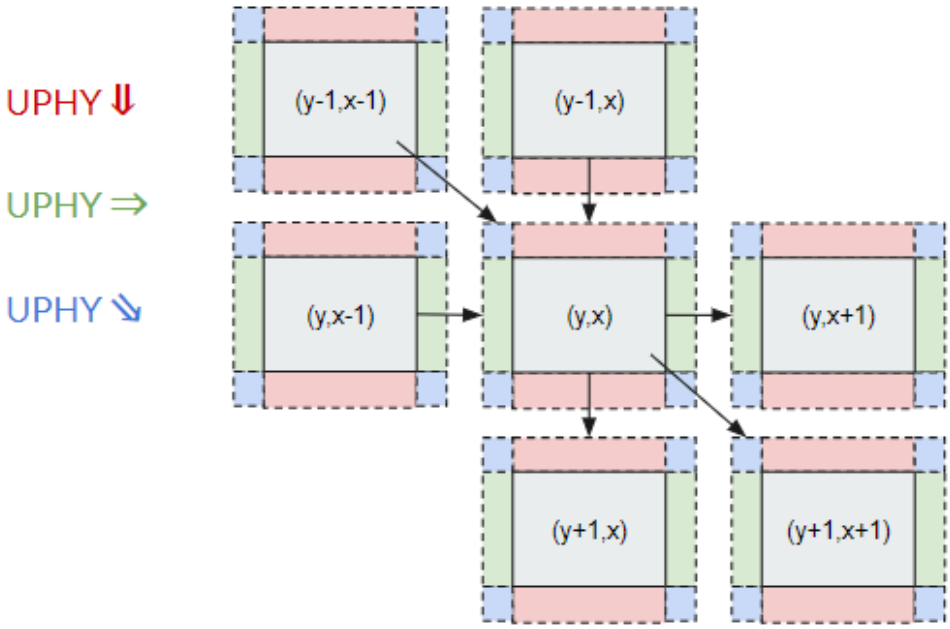

Figure 4: Splitting into blocs for UPHY matrix

Exchanges cost per process, where X and Y are the size of the considered problem :

- data: $(X/q)(Y/q) + 2(X/q) + 2(Y/q)$
- communications: 16
- transfers: $6(X/q) + 6(Y/q) + 4$

The blocks version doesn't suffer from the same flaw as the stripes version. We used a special MPI type `MPI_Type_vector` to exchange blocs columns on edges.

1.3 Recovering of communication

Default MPI communication mode is the blocking synchronous mode. Obviously, **transferring a lot of messages in a large nodes network will slow down calculations**. By recovering communications with computations, we let MPI manage exchanges in the background while our program is continuing it's execution : it is the *asynchronous* non blocking mode.

Because of the spacial dependencies (section 1.1), **we separate calculations in two parts** : interior and exterior (edges) parts. So we can continue the interior calculation (the bigger part) while edges are exchanged between processes. After it may be remains some messages not totally received yet, but it becomes insignificant.
A high level description of the asynchronous algorithm is (figure 5) :

1. $t-1$ grid export
2. Launching $t-1$ edges exchanges
3. Interior bloc t calculations (recovery of communications)
4. Waiting for $t-1$ edges receptions
5. t edges calculations
6. Waiting for $t-1$ edges sendings

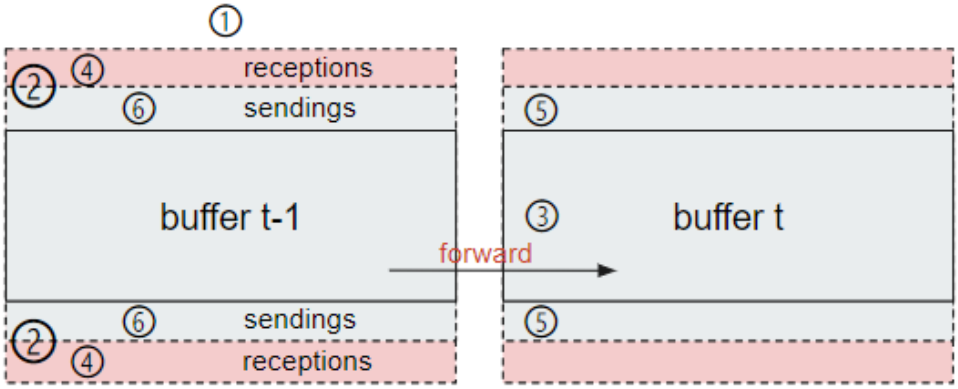

Figure 5: Recovering of communication with calculations

1.4 Exporting results in a file

MPI provides us functions to parallelize input and ouput [1]. We can open and close a file, read and write in both synchronous and asynchronous modes. However, the best tools are `MPI_File_set_view` and special MPI types. Theses generic methods can be exceeded by special case developed methods but takes time to code.

During the development, we experienced some unpredictable bugs due to the NFS server used in our cluster. While exporting was working fine locally on one machine (with multiple nodes), we had random troubles to reconstruct images with networked multiple machines. We have consulted other groups and they also lost a lot of time trying to resolve it.

1.4.1 Stripes

`MPI_File_set_view` changes process's view of data in file. With a data type and an offset, **we can specify a write displacement** so each process can write it's results at correct place in file (figure 6).

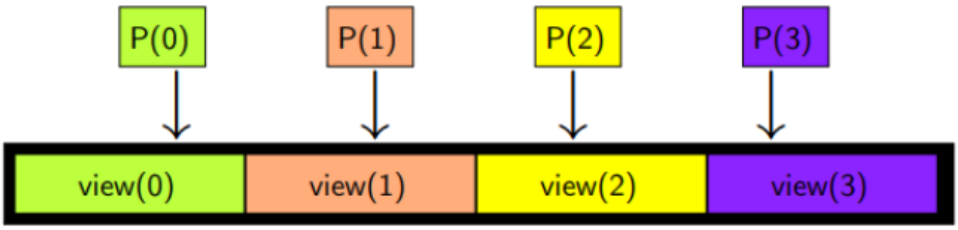

Figure 6: MPI set view for parallel writing in stripes mode

1.4.2 Blocs

Because we integrated edges in the main memory buffer, we can't just consider a sub-array special type (figure 7) for the file and then specify the address memory. Memory is not contiguous the same way the file is because of edges. **We also need to create a special type for the memory buffer**.

With a `MPI_Type_contiguous` we consider a type for one line of sub-array in memory. Then, with `MPI_Type_create_resized` we specify offset between 2 lines (which is size of a line + 2). This way, the sub-array special file type will read memory by skipping non wanted edges.

1.4.3 Export step

Exporting the results in a file may becomes impossible if the problem is big. The file quickly grows and network may be saturated. We implemented a new program

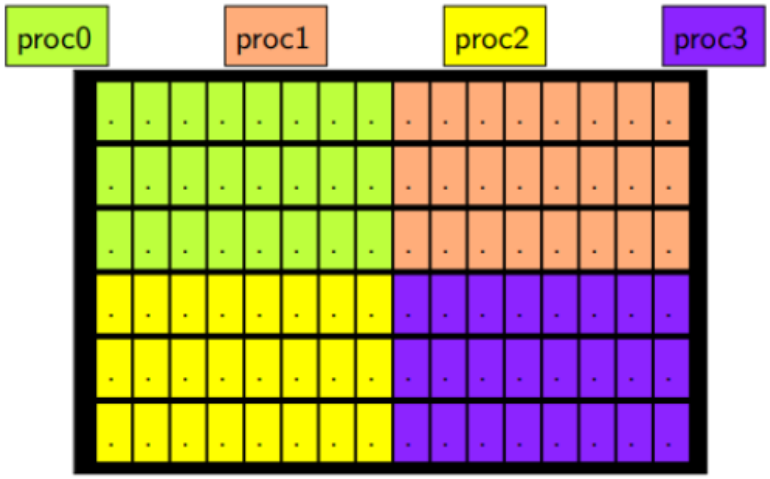

Figure 7: MPI special types for parallel writing in blocs mode

parameter `--export-step x` which indicates to save results every x frames. It also has the benefit to reduce the final file size.

```
1.26s make NODES=16 MODE=--block --async -x 512 -y 512 -t 40
0.49s make NODES=16 MODE=--block --async -x 512 -y 512 -t 80 --export-step 10
```

1.5 Verification and tests

We made choices to avoid as much as possible modifying the scientific specific code. But it remains necessary to test results, particularly the file export option. In order to control results accuracy, we used a script which removes previously generated files, run the test, save the last image of the result file and then use the `compare` program to reveal pixels that differ. Even if the images seems to be equals, we see a lot of red pixels in the figure 8 which indicates that images are not equals.

Figure 8: Example of calculations errors detected

1.6 Performances results

For the reference test, case 2, `-x 8192 -y 8192 -t 20` on 16 nodes, we have :

```
3.99 (75%) make NODES=16 -C parallel test2
3.95 (75%) make NODES=16 MODE=--async -C parallel test2
3.09 (96%) make NODES=16 MODE=--block -C parallel test2
3.06 (98%) make NODES=16 MODE=--block --async -C parallel test2
```

Which is good because we are approaching a **linear acceleration** (98%). We assume that bloc mode is better because it uses cache more efficiently. However, we don't see any gain from the calculation recovery. So we tried (at night) with a bigger test : `-x 32768 -y 32768 -t 40` on 64 nodes :

```
49.33 (47%) make NODES=64 -C parallel big_test
31.76 (74%) make NODES=64 MODE=--async -C parallel big_test
35.00 (64%) make NODES=64 MODE=--block -C parallel big_test
26.79 (87%) make NODES=64 MODE=--block --async -C parallel big_test
```

And now it appears that recovery really improves performances. To verify it, we added timers in the program to calculate how much time is devoted to messages exchanges or computations. For example, in bloc mode 64 nodes in the `big_test` :

```
Synchronous mode
       Message exchange : 10.23
       Calculations : 23.12
Asynchonous mode
       Message exchange : 0.79  <-- nodes almost never wait for communications
       Calculations : 24.63     <-- calculation recovery
```

2 Hybrid programming with OpenMP

MPI is mandatory to use *distributed memory* on machines to do bigger simulation but **communications can slow down performances in a too large network**. OpenMP allows us to simply have threads opened for the main `for` loop computations with only a few compiler instructions (C pragmas), this is called hybrid parallel programming.

The parallel hybrid programming consists in mixing multiple paradigms of parallel programming (shared memory) in order to take advantages of the different approaches [2]. In particular, we might want **to reduce communications and to exploit hyper-threading**.

2.1 Performances results

MPI is using shared memory to exchange messages between nodes in the same computer. This is really fast and it does compete with OpenMP. This can explain the results we obtained by comparing a MPI pur version and a hybrid one, with the exact same number of CPU core used (threads actually) :

```
26.79 make NODES=64 MODE=--block --async -C parallel big_test
50.36 make NODES=16 MODE=--block --async --hybrid -C parallel big_test
      (4 threads per node)
```

But if we enable more threads per each nodes, we acknowledge a gain :

```
3.06 make NODES=16 MODE=--block --async -C parallel test2
2.01 make NODES=16 MODE=--block --async --hybrid -C parallel test2
     (whith 4 threads per node)
```

That makes sense if we consider that MPI is able to parallelize the whole program (and use shared memory when possible) where as OpenMP only parallelize parts. Opening et closing threads has also a sequential constant cost.

2.2 Improvements

In opening, we might study more precisely how MPI manage it's communications, and if we could improve with OpenMP by considering tasks.

We can consider a task for waiting a single reception request and calculate the corresponding edge. This way, tasks are assigned to available threads. Moreover, we don't need to wait for all receptions before calculating edges of the matrix.

Because they is a fixed number of receptions and edges computations are residual in front of bloc interior calculations, we assumed that the time needed to implement this method did not justify the gain obtained. But it would be interesting exploring that kind of optimization to see if this is really relevant or not.

3 Processor optimization

Single Instruction Multiple Data (SIMD) is a processor level optimization that allows similar instructions in a program to be vectored and executed at the same time. The program will be executed on 128 or 256 bits large processor registers so we can compute four 64 bits results with one instruction, as shown in the figure 10.

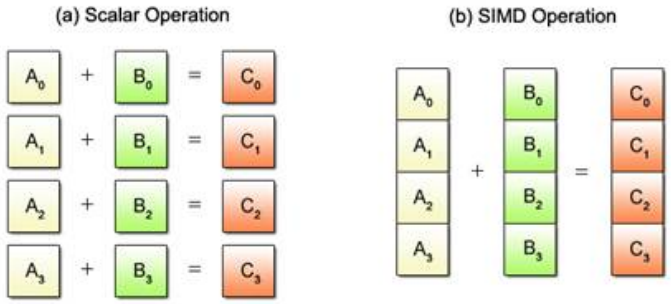

Figure 9: SIMD allows us to calculate multiple results in one single instruction

3.1 Memory access

Nowadays memory access is more likely to be the bottleneck. To help the processor access data quickly, it has been provided with a memory called cache, really fast because near the chip. When a memory access is requested, the processor read an entire line of memory and store it in the cache, using fast spacial and time location algorithms to find data without accessing the RAM memory everytime.

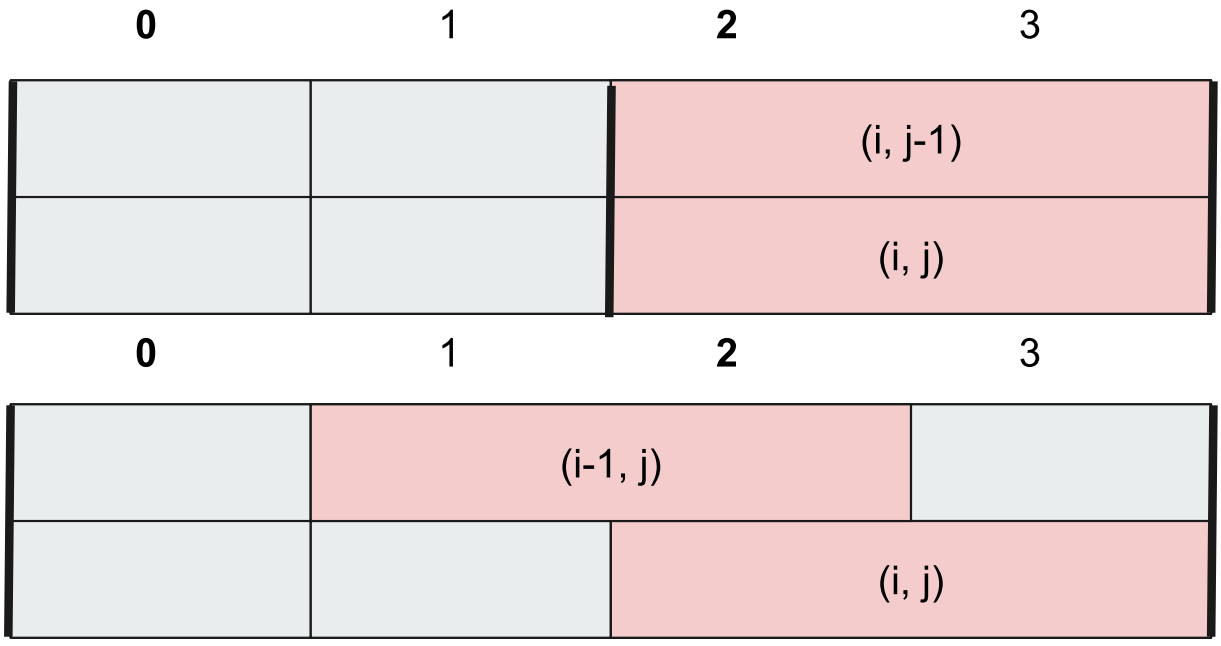

Figure 10: Loading at position `(i-1, j)` will always be unaligned.

Initially, matrices are array stored in memory column by column with an access (line, column). However, to help the processor looking in the cache instead of memory, we should access the matrices in the same order it is stored. **We choose to invert the array accesses and loops**.

By comparing with sequential versions only, we have a **performance gain of** 230% which demonstrates how a better cache using can really improve performances [3].

The memory accesses to the matrices are often done with offsets of ± 1, and are then

fundamentally *unaligned* (see Figure 10), this doesn't prevent the use of SIMD but can reduce its performance.

The floating point computations and memory accesses are in the same magnitude but there is a lot of computations per access (while beeing constant), so vectorization is not an obvious choise but should not be impossible from this point of view.

3.2 SIMD with OpenMP

We tried to use SIMD with OpenMP but we did not manage to gain performance. After researching on the Internet, we found out that SIMD by OpenMP is still immature compare to GCC auto vectorization.

3.3 SIMD with compiler GCC

With our improved understanding of the problem gained by our analysis of the code, we could easily guess that **the main obstacle in enabling SIMD instructions was the presence of *conditionnal branches*** between calculations.

However these conditions are border checks and waiting $t > 2$, if we wait $t > 2$ and only compute the inner rectangle of the simulation the conditions can be omitted. We created specialized versions of the calculation functions with conditions removed, we also need some transformations in the loop calling these functions.

Another obstacle for performance was the need to *inline* functions, the compiler can not guess that most of the computing time is done in the functions that compute the next `uPhy`, `vPhy`, `hPhy`, `uFil`, `vFil`, `hFil`. The inlinings can allow the compiler to simplify and reorder the code.

We also added the `__restrict__` keyword on the pointers to the matrices to allow the compiler to understand that the matrices are at disjoint memory locations and computations are independant, therefore allowing it to **reorder load and stores**.

In the integration of the SIMD code in the parallel code the compiler refused to vectorize the loops, we had to add `#pragma GCC ivdep`, indicating that the loop carries no dependency.

In order to be able to see the SIMD speedup we had to compute simulations with more steps, so that the initialization time were a small part of the execution time.

We show only the computations time, in order to better appreciate SIMD speedup.

Version auto vectorized by GCC:

```
55.24s for test -y 8192 -x 8192 -t 100 (173%)
15.65s for test -y 4096 -x 4096 -t 100 (197%)
```

Same version with only SIMD disabled, `-fno-tree-vectorize`:

```
95.51s for test -y 8192 -x 8192 -t 100
30.83 for test -y 4096 -x 4096 -t 100
```

Theses gains confirm the SIMD vectorization.

Conclusion

Because of the results, we can conclude that the parallelization of the program is a success. It is always possible to improve by writing specific code but it is longer and difficult to implement.

```
shallow -x 8192 -y 8192 -t 20 :
47.83 make -C sequentiel test2
1.58 (186%) make NODES=16 MODE=--block --async --hybrid -C parallel test2
```

With 47.83 seconds for sequential execution and only 1.58 seconds for final parallelized execution, the parallel efficiency is good (factor 2 is for simd) :

$$E(n,p) = \frac{S(n,p)}{p} = \frac{47.83/1.58}{2*16} = 0.946 \rightarrow 94.6\%$$

This has to be mitigated by the fact that simd required modifications of the sequential program that we did not spread to the sequential version.

For this to be possible, we used combined technologies to gain on different aspects of the execution : MPI, OpenMP, SIMD and other optimization.

It appears that we can really improve the sequential program by just using simple generic functions provided by theses libraries. We assume in a close future that programs will be able to automatically parallelize other programs.

Moreover, MPI is really helping building parallelized applications, especially with it's special types and built-in communication API. With the using of well named explicit variables, we manage to build a working program without a debug capability. Our results have been tested and verified.

Finally, we really appreciated looking for possible optimizations and it is a great pleasure obtaining good performances.

List of Figures

1 Shallow water simulation example . . . 3
2 Dependencies between matrices . . . 4
3 Splitting into stripes . . . 5
4 Splitting into blocs for UPHY matrix . . . 5
5 Recovering of communication with calculations . . . 6
6 MPI set view for parallel writing in stripes mode . . . 7
7 MPI special types for parallel writing in blocs mode . . . 8
8 Example of calculations errors detected . . . 8
9 SIMD allows us to calculate multiple results in one single instruction . . . 11
10 Loading at position `(i-1, j)` will always be unaligned. . . . 11

Acronyms

MPI Message Passing Interface. 2, 5, 6, 8, 9, 12

SIMD Single Instruction Multiple Data. 2, 9–12

References

[1] Ramses Van Zon. *MPI-IO*. 2013. URL: `https://wiki.scinet.utoronto.ca/wiki/images/8/88/Parallel_io_course_mpi_io.pdf` (visited on 05/09/2018).

[2] P. Wautelet P.-Fr. Lavallée. *Programmation hybride MPI-OpenMP*. 2017. URL: `http://www.idris.fr/media/formations/hybride/form_hybride.pdf` (visited on 05/09/2018).

[3] Ulrich Drepper. "What every programmer should know about memory". In: (2007).