PROJECT PROPOSAL PHPC-2018

A High Performance Implementation of Shallow Water Wave Equation with a finite volume solver : Tsunamis Simulation at Scale

Principal investigator (PI)	Arnaud Pannatier
Institution	EPFL-SCITAS
Address	Station 1, CH-1015 LAUSANNE
Involved researchers	Only PI
Date of submission	May 16, 2018
Expected end of project	July 6, 2018
Target machine	Mount Everest
Proposed acronym	TSUNAMI

Abstract

This project proposes a way to increase the performance of a sequential code using classical parallelism technic. It is based on a Matlab code of *Nicolas Richart* that simulates the evolution of a tsunami. The application is coded in C++ and will use the industrial standards MPI and OpenMP. The purpose of this report is to describe the state of the current project and to describe how the sequential code will parallelized.

1 Scientific Background

A parallel iterative finite difference method for solving the 2D elliptic PDE Poisson's equation on a distributed system using Message Passing Interface (MPI) and OpenMP is presented. This method is based on a domain decomposition where the global 2D domain is divided into multiple sub-domains using horizontal axis. The number of subdomains is specified by the number of processes. The Shallow Water Wave Equation is solved by explicit iterative schemes. The global error is shared by all processes.

1.1 Introducing the Model

The shallow water wave equation is a non-linear hyperbolic system of coupled partial differential equations often used to model various wave phenomenons. Simulation of ocean waves, river flows, hydraulic engineering and atmospheric modeling are among the many areas of application. The researcher has used the two dimensional version of the equations along with a right hand side source term as his model

$$\begin{cases} h_t + (hu)_x + (hv)_y = 0 \\ (hu)_t + (hu^2 + \frac{1}{2}gh^2)_x + (huv)_y = -ghz_x \\ (hv)_t + (huv)_x + (hv^2 + \frac{1}{2}gh^2)_y = -ghz_y \end{cases} \tag{1}$$

In the above, $h := h(x, y, t)$ denotes water height, $u := u(x, y, t)$ and $v := v(x, y, t)$ water velocity in x and y direction respectively, $z := z(x, y)$ topography and g= $9.82 m/s^2$ the gravitational constant.

1.2 Introducing the Numerical Scheme

Finite volume schemes are a popular approach for computing an approximate solution to hyperbolic equations as the underlying physics is represented in a natural way. Let $I_{i,j} = [x_{i-1}, x_{i+1}] \times [y_{j-1}, y_{j+1}]$

define a structured rectangular uniform mesh. In a finite-volume scheme, one seeks to find the cell average $\bar{q}_{i,j}(t_n)$ that approximates $q(x_i, y_j, t_n)$ at every cell $I_{i,j}$ for a given time-step t_n in the sense

$$\bar{q}_{i,j}(t_n) = \frac{1}{\Delta x \Delta y} \int_{I_{i,j}} q(x_i, y_j, t_n) dydx \tag{2}$$

The researcher has used the Lax-Friedrichs method to discretize the problem. The method is explicit meaning that the solution $\bar{q}^{n+1}_{i,j}$ for all i, j at timestep t_{n+1} may be computed directly from the solution at the previous timestep $\bar{q}^{n}_{i,j}$ without solving any system of equations using the following three equations

$$\begin{aligned}
\bar{h}^{n+1}_{i,j} =& \frac{1}{4}[\bar{h}^n_{i,j-1} + \bar{h}^n_{i,j+1} + \bar{h}^n_{i-1,j} + \bar{h}^n_{i+1,j}] && (3)\\
&+ \frac{\Delta t^n}{2\Delta x}[\bar{hu}^n_{i,j-1} - \bar{hu}^n_{i,j+1} + (hv)^n_{i,j-1} - (hv)^n_{i,j+1}] && (4)\\
\bar{hu}^{n+1}_{i,j} =& \frac{1}{4}[\bar{hu}^n_{i,j-1} + \bar{hu}^n_{i,j+1} + \bar{hu}^n_{i-1,j} + \bar{hu}^n_{i+1,j}] - \Delta t^n g h^{n+1} \partial_x(z) && (5)\\
&+ \frac{\Delta t^n}{2\Delta x}[\frac{(\bar{hu}^n_{i,j-1})^2}{h^n_{i,j-1}} + \frac{1}{2}g(h)^n_{i,j-1} - \frac{(\bar{hu}^n_{i,j+1})^2}{h^n_{i,j+1}} - \frac{1}{2}g\bar{h}^n_{i,j+1}] && (6)
\end{aligned}$$

$\bar{hv}^{n+1}_{i,j}$ is calculated in a similar manner.

After each simulation time-step, we need to compute a new time-step length. This is needed to make sure that the CFL condition is satisfied. The time-step Δt^n

$$\Delta t^n = \min_{i,j} \frac{\Delta x}{\sqrt{2}\nu^n_{i,j}} \tag{7}$$

is found computing first ν^n using

$$\nu^n_{i,j} = \sqrt{\max_{i,j}\left(\left|\frac{\bar{hu}^n_{i,j}}{\bar{h}^n_{i,j}} + g\bar{h}^n_{i,j}\right|, \left|\frac{\bar{hu}^n_{i,j}}{\bar{h}^n_{i,j}} - g\bar{h}^n_{i,j}\right|\right)^2 + \max_{i,j}\left(\left|\frac{\bar{hv}^n_{i,j}}{\bar{h}^n_{i,j}} + g\bar{h}^n_{i,j}\right|, \left|\frac{\bar{hv}^n_{i,j}}{\bar{h}^n_{i,j}} - g\bar{h}^n_{i,j}\right|\right)^2} \tag{8}$$

After computing a new time-step $\bar{q}^{n+1}_{i,j}$, cells that are below a certain water threshold are made inactive :

$$\bar{h}^{n+1}_{i,i} \leq 0 \rightarrow \bar{h}^{n+1}_{i,i} = 10^{-5} \tag{9}$$

$$\bar{h}^{n+1}_{i,i} \leq 10^{-4} \rightarrow hu^{n+1}_{i,i} = hv^{n+1}_{i,i} = 0 \tag{10}$$

2 Implementations

The application is implemented in C++. We will investigate two implementations of the same solver :

- A shared memory version using the OpenMP paradigm [1]

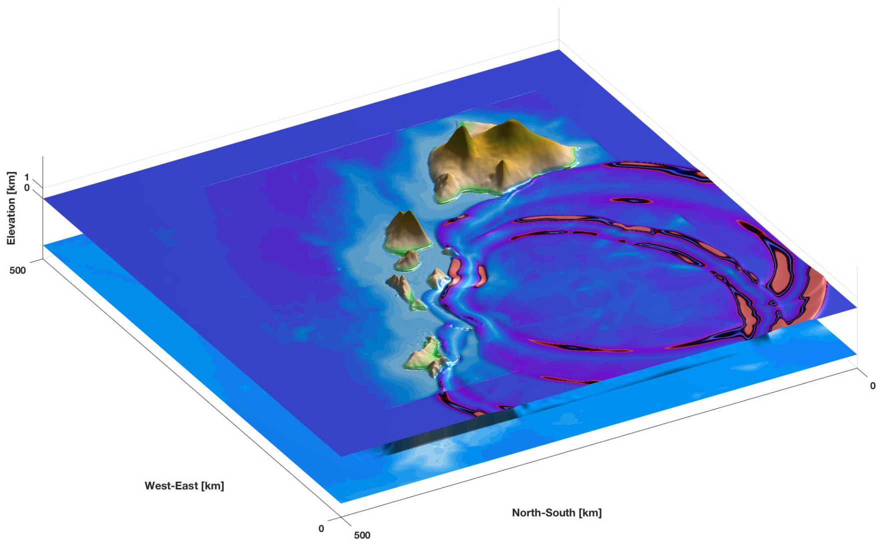

Figure 1: Simulation of a Tsunami with $N = 4001$, $T_{end} = 0.2$

- A distributed memory version using the MPI library [2]

The code has been be fully debugged using the `gdb` debugger. The `Valgrind` tool has been used to remove all the memory leaks. [Tab.] The profiling tool `gprof` has been applied on the sequential and the parallel case.[Tab.]

2.1 Description of the Code

The code is arranged in the following way. The algorithm will compute the evolution of the height of the wave $H(x, y)$, based on the knowledge of the topography and the evolution of the speed of the wave in the direction x and y. In order to start the computation, the algorithm first read the initial conditions for the different values (H,HU,HV, Zdx, Zdy) in binary files. The duration of the evolution T_{end} is specified by the user. While this time is not reached, the algorithm will compute the next variable time step and the evolution of the height and the speed : H,HU,HV. The update of the variables only needs information of the direct neighbor of a point (x,y). When the simulation is over the height of the wave at the final time step is stored in a file.

2.2 Parallelization

2.2.1 Optimization on one core : Vectorization

TODO : BLABLABLA

2.2.2 MPI implementation

In order to translate the sequential conde in a parallel fashion, the following design is used : as `C++` is a row major language, the grid is splited between process in a block of $\frac{N}{P}$ complete

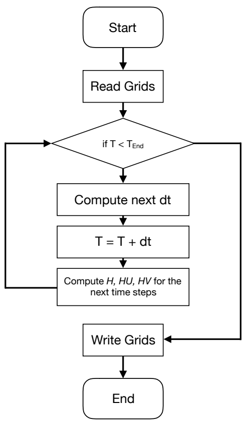

Figure 2: Flowchart of the sequential program

rows [FIG. 3]. Some other design have been analyzed but seemed less appropriate for this case : splitting using complete columns (the translated approach) was not a good solution as the langage uses a row-major convention. Distributing the data into squares was also envisaged, but again as the langage that is used is row-major it is preferable for a process to have most of its points arranged in row in order to use vectorization, so this alternative was abandonned. Furthermore it was more complicated to organize the ghost cells in the square architecture. Splitting into $\frac{N}{P}$ complete rows was the more natural way to implement the data distribution and it natural to organize the ghosts cells.

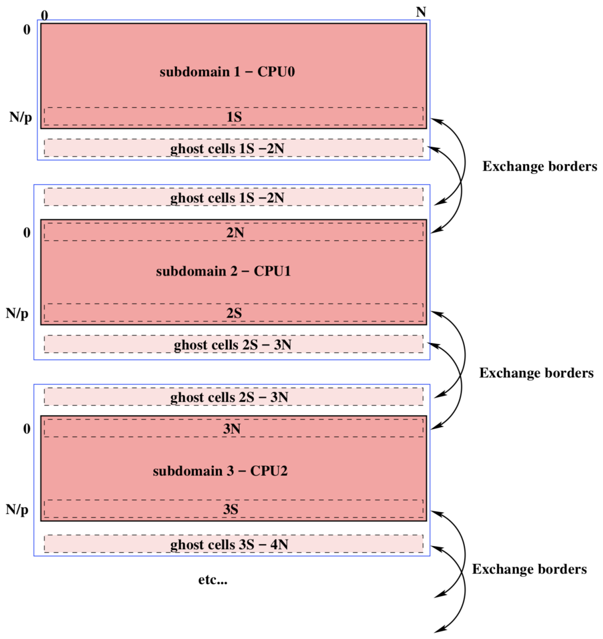

Figure 3: Data distribution strategy for the MPI approach. [4]

The reader, writer and time step computation as well as the update of the height and the

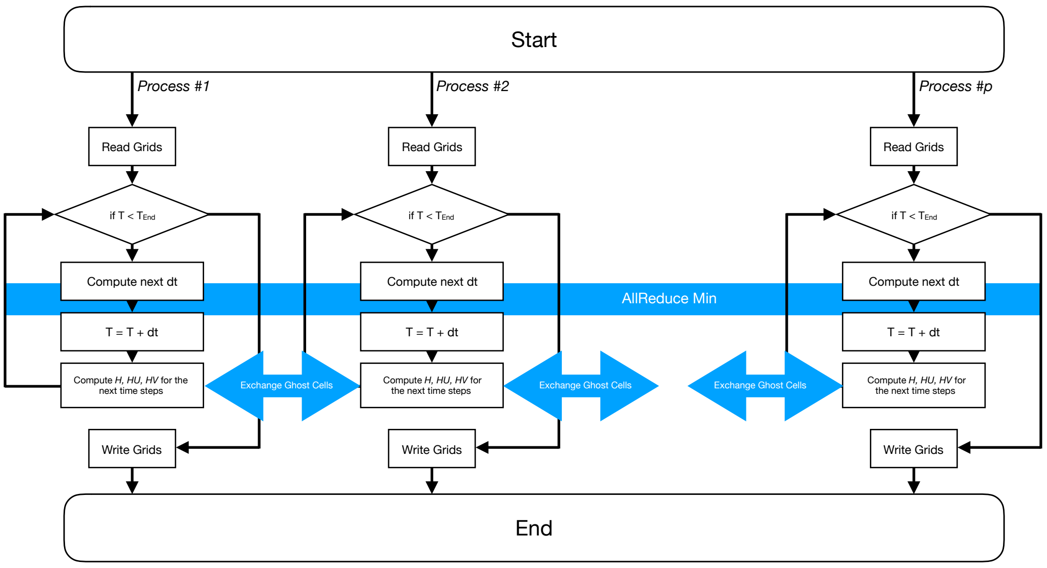

Figure 4: Flowchart of the parallel program using MPI

speed H, HU, HV are implemented.

The parallelization of the reader and writer will make use of the Parallel I/O of MPI. As `Matlab` is a column-major langage and, in the contrary, `C++` is a row-major language. A column-major reader and writer have been developed in order to work with the same files.

For the parallelization of the main loop, we need first to compute at each step the next time step and then to compute the future values of each grids H, HU, HV. Computing the next time step implies that the max over a grid should be found. Each process will then return the max over his subgrid and the final max will be computed at the end, this final computation will be done on a single node. The parallelization of the update of H, HU, HV will be approached in a similar manner. The grid will be split as before. Before and after each subgrid (except the first one and the last one) a row of ghost cells will be added, in order to allow the computation. Most of the code will therefore be parallelized.

3 Prior results

Translating the code from `Matlab` to `C++` allows us to compute the solution approximately 5 times faster using vectorization. This performance will be improved drastically using parallelism technic.

The results were compared with Matlab for the case $n = 2001$ and $n = 4001$. It is giving the same results. Some error can be seen in the place where the height of the wave varies the most due to numerical approximation [FIG.] . For the case of $n = 8001$, no comparaison with Matlab were done due to the too large amount of time needed to run the code on Matlab.

3.1 Strong scaling

The speed up of a precise parallelism problem depends mostly on the percentage of sequential code that can be parallelized. In the case of this problem, most of the code can be paral-

	Operation	Complexity
Computation	Reading	$\mathcal{O}\left(\frac{N^2}{p}\right)$
	Compute Δt	$\mathcal{O}\left(\frac{N^2}{p} * n_{\Delta t}\right)$
	Compute H, HU, HV	$\mathcal{O}\left(\frac{N^2}{p} * n_{\Delta t}\right)$
	Writing	$\mathcal{O}\left(\frac{N^2}{p}\right)$
Communication	Reading	0
	Compute Δt	$\mathcal{O}\left(\log(p) * n_{\Delta t}\right)$
	Compute H, HU, HV	$\mathcal{O}\left(N * n_{\Delta t}\right)$
	Writing	0

Table 1: Computation and communication complexities, p is the number of process, N is the number of points on the side of the grid, $n\Delta t$ is the number of timesteps required to compute the simulation.

	Matlab [s]	C++ [s]
2001x2001	1233.068	226.741
4001x4001	10625.835	2003.880

Table 2: Total time in seconds for a run on a 3.1 GHz Intel i5

lelized. A first estimation can be deduced from the profiling table for the grid 2001×2001 [TAB. 3]. Summing up the three main functions that can be parallelized (`compute_step, compute_mu_and_set_dt` and `imposeTolerances`) gives a value of 98.06% of code parallelizable. The speed up is given by Amdahl's Law :

$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}} \tag{11}$$

Where α is the fraction of non-parallelizable code, in this case 1.94%, and p is the number of cores that is used to compute the simulation.

Computing the speed up using strong scaling for this problem gives the following curve [FIG. 7]. This theoretical result will be compared with the real one when the problem is parallelized.

3.2 Weak scaling

The weak scaling is another way to measure the speed up. The main difference with the strong scaling is that in this case, the size of the problem can change. It is given by Gustaon's law :

$$S_p = p - \beta(n)(p-1) \tag{12}$$

Where $\beta(n)$ is the fraction of non-parallelizable code, and p is the number of cores that is used to compute the simulation. We have the information on three different setup in the projet. Corresponding to the grid 2001×2001, 4001×4001, 8001×8001. As our approach will split the grid only by row, we can expect that the workload for each processor to increase as n for the computation on a grid. Furthermore, adding more precision will reduce the Δx leading to smaller Δt. We can expect this to increase the workload by computer in an order of n as well. Theses hypotheses and the exact speed up will be computed in the final report.

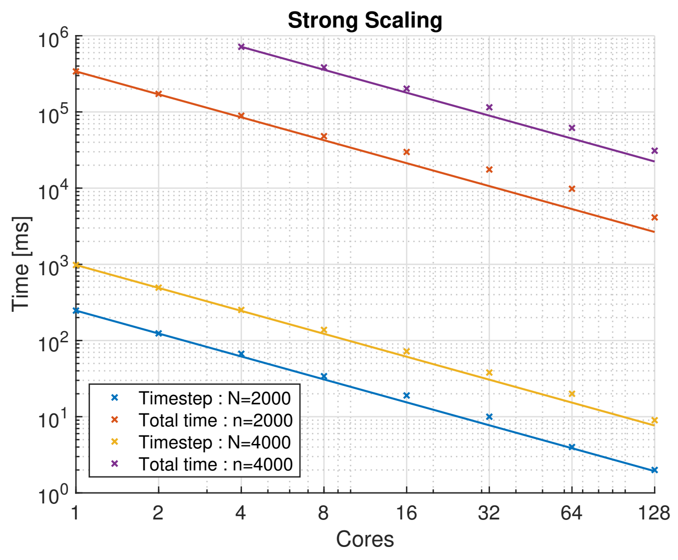

Figure 5: Strong Scaling on Deneb2 for the case of $N = 2001, 4001, 8001$, the total time and the time for a step are displayed.

4 Resources budget

`This part has not been modified but will be completed for the final report`

In order to fulfill the requirements of our project, we present hereafter the resource budget.

4.1 Computing Power

`!! FIXME !!`

4.2 Raw storage

`!! FIXME !!`

4.3 Grand Total

Total number of requested cores	256 - 512
Minimum total memory	`!! FIXME !!`
Maximum total memory	`!! FIXME !!`
Temporary disk space for a single run	`!! FIXME !!`
Permanent disk space for the entire project	`!! FIXME !!`
Communications	Pure MPI
License	own code (BSD)
Code publicly available ?	Yes [5]
Library requirements	MPI
Architectures where code ran	Intel Xeon

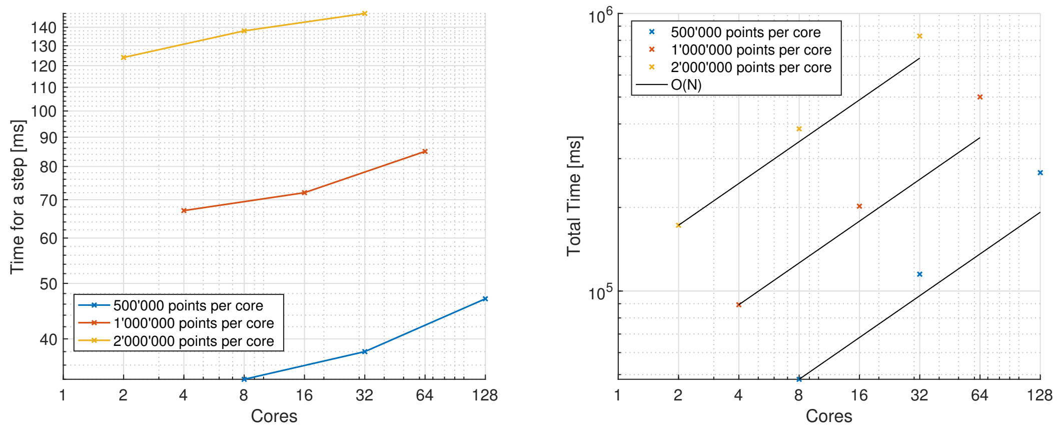

Figure 6: Weak Scaling on Deneb2 for the case of 500'000, 1'000'000 and 2'000'000 of points per core, the total time and the time for a step are displayed.

5 Scientific outcome

This work will lead to a fast implementation for simulation of Tsunamis. It will allow to do impressive simulations without waiting too long. It will also learn to the principal investigator how to parallelize code in a good and efficient way.

References

[1] Dagum L. and Menon R.,*OpenMP: An Industry-Standard API for Shared-Memory Programming*, IEEE Computational Science & Engineering, Volume 5 Issue 1, pp 46-55, January 1998

[2] The MPI Forum, *MPI: A Message-Passing Interface Standard*, Technical Report, 1994

[3] Scientific IT and Application Support, `http://scitas.epfl.ch`, 2015

[4] Keller Vincent - Rezzonico Vittoria, *PHPC Course - MATH-454 - EPFL*, `http://edu.epfl.ch/coursebook/fr/parallel-and-high-performance-computing-MATH-454` , 2018

[5] Arnaud Pannatier *Project repository on c4sciences*, `https://c4science.ch/source/phpctsunamiproject/`, 2018

A Profiling

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
76.47	113.37	113.37	352	322.07	335.59	`Simulation::compute_step`
19.19	141.82	28.45	352	80.82	80.82	`Simulation::compute_mu_and_set_dt`
2.40	145.37	3.55	1056	3.36	3.36	`Grid::imposeTolerances`
1.58	147.71	2.34	4219780580	0.00	0.0	`DoubleBuffer::current`
0.30	148.16	0.45	3	150.04	150.04	`DoubleBuffer::DoubleBuffer`
0.04	148.22	0.06	5	12.00	12.00	`Reader::readGridFromFile`
0.03	148.27	0.05	1056	0.05	0.05	`Grid::applyBoundaryConditions`
0.01	148.29	0.02	1	20.01	20.01	`Reader::writeGridInFile`
0.00	148.29	0.00	2115	0.00	0.00	`DoubleBuffer::old`
0.00	148.29	0.00	1056	0.00	0.00	`DoubleBuffer::swap`

Table 3: Profiling on Deneb1