HPC Shallow Water project : Theoretical Analysis

Antoine Hoffmann
Parallel and High Performance Computing MATH-454, EPFL
E-mail: antoine.hoffmann@epfl.ch

May 16, 2018

1 Introduction

This project consist to parallelize a given finite volume solver MATLAB code which simulates the propagation of a Tsunami on the scale of an island in the shallow water modeling framework.

2 Presentation of the sequential C++ code

2.1 Sequential Algorithm

The MATLAB code has been firstly translated into a C++ sequential code. The main work consisted mainly in a structural reorganization of the algorithm by defining a collection of C++ functions. Those functions where written in a second C++ file linked to the main program with a hierarchical makefile in view to increase the readability and analysis of performance. The algorithm and the finite volume solver did not need to be deeply changed and one can read a pseudo code presentation of the sequential C++ implementation at Algorithm 1.

Algorithm 1 Sequential C++ Finite Volume Solver

n_x = 1D number of grid cells on one side of the domain
N = Total number of grid cells
dx = Grid spacening in [km]
Allocating $(H, HU, HV, Z_{dx}, Z_{dy}, H_t, HU_t, HV_t)$ ▷ Memory allocation
$(Z_{dx}, Z_{dy}) \Leftarrow$ Topology data file ▷ Load gradients of the topology from binary data file
$(H, HU, HV) \Leftarrow$ Initial state data file ▷ Load initial state of the height and the velocities of the water
while $T < T_{\text{end}}$ **do** ▷ Simulating until a given time T_{end}
  $dt \Leftarrow$ new_dt(H, HU, HV, dx, N) ▷ Updating dt from finite volume scheme
  if $T + dt > T_{\text{end}}$ **then**
    $dt \Leftarrow T_{\text{end}} - T$ ▷ To stop exactly at T_{end}
  $(H_t, HU_t, HV_t) \Leftarrow (H, HU, HV)$ ▷ Copy current state to temporary variables
  $(H_t, HU_t, HV_t) \Leftarrow$ enforce_BC(H_t, HU_t, HV_t, dx) ▷ Enforce boundary conditions
  $(H, HU, HV) \Leftarrow$ FV_time_step$(Z_{dx}, Z_{dy}, H_t, HU_t, HV_t, dt, nx)$ ▷ Perform one time step of finite volume scheme
  $(H, HU, HV) \Leftarrow$ impose_tolerances(H, HU, HV)
  $T \Leftarrow T + dt$
Save solution to disk
Free memory space

2.2 Validation

Some data are also taken from the C++ code such as the final state of the water level and an history of the time step evolution during the simulation. Those two results were used to verify the correctness of the C++ code. Figure 1 and Figure 2 confirm that the sequential C++ code is working correctly with a negligible error in comparison to the real values.

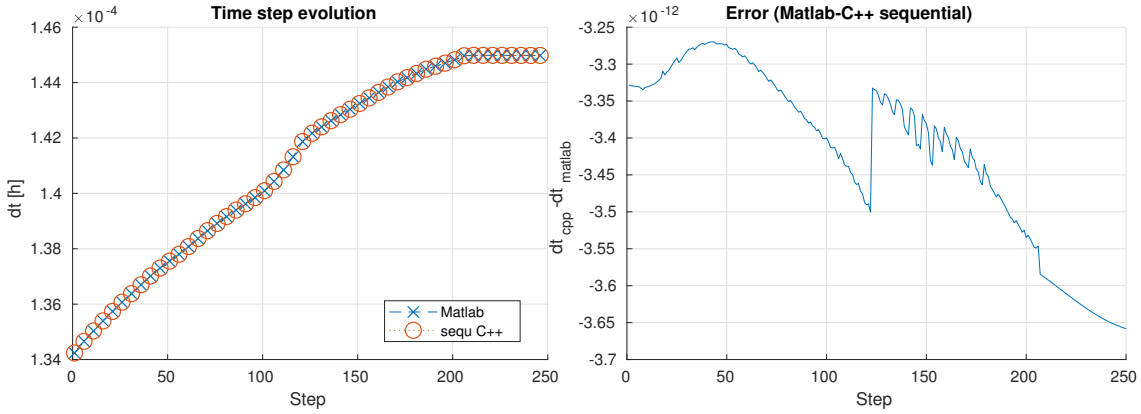

Figure 1: Comparison between the tracking of the time step in the Matlab and C++ sequential codes.

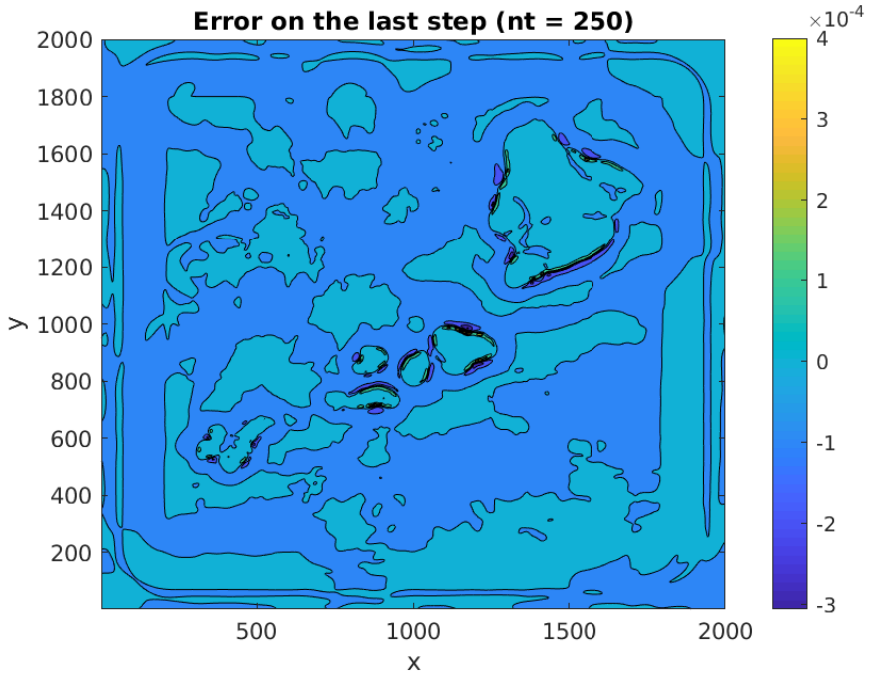

Figure 2: Error between the last step of the MATLAB code and sequential C++ one.

2.3 Performance comparison

Even if one could not expect an improvement in performance as great as the future parallelization of the algorithm, it can be interesting to figure out if the C++ language version performs already better the sequential algorithm than the MATLAB version. The performances in computational time has been measured through six independent launch of `compute.m` and `compute.cpp`. These measurement are reported on Figure 3 where one can observe a $\approx 250\%$

increase of the performance just by transferring the solver from MATLAB to C++. Of course the performance of the C++ code is related to the compilation process did with the most aggressive optimization option (`-Ofast`). The least optimized compilation (`-O0`) shows extremely reduced performances and take more than 300 seconds to run.

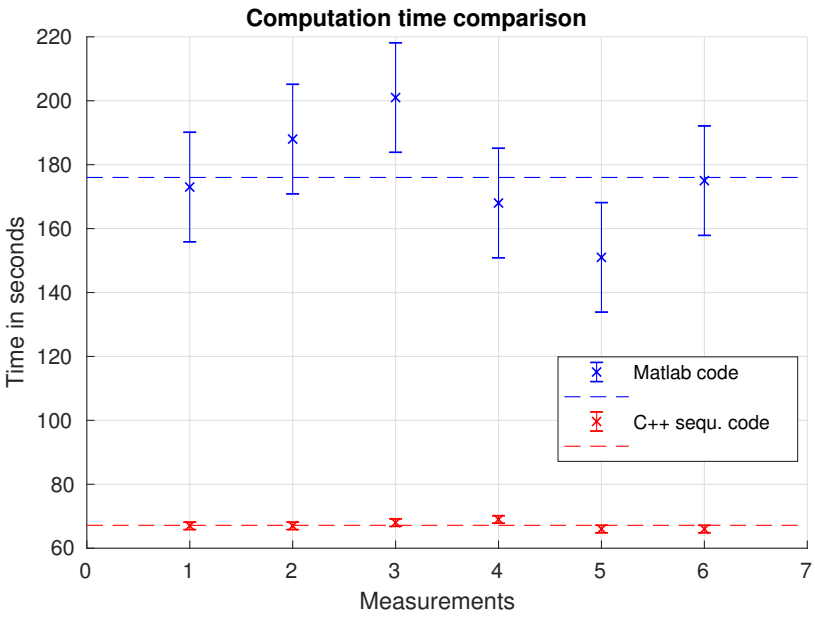

Figure 3: Performance comparison between MATLAB and sequential C++ code for 250 iterations and `g++ -Ofast` compilation option on an Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz.

2.4 Computing time analysis

Before starting the parallelization of our sequential code, it is important to perform a profiling of the computational cost of each steps in the Algorithm 1. The Table 1 presents the computational times (expressed in ticks) taken by the main process of our sequential code and the percentage it represents w.r.t. the global computational time. One can notice that the most costly part of the algorithm is to compute the steps of the finite volume explicit scheme. Keeping this in mind, a parallelization of the function `FV_time_step` must be the priority.

Step	Comp. Time [ms]	Share of total time
Variable Initialization	0.0432 ± 0.0447	$< 0.0001\%$
Initial state loading	57.021 ± 3.771	$0.0880 \pm 0.0073\%$
`new_dt`	3532.6 ± 79.5	$5.4466 \pm 0.0741\%$
Copy to temporary variables	2870.8 ± 51.017	$4.4271 \pm 0.0874\%$
`enfore_BC`	34.669 ± 1.9657	$0.0534 \pm 0.0017\%$
`FV_time_step`	57608 ± 1551.2	$88.8071 \pm 0.1638\%$
`impose_tolerances`	729.33 ± 21.124	$1.1245 \pm 0.0239\%$
Save output	25.334 ± 1.4030	$0.0390 \pm 0.0014\%$
Free memory space	9.1721 ± 0.3275	$0.0141 \pm 0.0002\%$

Table 1: Results for 10 manual profiling of the sequential C++ code for 250 step.

3 Parallelism Preview with CUDA

The GPU programming language CUDA has been chosen to implement a parallel version of the C++ sequential code.
The parallelization of the main `while` loop of Algorithm 1 can obviously not be parallelized since it needs the

results of the previous iteration at each steps. It is however not the case for the functions called inside the loop, i.e. `new_dt`, `enforce_BC`, `FV_time_step` and `impose_tolerances`.

3.1 Parallelization of `FV_time_step`

As demonstrated on Table 1, the `FV_time_step` function has the greatest potential of performance increase.
Since the treated variables are two dimensional arrays, the most natural way to parallelize this function is to divide the grid in to $N_b =$ `gridDim.x` regions and to declare a two dimensional grid composed of two dimensional blocks s.t.. Each region will be treated by one block containing $N_t = N/N_b$ threads. However in our case, the dimensions of the arrays are either 2001×2001, 4001×4001 or 8001×8001, dimensions that cannot be easily split into regular squared domains. For the instance, the parallel program will add an additional block which will execute the last processes which are not dividable by our .
It is important also to be aware that the process updates the variables H, HU and HV by performing a computation similar to a second degree derivative finite difference scheme from the temporary variables $H_t.HU_t, HV_t$. If the blocks have entirely access to those variables, the parallel computation of the finite volume step should not show difficulties. It can be however costly to load entirely all the variables at each time step but this has to be experimented later on.

3.2 Parallelization of `new_dt`

The `new_dt` function is aimed to compute the new time step respecting the CFL condition of the problem. This process is done by finding the maximum in an element-wise computation on the two dimensional arrays H, HU, HV. The minimal complexity of a parallel maximum algorithm is $O(n \log n)$ which is already a significant improve in comparison to the sequential one which is $O(n^2)$. However, since this process take only arround 5% of the computational time, the importance of such an optimization should be evaluated after parallelizing the finite volume step function.

Appendix

Since I am not familiar at all with c4sciences repository management, the code can be found on this Github repository.