CS 5220
Project 2 - Shallow Water Simulation

Marc Aurele Gilles (mtg79)
Sheroze Sheriffdeen(mss385)

October 19, 2015

1   Introduction

Define structured grid computations
Define shallow water simulations
Math overview and associated abstractions in code

The goal of this project to profile, parallelize and tune a particular structured grid computation: a shallow-water equations solver.

The shallow-water equations are a two-dimensional PDE system that waves that are very long compared to the water depth. The variables of the PDE system are the water height h, and the velocity components (u, v). The governing equations are

$$U_t = F(U)_x + G(U)_y$$

where

$$U = \begin{bmatrix} h \\ hu \\ hv \end{bmatrix}, F = \begin{bmatrix} hu \\ h^2u + gh^2/2 \\ huv \end{bmatrix}, G = \begin{bmatrix} hv \\ huv \\ h^2v + gh^2/2 \end{bmatrix}$$

2   Design Decisions

The following sections describe the implementation changes from the original code found at `https://github.com/cornell-cs5220-f15/water`.

2.1   Memory Layout

The original solution a two dimensional vectors of 3-vectors to represent U_t, $F(U)_x$, and $G(U)_y$. During each time step, the solution accesses each element in the 2-D grid sequentially. Then, the `vector<vector<real>>` representation leads to memory accesses that are not local spatially.

Therefore, our solution chooses to use 3 separate two dimensional vectors per objects, U_t, $F(U)_x$, and $G(U)_y$. Thus, we are required in general to perform three loop iterations in place of a single loop in the original solution. But this approach leverages spatial locality, especially in `compute_step` and `limited_derivs` functions.

2.2 Vectorization

By observing the profiling information, we noticed that the original solution spends majority of its computational time in the functions `limited_derivs`, `compute_step` and `compute_fg_speeds`. By adopting the newer memory layout, we enabled spatially local memory accesses. We were also able to decompose `for` loops in the solution to improve vectorization. Refer to `ipo_out_vectorization.optrpt` in `https://github.com/sheroze1123/water/tree/vectorization` for more information.

In `compute_fg_speeds`, we performed two separate loops to compute flux and wave speeds. The flux computation and the wave speed computation for the complete grid is not handled by the `Physics` class. We used `#pragma simd` directives to instruct the compiler the ability to vectorize these computations. The compiler was successfully able to vectorize these functions with an estimated potential speedup of 6.7.

`limited_derivs` uses the `limdiff` function in `minmod.h`. To improve the vectorization of this computation, we changed the implementation of `limdiff` in the following ways.

1. `limdiff` now performs the computation on the complete grid instead of at one grid point.

2. `limdiff` was decomposed as `limdiff_x` and `limdiff_y` to perform the limiter along the x dimension and the y dimension separately while still retaining unit stride.

2.3 Parallelization

The updated memory layout is amenable to parallelization via OpenMP, particular during the costly operations of `limited_derivs` and `compute_step`.

At the beginning of a call to `limited_derivs` we initialize a team of threads using `#pragma omp parallel`. Each application of the limiter to the components of U_t, $F(U)_x$, and $G(U)_y$ can be computed in parallel since we removed the data dependencies by creating 3 separate two dimensional arrays per component. We use the `#pragma omp for` directive to parallelize the grid computation of each component.

In `compute_step`, we initialize a team of threads using `#pragma omp parallel`. The predictor step is now performed in parallel. Then, we create a barrier before proceeding to the corrector step since the the corrector step needs information from the predictor. After computing the corrector step, we perform a barrier before finally copying back to storage in parallel. The performance increase we observed via parallelization reached up to 3+ speedup. Results are plotted in section 3.2.

2.4 Domain Decomposition

2.4.1 General Setup

We decomposed the main grid into equally sized sub-grids, by first copying each sub-grid into a separate array. Each sub-grid array contains its part of the main grid an outer layer of ghost cells which contain information about the neighboring cells or the cells on the opposite side of the main grid if the sub-domain is close to the boundary of the main grid as we use periodic boundary condition. We then perform some number of time steps on each independent sub-grid and finally synchronize by copying back the sub-grid (without the ghost cells) onto the main grid.

2.4.2 Ghost cells

The width of the layer of ghost cells for each sub-grid is 3t, where t is the number of time steps we wish to advance before synchronizing back onto the main grid. For each sub-grid, we declare a simulation object, and run the simulation essentially in the same manner as we would on the whole grid, except we never apply periodic boundary conditions on the sub-grids. Because of this, at each time step we perform without synchronization, we "loose" 3 layers of ghost cells, in the sense that the computation within this layer is erroneous. However, the rest of the grid is not contaminated with error, as information spreads through our simulation only by 3 layers per time step.

2.4.3 Estimating the wave speed

One of the main barrier to using a domain-decomposition parallelization for this simulation is that we need to estimate the wave speed at each time step to be able to estimate a how big of a time step we can perform for the next iteration. Hence to allow each sub-grid to run from time t=0 to time t=tfinal, we would need know in advance the speeds at time t, t+dt1, t+(dt1+dt2)... $t + \sum_{i=1}^{k-1} dti$ which we obviously don't have access to at time t. The problem is that we need to know at least an upper bound on the wave speed at each time step to be able to decide how many time steps it will take to reach tfinal, as the number of time steps needed is directly proportional to the number of ghost cells we need to allocate for each sub-grid at time t. Our solution is to estimate an upper bound on the max wave speed, we set the max wave speed for the next k iterations to a be a multiple r of the speed wave at time t (which we can compute before parallelizing)(win our code, we set r=2). Furthermore, for each sub grid at each time checks independently if wave speed on the subdomain is greater than our bound, if it is the case our program throws an error and stops (but no synchronization to do this check). A better alternative, which we have not yet implemented would be to stop the computation if the wave speed is greater than our bound at time, double k and start the computation all over again from time t.

3 Analysis

3.1 Profiling

The following time profiles were obtained on a 200x200 grid by advancing 100 frames.

3.1.1 Original solution

We began the optimization by analyzing the time profiles of the original code. The following table shows the top 4 functions by CPU time.

Function	Module	CPU Time	Spin Time	Overhead Time
Central2D<Shallow2D , MinMod<float >>::limited_derivs	shallow	2.529s	0s	0s
Central2D<Shallow2D , MinMod<float >>::compute_step	shallow	1.210s	0s	0s
Central2D<Shallow2D , MinMod<float >>::compute_fg_speeds	shallow	0.426s	0s	0s
_IO_file_xsputn	libc−2.12.so	0.027s	0s	0s
_IO_fwrite	libc−2.12.so	0.025s	0s	0s

3.1.2 Memory Layout Update

3.1.3 Vectorization

Profiling of vectorization shows good improvements in performance, especially in the `limited_derivs` and `compute_fg_speeds` functions, but a reduction in performance in the `compute_step` func-

tion. The repetition of `limdiff_x` and `limdiff_y` is due to separation of components and axes in `limited_derivs`.

Function	Module	CPU Time	Spin Time	Overhead Time
compute_step	shallow	2.136 s	0 s	0 s
limited_derivs	shallow	0.448 s	0 s	0 s
compute_fg_speeds	shallow	0.362 s	0 s	0 s
limdiff_y	shallow	0.204 s	0 s	0 s
limdiff_x	shallow	0.201 s	0 s	0 s
limdiff_x	shallow	0.200 s	0 s	0 s
limdiff_y	shallow	0.200 s	0 s	0 s
limdiff_x	shallow	0.197 s	0 s	0 s
limdiff_x	shallow	0.192 s	0 s	0 s
limdiff_y	shallow	0.192 s	0 s	0 s
limdiff_x	shallow	0.188 s	0 s	0 s
limdiff_y	shallow	0.186 s	0 s	0 s
limdiff_y	shallow	0.184 s	0 s	0 s
vector<float, allocator<float >>::operator []	shallow	0.062 s	0 s	0 s
vector<float, allocator<float >>::operator []	shallow	0.061 s	0 s	0 s
vector<float, allocator<float >>::operator []	shallow	0.051 s	0 s	0 s

3.1.4   Parallelization

3.1.5   Domain Decomposition

Our domain decomposition code suffers from instability. If the subgrid is set to be the whole grid plus the layers of ghost cells, the code is stable. Also, if the time steps taken are very small: about 10 times smaller than they ought to be, the code is also stable. In all other situation we observe some small scale instability which is certainly due to a bug in our implementation. We hope to have this fixed by the final report submission deadline

3.2   Scaling Study

3.2.1   Strong Scaling Study

Using a 500x500 grid and 100 frames, we observe the speedup with respect to the number of threads in our parallel implementation.
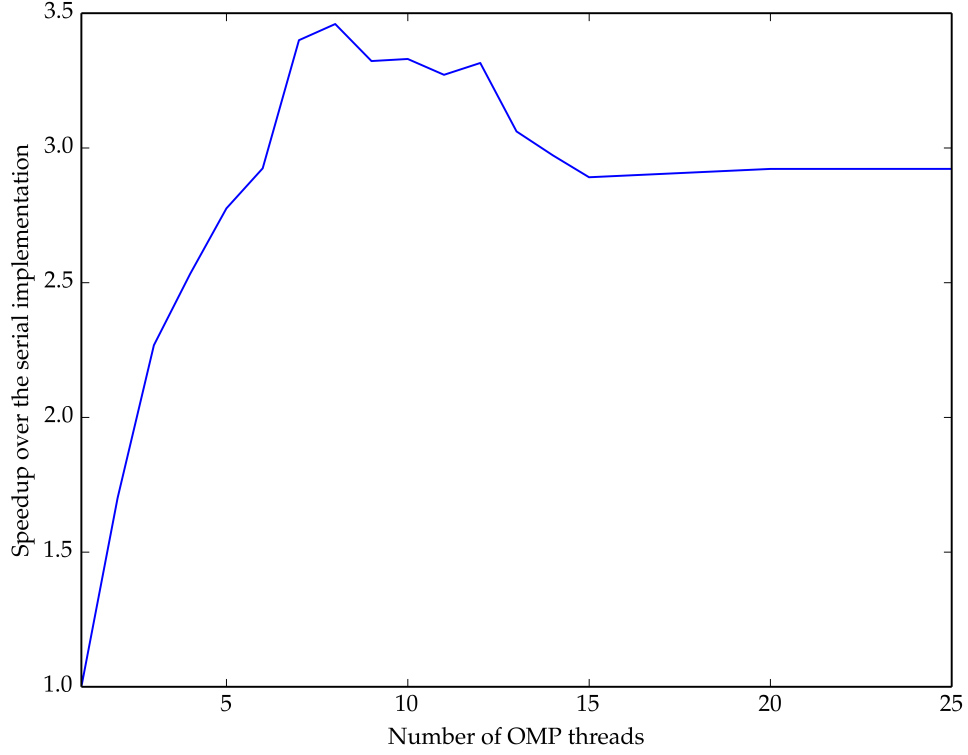
Figure 1: Speedup as a function of the number of threads

3.2.2 Weak Scaling Study

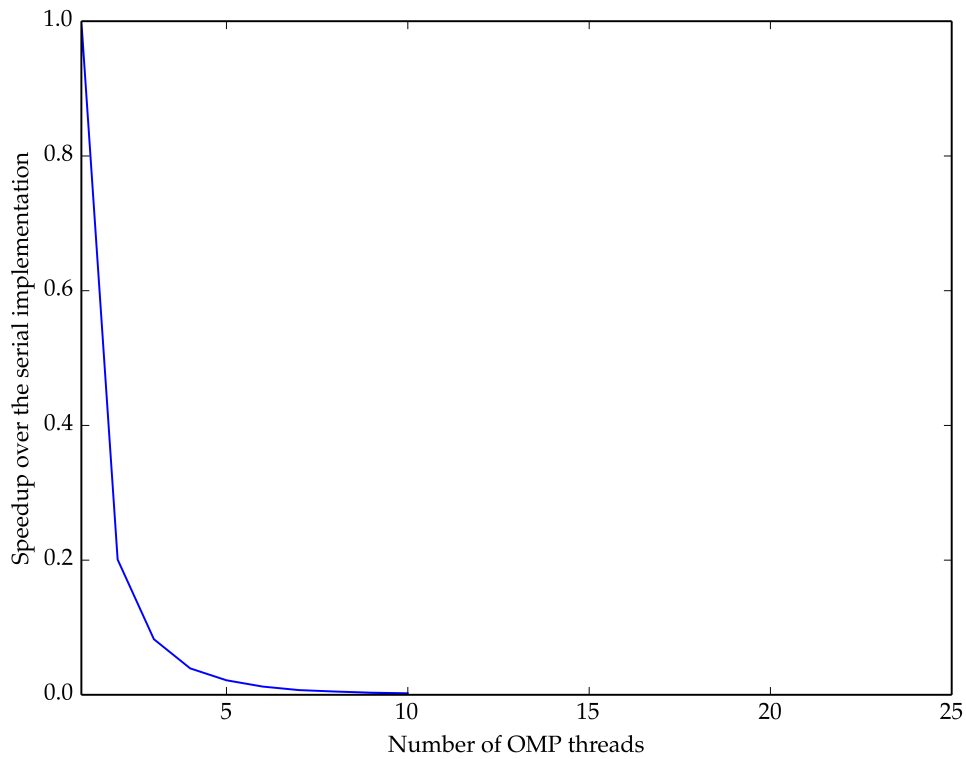We vary the threads but keep the problem size/thread constant.

Figure 2: Speedup as a function of the number of threads

References

[1] Data Alignment to Assist Vectorization. (n.d.). Retrieved September 30, 2015, from `https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization`