CS 5220
Project 2 - Shallow Water Simulation

Marc Aurele Gilles (mtg79)
Sheroze Sheriffdeen(mss385)

October 29, 2015

1 Introduction

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 describes 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`. Our solution can be found at `https://github.com/sheroze1123/water` in the `master` branch.

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 $1.5t$, 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 two time steps 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 2 time steps.

2.4.3 Estimating the wave speed

The main barrier to in adopting 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 $t = t_{final}$, we would need know in advance the speeds at time t, $t + dt_1$, $t + (dt_1 + dt_2)$ ... $t + \sum_{i=1}^{k-1} dt_i$. But we obviously do not have access to those speeds 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 t_{final}, 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, set the max wave speed for the next k iterations to a be a multiple r of the speed of the wave at time t which we can compute before parallelizing. (in our code, we set $r = 1.5$).

Furthermore, for each sub grid at each time checks independently if the wave speed on the subdomain is greater than our bound, if it is the case our program throws an error and stops. This failure check occurs locally to each thread, so there is no need for synchronization.

2.4.4 Avoiding Race conditions

In order to avoid the race condition that can occur by a thread mapping back the state of the simulation on a later time steps onto the main grid before another thread got to copy from the main grid the simulation at the current time step, we use two different main grid (main grid 1 and main grid 2). At the first iteration (which computes multiple time steps), all thread copy from main grid 1, do computations, and write onto main grid 2. When the first iteration is done, all thread copy from main grid 2, do computations, and write onto main grid 1. We alternate between the grids in this manner until we have reached tfinal, making sure that the final state is written onto main grid 1 (which is passed to the driver function). In this manner, we avoid race conditions between the threads, and keep the synchronization costs to a minimum.

2.4.5 Choosing a number of time steps

The size of the computation that each subgrid has to do for each time steps depends on the number of time steps it has to accomplish before synchronizing back to the main grid. Specifically, the subgrid with ghost cells have sizes $(n + 2(t\frac{3}{2} + 3))^2$ where n is the size of the part of the main grid that is assigned to that the subgrid, and t is the number of time steps taken before synchronization.

The 3/2 factor is due to the fact that the information spreads at the rate of 3 layers per 2 time steps in our computation, the 2 factor to account for the the fact that we need one layers on each side of the grid, and the constant 3 is implementation dependent. We use the following heuristic to choose the number of time steps that we perform: the work per time steps on the subgrid with ghost cells has to be at most a multiple c (in our code, we use $c = 1.5$) of the work that would be performed on the subgrid alone (with no ghost cells). This leads us to the inequation:

$$(n + 2(t\frac{3}{2} + 3))^2 \leq cn^2$$

A few algebra steps lead us to:

$$t \leq \frac{(\sqrt{c} - 1)n}{3} - 2$$

We choose the largest even number t that satisfies this bound, or 2, whichever is largest.

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. We note that the bottleneck in the original solution lies in `limited_derivs` and `compute_step`.

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 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` function. 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	2.136s	0s	0s
limited_derivs	shallow	0.448s	0s	0s
compute_fg_speeds	shallow	0.362s	0s	0s
limdiff_y	shallow	0.204s	0s	0s
limdiff_x	shallow	0.201s	0s	0s
limdiff_x	shallow	0.200s	0s	0s
limdiff_y	shallow	0.200s	0s	0s
limdiff_x	shallow	0.197s	0s	0s
limdiff_x	shallow	0.192s	0s	0s
limdiff_y	shallow	0.192s	0s	0s
limdiff_x	shallow	0.188s	0s	0s
limdiff_y	shallow	0.186s	0s	0s
limdiff_y	shallow	0.184s	0s	0s
vector<float , allocator<float >>::operator []	shallow	0.062s	0s	0s
vector<float , allocator<float >>::operator []	shallow	0.061s	0s	0s
vector<float , allocator<float >>::operator []	shallow	0.051s	0s	0s

3.1.3 Parallelization

For parallelization, we list the functions that take up the most CPU time and their associated OpenMP spin times. Although we see speedups in the parallel code, the profiling shows that we spend a lot of CPU time spinning. This indicates load imbalance issues in our code that we aim to address in the final report.

```
Function                          Module          CPU Time   Spin Time(OpenMP)
--------------------------------  --------------  --------   -----------------
__kmp_wait_template<kmp_flag_64>  libiomp5.so     47.056s              47.056s
compute_step                      shallow          3.919s                   0s
__kmp_wait_template<kmp_flag_64>  libiomp5.so      3.001s               3.001s
[Outside any known module]        [Unknown]        2.635s                   0s
__kmp_yield                       libiomp5.so      1.114s               1.114s
notdone_check                     libiomp5.so      1.097s               1.097s
__kmp_x86_pause                   libiomp5.so      0.973s               0.973s
__kmp_x86_pause                   libiomp5.so      0.887s               0.887s
compute_fg_speeds                 shallow          0.648s                   0s
limdiff_x                         shallow          0.424s                   0s
limdiff_x                         shallow          0.419s                   0s
limdiff_x                         shallow          0.418s                   0s
limdiff_x                         shallow          0.417s                   0s
limdiff_x                         shallow          0.417s                   0s
limdiff_y                         shallow          0.413s                   0s
limdiff_y                         shallow          0.406s                   0s
limdiff_x                         shallow          0.402s                   0s
limdiff_y                         shallow          0.386s                   0s
limdiff_y                         shallow          0.380s                   0s
limdiff_y                         shallow          0.374s                   0s
limdiff_y                         shallow          0.357s                   0s
```

3.2 Scaling Study

3.2.1 Strong Scaling Study

We observe the speedup with respect to the number of threads in our parallel implementation. Here, speedup s is defined to be,

$$\mathrm{s} = \frac{t_{parallel}}{t_{serial}} \tag{1}$$

3.2.1.1 Parallel approach with vectorization

Following is the strong scaling study where we solve the problem using a single domain with parallelization and vectorization. We solve the problem on a 500x500 grid for 100 frames. The observed best performance occurs when the number of threads are ≈ 10. Increasing the number of threads further results in plateauing in speedup which can be attributed to extra threads not being employed due to limited parallelization. The plateau is lower than the observed maximum speedup. We believe that this is due to thread scheduling overhead. Note that the original solution has a speedup < 1 in the following plot. The implementation of this version can be found at the `vector_parallel` branch.

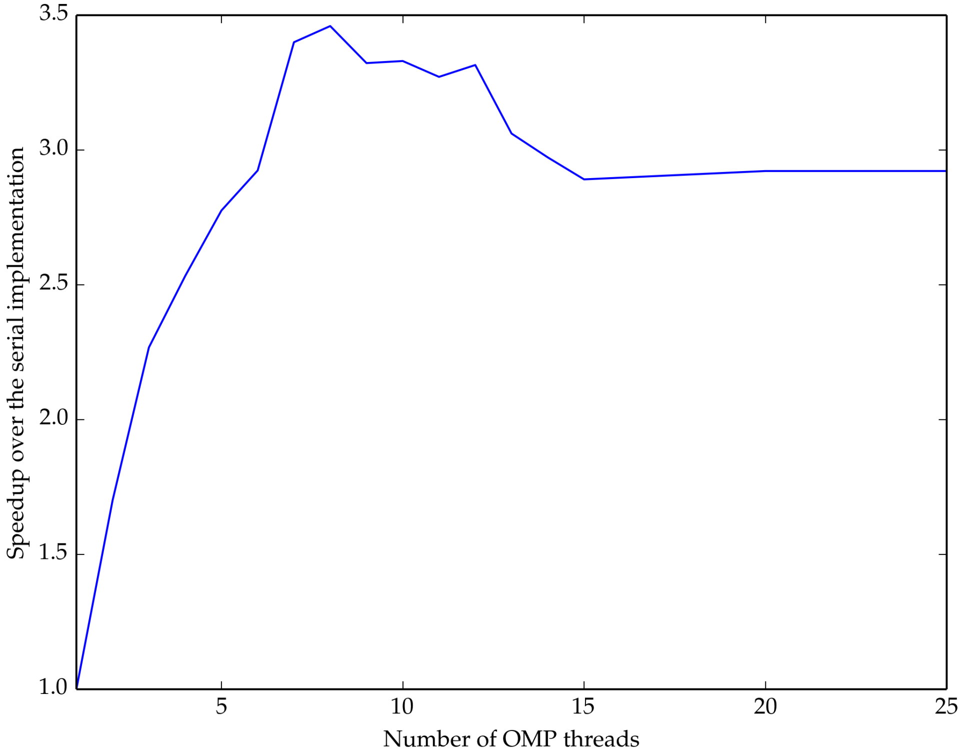

Figure 1: Speedup as a function of the number of threads

3.2.1.2 Parallel domain decomposition

The strong scaling study for the domain decomposed version of the solution shows good speedup with increasing number of threads. The problem was solved on a 1000x1000 grid for 100 frames. The implementation can be found at the `spatial_vector_merge` branch.

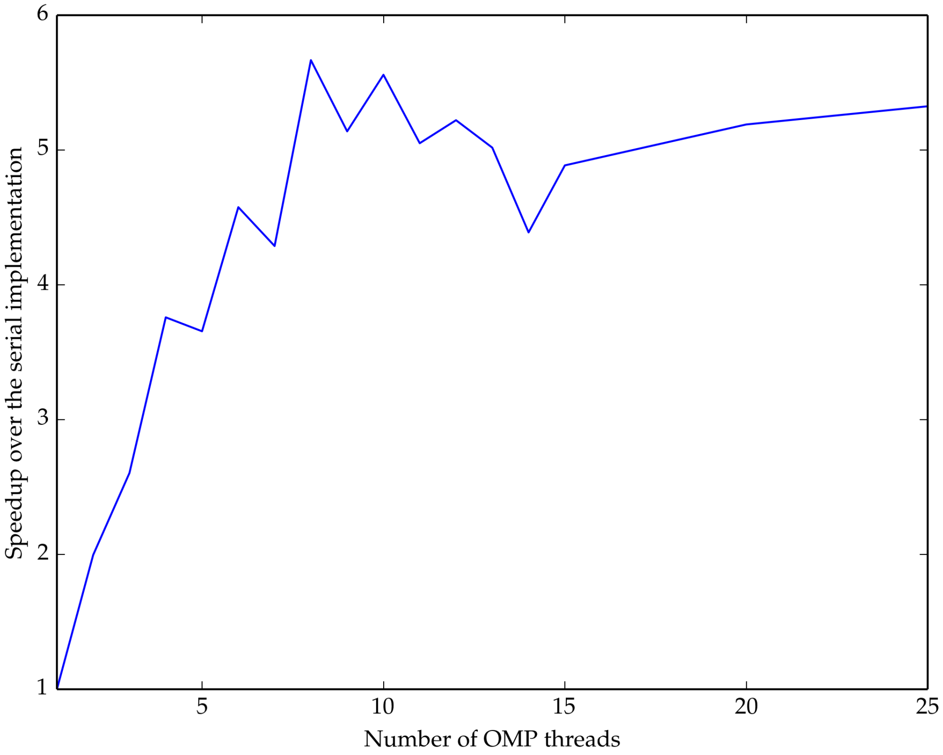

Figure 2: Speedup as a function of the number of threads

In the current implementation, each thread get a n/4 by n/4 part of the main grid, which might explain the speed up being relatively constant after 16. Although note that the extra thread are used anyway in the parallel computation of each subgrid.

3.2.2 Weak Scaling Study

We vary the threads but keep the problem size per thread constant. In figure 3, each thread ideally computes a sub-problem equivalent to advancing a 300 x 300 grid by 100 frames. We plot the speedup observed while increasing the number of threads and the problem size such that work done per thread is approximately equal.

3.2.2.1 Parallel approach with vectorization

In the single domain, parallelized approach, we observe a exponential decrease in speedup per core. We believe that this can be attributed to load imbalance between threads and thread scheduling overhead. The implementation of this version can be found at the `vector_parallel` branch.

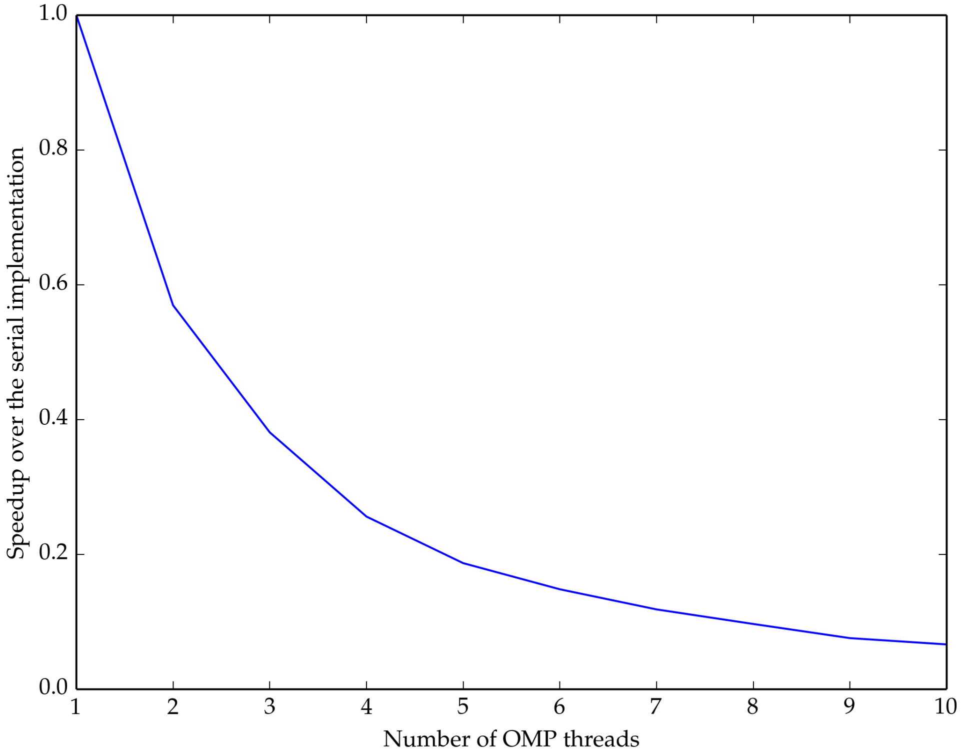

Figure 3: Speedup as a function of the number of threads

3.2.2.2 Parallel domain decomposition

The implementation can be found at the `spatial_vector_merge` branch. In the domain decomposition case, we observe a strong decrease in speedup as the number of OMP threads grow. On top of load imbalance between threads and thread scheduling, this might be related to the increasing synchronization and computation cost that occur when n increases. Indeed, when n increases the number of time steps before synchronization increases, therefore each subgrid gets more ghost cells and therefore more work and more communication (cf section 2.4.5).

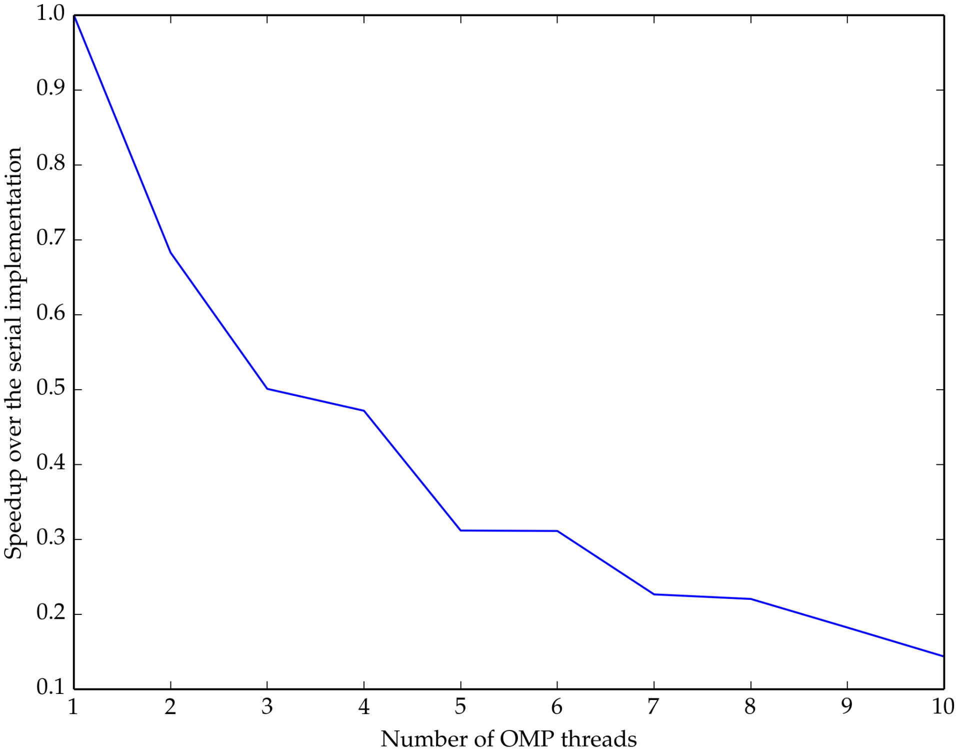

Figure 4: Speedup as a function of the number of threads

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`

[2] Yliluoma, J. (n.d.). Guide into OpenMP: Easy multithreading programming for C. Retrieved October 19, 2015, from `http://bisqwit.iki.fi/story/howto/openmp/`