CS 5220 Project 2
Shallow Water Simulation

Batu Inal [bi49], Bob(Kunhe) Chen [kc853]
Cornell University

Partners: Enrique Rojas [elr96], Wenjia Gu [wg233]

1 Theory

1.1 Shallow water simulation

For a simplified shallow water physics model, we look at water height in the z-direction and its momentum in the xy-plane, which describes an incompressible and momentum-conserving water flow. In mathematical terms, the system dynamics can be written as a set of partial differential equations:

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

In a more compact form:

$$U_t + F(U)_x + G(U)_u = 0$$

Notice that this is a first order linear PDE, we can discretize the space onto a finite grid and use Lax-Wendroff scheme. Lax-Wendroff scheme is of order (Δx^2, Δt^2) as long as the Courant Friedrichs Lewy (CFL) condition is satisfied. Implicitly, we can think Lax-Wendroff as a two-step method in which it updates the current state with half step to compute a intermediate state and then use those states in the next step to compute finite difference to approximate derivative for the actual computation. We enforce a periodic boundary condition on the system to complete our physics model for simulation. We refer interested reader to *Explicity Shallow Water Simulations on GPUs: Guidelines and Best Practices* for more details on the physics model.

1.2 J-T algorithm

In reality, Lax-Wendroff scheme works well with finite derivatives but it breaks when shock appears. For the shallow water physics model, it is inevitable that a shock appears because water with larger height will travel faster,eventually forming a discontinuity when it catches up with water of smaller height. To address this problem, we introduce a limiter function to approximate the shock with a large derivative in our finite difference equations. Instead of original Lax-Wendroff scheme, we adopt a finite difference scheme for solving hyperbolic PDE systems proposed by Jiang and Tadmor.

The Jiang-Tadmor scheme works by alternating between a main grid and a staggered grid offset by half a step in each direction. In the implementation of this scheme, we always take even number of steps to ensure that we only have to concern ourselves with the values on the main grid. In this update scheme, the value on each cell in the next step depends on the three cells that are adjacent to it. This value dependency is important in our application of domain decomposition using ghost cells.

1.3 Ghost Cell Expansion

We expand our domain to address the problem of information dependency in our finite difference scheme with periodic boundary condition. The idea is to expand the domain size by `ng` and populate

those spaces using a periodic condition. In this way, we do not have to have cells communicate to each other across the whole domain to implement this periodic boundary condition. And if we increase the number of ghost cells, we do more redundant computation but will not have to update the periodic condition for every step.

Moreover, the concept of ghost cell expansion is central to the implementation of domain decomposition. Once we divide the space into subdomains, we want the inter-subdomain communication to be as infrequent as possible for efficient parallel computing. Just as we use ghost cells to avoid communication across the domain, we can do the same to avoid communication between subdomains.

1.4 Domain Decomposition

The idea of domain decomposition is simple: it divides the space into subspaces and distribute the data to different threads for parallel computing. As discussed above, we can expand the subdomain and pass more information to each thread to reduce communication between threads and make the parallel work more efficient.

An ideal case will be for us to divide the domain evenly and distribute them for different threads. Once all the threads finish their work, we sync the information and repopulate the ghost cells for the next iteration. Finally, when the computation is done, we reconstruct the main grid using data from each subdomain. On each thread, we do more overhead work to get ghost cells and do extra computation on them to reduce the frequency for communication.

2 What Is Done For Serial Code

We updated our work from C++ code to C code because of the changed data structure greatly increased the running speed by enabling compiler to vectorize the loops in the code.

2.1 Data Structure

Initially, the C++ code has an array of structure for computation. More specifically, the grid values are stored in an array of data in the $\mathbb{R}^3$ vector U. But this data structure will break the spatial locality when we try to update the values in a loop. A more computationally efficient way to arrange the data is to have structured array where the values U_1, U_2 and U_3 are stored in different sections of a long array. As a result, the compiler is able to vectorize the inner loops much more.

3 What is Done For Parallel Code

3.1 Domain Decomposition

In the core function `central2d_step()`, we start by making copies of the domain. We define a function `copy_subdomain()` that essentially assigns a smaller data array for the storage of U. The size is determined by the number of threads used for OpenMP.

3.2 Ghost Cell Update

In the original code, we change the number of ghost cells `ng` and the update range for `central2d_step()` function, so that we have more ghost cells and we update them all every time. As a result, we do

not have to apply the periodic update between odd and even steps. In fact, we do three batches of odd&even step updates before synchronizing the value of U to increase the local computational cost.

We synchronize before and after each batch step. We call a function `update_subdomain()` to get update the values in ghost cells before the batch starts and after it we have a barrier to pause the computation and synchronize by calling `sync_subdomain()`.

3.3 OpenMP

The OpemMP program is started for each frame by calling `#pragma omp parallel`. We call `#pragma omp for` to parallelize the batch step update. We assign each subdomain to a thread using its identifier and synchronize at the end before a `#pragma omp barrier`

4 Implementation and Testing

4.1 Bottleneck Study

To develop the profiling reports, we use the amplxe-cl tool. From the hotspots report option, we found that the functions `limited_derivs`, `compute_step` and `compute_fg_speeds` were consuming the most cpu time, by at least one order of magnitude as we can see in Figure 1.

Function	Module	CPU Time
Central2D<Shallow2D, MinMod<float>>::limited_derivs	shallow	1.823s
Central2D<Shallow2D, MinMod<float>>::compute_step	shallow	0.976s
Central2D<Shallow2D, MinMod<float>>::compute_fg_speeds	shallow	0.621s
_IO_fwrite	libc-2.12.so	0.060s
main	shallow	0.050s

Figure 1: *Output from Amplxe-cl demonstarting the time spent in major function*

As we can observe from above around 60 percent of the time is spent in the function `limited_derivs()`. Which we believe is accounted to the following bottleneck code in limdiffs, which is called by `limited_derivs()`, where loops go over every single cell:

```
// Apply limiter to all components in a vector
    static void limdiff(vec& du, const vec& um, const vec& u0, const vec& up) {
        for (int m = 0; m < du.size(); ++m)
            du[m] = Limiter::limdiff(um[m], u0[m], up[m]);
    }
```

In further analyzing the code through amplxe-cl tool we obtain the following report which proves our previous hypothesis

```
Source Line  Source                                                                    CPU Time  Spin Time  Overhead Time
-----------  ------------------------------------------------------------------------  --------  ---------  -------------
155              std::vector<vec> gy_;           // y differences of g
156              std::vector<vec> v_;            // Solution values at next step
157
158              // Array accessor functions
159
160              int offset(int ix, int iy) const { return iy*nx_all+ix; }
161
162              vec& u(int ix, int iy)    { return u_[offset(ix,iy)]; }
163              vec& v(int ix, int iy)    { return v_[offset(ix,iy)]; }
164              vec& f(int ix, int iy)    { return f_[offset(ix,iy)]; }
165              vec& g(int ix, int iy)    { return g_[offset(ix,iy)]; }
166
167              vec& ux(int ix, int iy)   { return ux_[offset(ix,iy)]; }
168              vec& uy(int ix, int iy)   { return uy_[offset(ix,iy)]; }
169              vec& fx(int ix, int iy)   { return fx_[offset(ix,iy)]; }
170              vec& gy(int ix, int iy)   { return gy_[offset(ix,iy)]; }
171
172              // Wrapped accessor (periodic BC)
173              int ioffset(int ix, int iy) {
174                  return offset( (ix+nx-nghost) % nx + nghost,
175                                 (iy+ny-nghost) % ny + nghost );
176              }
177
178              vec& uwrap(int ix, int iy)  { return u_[ioffset(ix,iy)]; }
179
180              // Apply limiter to all components in a vector
181              static void limdiff(vec& du, const vec& um, const vec& u0, const vec& up) {
182                  for (int m = 0; m < du.size(); ++m)
183                      du[m] = Limiter::limdiff(um[m], u0[m], up[m]);                    0.202s         0s             0s
184              }
185
186              // Stages of the main algorithm
187              void apply_periodic();
188              void compute_fg_speeds(real& cx, real& cy);
189              void limited_derivs();
190              void compute_step(int io, real dt);
191
192          };



287          template <class Physics, class Limiter>
288          void Central2D<Physics, Limiter>::limited_derivs()
289          {
290              for (int iy = 1; iy < ny_all-1; ++iy)
291                  for (int ix = 1; ix < nx_all-1; ++ix) {                           0.005s         0s             0s
292
293                      // x derivs
294                      limdiff( ux(ix,iy), u(ix-1,iy), u(ix,iy), u(ix+1,iy) );
295                      limdiff( fx(ix,iy), f(ix-1,iy), f(ix,iy), f(ix+1,iy) );
296
297                      // y derivs
298                      limdiff( uy(ix,iy), u(ix,iy-1), u(ix,iy), u(ix,iy+1) );
299                      limdiff( gy(ix,iy), g(ix,iy-1), g(ix,iy), g(ix,iy+1) );
300                  }
301          }
302
```

Figure 2: *Inline report for function limited derivs*

After changing some flags from the the Make.in.icc file, namely: -O3, -fast, -opt-prefetch, -unroll mainly for loop unrolling and -aggressive , -parallel, -ansi-alias, -ftree-vectorize, -xHost, -axCORE-AVX2 to increase the vectorization capabilities of each single processor, we realize that the assembly code of the functions above had already been vectorized so we could not further optimize by the help of vectorization for the specific function in hand. We believe that this is accounted due to the data not being stored in contiguous memory so every time there is the overhead of fetching data from memory perhaps thrashing the cache. As suggested by Prof.Bindel the main idea here would be to work on improving data dependency.

As we had recognized earlier, `compute_step()` is also one of our longest CPU-time taking functions. We believe that this is due to the fact, as it can be witnessed from the analysis below, that majority of the time is spent on the correction loops, then prediction step loop and lastly the copying loop. The code and the analysis for `compute_step()` is presented below:

```
332             // Predictor (flux values of f and g at half step)
333             for (int iy = 1; iy < ny_all-1; ++iy)                                   0.001s        0s            0s
334                 for (int ix = 1; ix < nx_all-1; ++ix) {                             0.001s        0s            0s
335                     vec uh = u(ix,iy);                                              0.008s        0s            0s
336                     for (int m = 0; m < uh.size(); ++m) {
337                         uh[m] -= dtcdx2 * fx(ix,iy)[m];
338                         uh[m] -= dtcdy2 * gy(ix,iy)[m];                             0.030s        0s            0s
339                     }
340                     Physics::flux(f(ix,iy), g(ix,iy), uh);
341                 }
342
343             // Corrector (finish the step)
344             for (int iy = nghost-io; iy < ny+nghost-io; ++iy)
345                 for (int ix = nghost-io; ix < nx+nghost-io; ++ix) {                 0.008s        0s            0s
346                     for (int m = 0; m < v(ix,iy).size(); ++m) {
347                         v(ix,iy)[m] =                                               0.105s        0s            0s
348                             0.2500 * ( u(ix,  iy)[m] + u(ix+1,iy  )[m] +            0.003s        0s            0s
349                                        u(ix,iy+1)[m] + u(ix+1,iy+1)[m] ) -          0.034s        0s            0s
350                             0.0625 * ( ux(ix+1,iy  )[m] - ux(ix,iy  )[m] +
351                                        ux(ix+1,iy+1)[m] - ux(ix,iy+1)[m] +
352                                        uy(ix,  iy+1)[m] - uy(ix,  iy)[m] +
353                                        uy(ix+1,iy+1)[m] - uy(ix+1,iy)[m] ) -        0.009s        0s            0s
354                             dtcdx2 * ( f(ix+1,iy  )[m] - f(ix,iy  )[m] +            0.001s        0s            0s
355                                        f(ix+1,iy+1)[m] - f(ix,iy+1)[m] ) -          0.037s        0s            0s
356                             dtcdy2 * ( g(ix,  iy+1)[m] - g(ix,  iy)[m] +            0.005s        0s            0s
357                                        g(ix+1,iy+1)[m] - g(ix+1,iy)[m] );           0.029s        0s            0s
358                     }
359                 }
360
361             // Copy from v storage back to main grid
362             for (int j = nghost; j < ny+nghost; ++j){
363                 for (int i = nghost; i < nx+nghost; ++i){                           0.005s        0s            0s
364                     u(i,j) = v(i-io,j-io);                                          0.022s        0s            0s
365                 }
366             }
367         }
368
```

Figure 3: *Inline report for function limited_derivs*

The last function we analyzed while profiling our code was compute fg speeds. We realized that the most time spent in the code without any ghost cell decomposition was the std::max function call, figured out from generated compiler optimization output, due to the fact that the compiler was not able to optimize due to a function call.

```
Source Line  Source                                                                  CPU Time  Spin Time  Overhead Time
-----------  ----------------------------------------------------------------------  --------  ---------  -------------
258           * bound on the CFL number).
259           */
260
261          template <class Physics, class Limiter>
262          void Central2D<Physics, Limiter>::compute_fg_speeds(real& cx_, real& cy_)
263          {
264              using namespace std;
265              real cx = 1.0e-15;
266              real cy = 1.0e-15;
267              for (int iy = 0; iy < ny_all; ++iy)
268                  for (int ix = 0; ix < nx_all; ++ix) {                           0.013s        0s            0s
269                      real cell_cx, cell_cy;
270                      Physics::flux(f(ix,iy), g(ix,iy), u(ix,iy));
271                      Physics::wave_speed(cell_cx, cell_cy, u(ix,iy));
272                      cx = max(cx, cell_cx);                                      0.020s        0s            0s
273                      cy = max(cy, cell_cy);                                      0.008s        0s            0s
274                  }
275              cx_ = cx;
276              cy_ = cy;
277          }
278
279          /**
280           * ### Derivatives with limiters
281           *
282           * In order to advance the time step, we also need to estimate
```

Figure 4: *Inline report for function compute fg speeds*

4.2 Initial Attempts

Our initial attempt to tune the code was to naively identify loops where `#pragma omp parallel` for would not break any correctness and yield better performance results; however in this implementation for only a grid of size 200 we yielded performance degradation of around 700 percent. The results from the serial version and naive parallel version can be observed below.

```
amplxe: Executing actions 50 % Generating a report                                  Function
-------------------------------------------------------------  -------------  --------  ---------  --------------
Central2D<Shallow2D, MinMod<float>>::limited_derivs            shallow        0.007s          0s              0s
[Outside any known module]                                     [Unknown]      0.002s          0s              0s
Central2D<Shallow2D, MinMod<float>>::compute_fg_speeds         shallow        0.001s          0s              0s
Central2D<Shallow2D, MinMod<float>>::compute_step              shallow        0.001s          0s              0s
Central2D<Shallow2D, MinMod<float>>::run                       shallow        0.001s          0s              0s
_IO_file_xsputn                                                libc-2.12.so   0.001s          0s              0s
do_lookup_x                                                    ld-2.12.so     0.001s          0s              0s
amplxe: Executing actions 100 % done
```

Figure 5: *Serial Performance Analysis on Functions for Grid size 200*

```
P)  Spin Time:Lock Contention (OpenMP)  Spin Time:Communication (MPI)  Spin Time:Other  Overhea
-------  -------------------------------  -------------------------------  ---------  ---
__kmp_wait_template<kmp_flag_64>                                libiomp5.so                3.571s
     0s                                0s                                0s
__kmp_wait_template<kmp_flag_64>                                libiomp5.so                0.470s
     0s                                0s                                0s
[Outside any known module]                                      [Unknown]                  0.106s
     0s                                0s                                0s
__kmp_yield                                                     libiomp5.so                0.089s
     0s                                0s                                0s
kmp_basic_flag<unsigned long long>::notdone_check               libiomp5.so                0.077s
     0s                                0s                                0s
__kmp_x86_pause                                                 libiomp5.so                0.072s
     0s                                0s                                0s
__kmp_x86_pause                                                 libiomp5.so                0.067s
     0s                                0s                                0s
__kmp_hyper_barrier_release                                     libiomp5.so                0.016s
     0s                                0s                                0s
kmp_basic_flag<unsigned long long>::notdone_check               libiomp5.so                0.016s
     0s                                0s                                0s
__kmp_x86_pause                                                 libiomp5.so                0.015s
     0s                                0s                                0s
__kmp_x86_pause                                                 libiomp5.so                0.013s
     0s                                0s                                0s
Central2D<Shallow2D, MinMod<float>>::compute_step               shallow                    0.006s
     0s                                0s                                0s
__kmp_x86_pause                                                 libiomp5.so                0.006s
     0s                                0s                                0s
Central2D<Shallow2D, MinMod<float>>::limited_derivs             shallow                    0.005s
     0s                                0s                                0s
__kmp_store_x87_fpu_control_word                                libiomp5.so                0.003s
     0s                                0s                                0s
[OpenMP dispatcher]                                             libiomp5.so                0.002s
```

Figure 6: *Naive Parallel Performance Analysis on Functions for Grid size 200*

We were not surprised to see the harsh degradation in performance as the problem in hand was not parallelizable through the given code-base. Therefore the time spent for the threads to communicate/wait for each other exceeded the amount of work they were actually required to do. Since the work inside the for-loops were not intensively heavy, the time to create the threads even took longer than the computation they performed within the loops. Another pitfall of the naive approach was the misuse of the OpenMP library where instead of creating a team of threads at the beginning and utilizing them accordingly, a team of threads was created then re-destroyed for every single loop, aforementioned, with the syntax of `#pragma omp parallel` for. Another simple attempt to tune the code was to simply get rid of the function call to max within `compute_fg` speed. The initial code of: `dt = cfl/std::max(cx/dx, cy/dy)` was replaced by real `real dummy0=best_cx/dx, dummy1=best_cy/dy;dt=cfl/(dummy0>dummy1?dummy0:dummy1`. Also approved by the compiler optimization report, this change was able to improve our performance. The summary of the optimization can be seen below:

```
amplxe: Using result path `/home/b149/nw2/r004an'
amplxe: Executing actions 50 % Generating a report                              Function
        Module          CPU Time  Spin Time  Overhead Time
-------------------------------------------------------------------------------------------------------------------------------------------------------------
Central2D<Shallow2D, MinMod<float>>::limited_derivs                                                 shallow         1.152s         0s           0s
Central2D<Shallow2D, MinMod<float>>::compute_step                                                   shallow         0.577s         0s           0s
Central2D<Shallow2D, MinMod<float>>::compute_fg_speeds                                              shallow         0.199s         0s           0s
[Outside any known module]                                                                          [Unknown]       0.016s         0s           0s
_IO_fwrite                                                                                          libc-2.12.so    0.015s         0s           0s
_IO_file_xsputn                                                                                     libc-2.12.so    0.012s         0s           0s
SimViz<Central2D<Shallow2D, MinMod<float>>>::write_frame                                            shallow         0.005s         0s           0s
Central2D<Shallow2D, MinMod<float>>::run                                                            shallow         0.004s         0s           0s
Central2D<Shallow2D, MinMod<float>>::solution_check                                                 shallow         0.004s         0s           0s
std::array<float, (unsigned long)3>::operator[]                                                     shallow         0.003s         0s           0s
Central2D<Shallow2D, MinMod<float>>::solution_check                                                 shallow         0.001s         0s           0s
do_lookup_x                                                                                         ld-2.12.so      0.001s         0s           0s
std::array<float, (unsigned long)3>::operator[]                                                     shallow         0.001s         0s           0s
std::array<float, (unsigned long)3>::operator[]                                                     shallow         0.001s         0s           0s
std::vector<std::array<float, (unsigned long)3>, std::allocator<std::array<float, (unsigned long)3>>>::operator[]   shallow         0.001s         0s           0s
std::vector<std::array<float, (unsigned long)3>, std::allocator<std::array<float, (unsigned long)3>>>::operator[]   shallow         0.001s         0s           0s
amplxe: Executing actions 100 % done
```

Figure 7: *Original Code, ::run takes 0.004s*

```
        Module          CPU Time  Spin Time  Overhead Time
-------------------------------------------------------------------------------------------------------------------------------------------------------------
Central2D<Shallow2D, MinMod<float>>::limited_derivs                                                 shallow         1.119s         0s           0s
Central2D<Shallow2D, MinMod<float>>::compute_step                                                   shallow         0.557s         0s           0s
Central2D<Shallow2D, MinMod<float>>::compute_fg_speeds                                              shallow         0.195s         0s           0s
[Outside any known module]                                                                          [Unknown]       0.014s         0s           0s
_IO_file_xsputn                                                                                     libc-2.12.so    0.011s         0s           0s
SimViz<Central2D<Shallow2D, MinMod<float>>>::write_frame                                            shallow         0.009s         0s           0s
_IO_fwrite                                                                                          libc-2.12.so    0.009s         0s           0s
std::array<float, (unsigned long)3>::operator[]                                                     shallow         0.004s         0s           0s
Central2D<Shallow2D, MinMod<float>>::solution_check                                                 shallow         0.003s         0s           0s
Central2D<Shallow2D, MinMod<float>>::offset                                                         shallow         0.002s         0s           0s
Central2D<Shallow2D, MinMod<float>>::run                                                            shallow         0.002s         0s           0s
std::vector<std::array<float, (unsigned long)3>, std::allocator<std::array<float, (unsigned long)3>>>::operator[]   shallow         0.002s         0s           0s
Central2D<Shallow2D, MinMod<float>>::Central2D                                                      shallow         0.001s         0s           0s
Central2D<Shallow2D, MinMod<float>>::solution_check                                                 shallow         0.001s         0s           0s
SimViz<Central2D<Shallow2D, MinMod<float>>>::write_frame                                            shallow         0.001s         0s           0s
check_match.12445                                                                                   ld-2.12.so      0.001s         0s           0s
std::array<float, (unsigned long)3>::operator[]                                                     shallow         0.001s         0s           0s
std::vector<std::array<float, (unsigned long)3>, std::allocator<std::array<float, (unsigned long)3>>>::operator[]   shallow         0.001s         0s           0s
amplxe: Executing actions 100 % done
```

Figure 8: *Compiler Optimized Code, ::run takes 0.002s*

4.3 C Code

Finally, we switched to C code, which has optimized data structure of inner loops. The switch increases the serial code performance by a factor of approximately 4 for large case.

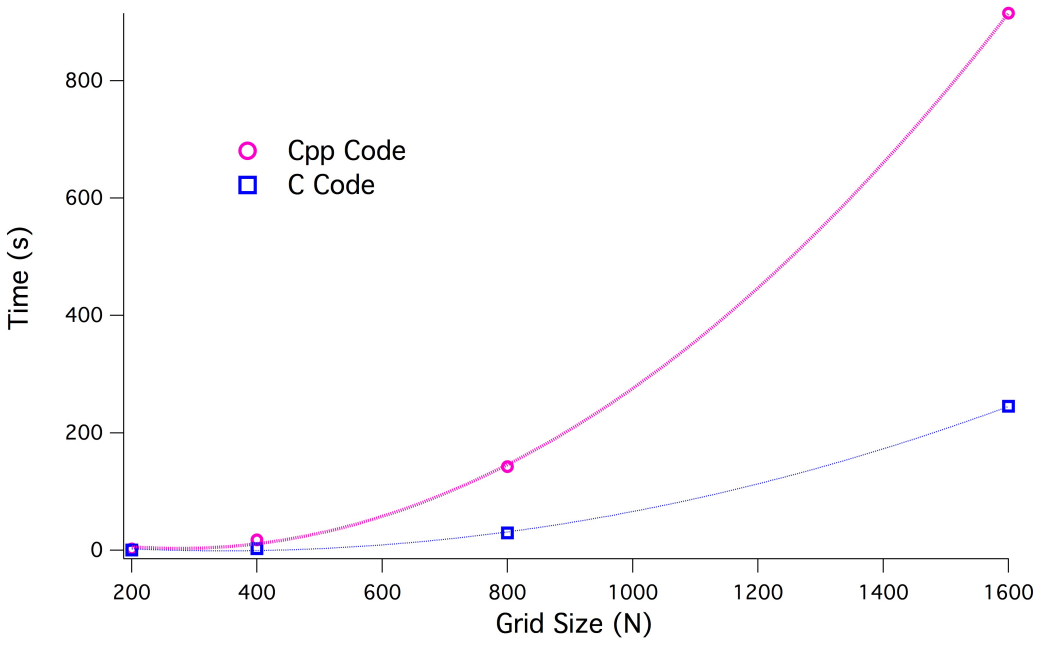

Figure 9: *Base C code performance vs Base C++ code performance*

Grid Size	C++ Version	C Version
200	2.227	0.362
400	17.627	2.735
800	142.585	29.531
1600	915.210	245.206

4.4 Implementation for Parallel Code

Despite the change from C++ code to C code, the essential idea remains the same. We did similar bottleneck and function testing to find out where the most time-consuming loops are. Then we implemented the domain decomposition in the C code and ran it using different threads. The results are compared using different grid sizes:

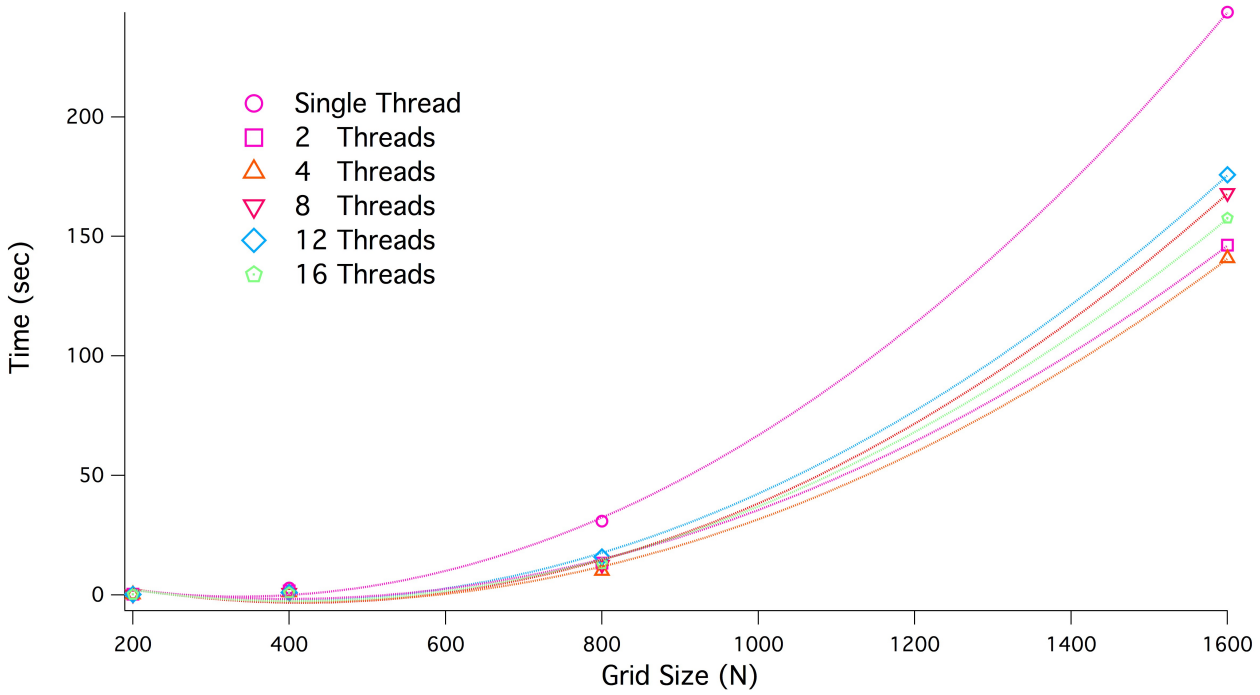

Figure 10: *Performance Comparison for Parallel Code*

Grid Size	1 Thread	2 Threads	4 Threads	8 Threads	12 Threads	16 Threads
200	0.438	0.283	0.194	0.170	0.236	0.170
400	2.826	1.688	1.070	0.816	1.020	0.839
800	30.903	13.274	9.986	12.627	15.878	13.096
1600	243.815	146.331	140.878	168.168	175.798	157.605

There is also some fine toning to the original code such that our single thread case runs faster than the base C code. It comes mostly from more careful memcpy usage and cutting down some overheads.

4.5 Strong Scaling

We ran the code on 800×800 grid size, the strong scaling results are plotted:

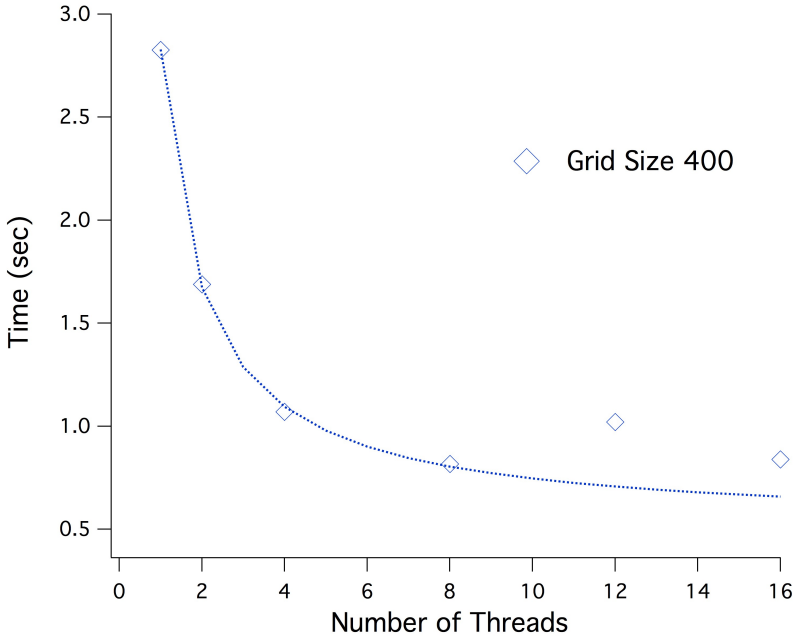

Figure 11: *Strong Scaling Plot*

Note that we left out the 12-core and 16-core cases in the curve fitting process. 12-core case is bad because it has odd multiples of cores and OpenMP tends to behave strangely. The 16-core slows down mainly because the increase in overhead cost. $50 = (800/16)$ cells in y direction per subdomain is allocated to each core and that is too small a batch size for the problem to compute efficiently in parallel.

4.6 Ghost Cell Size Tuning

We did not have time to perform a detailed study on tuning the ghost cell size, so we give some initial results here:

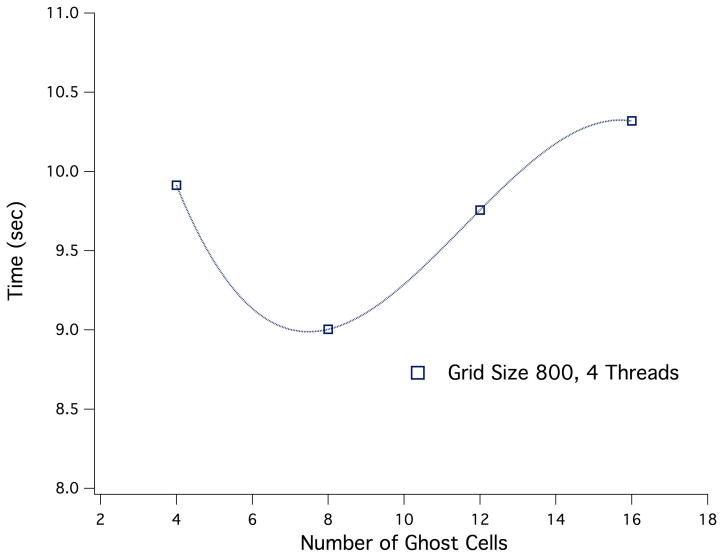

Figure 12: *Performance study for different ghost cell size*

	NG = 16	NG = 12	NG = 8	NG = 4
Time (in Sec)	10.319	9.757	9.004	9.914

The ghost cell size has to be a multiple of 4 to complete a layer. As we can see, there is about 10% performance difference in the rough study we do. We are using three ghost cell layers in our above simulation but this number should adapt to the actual subdomain size for each core to optimize the performance. Given more time, we can do more test runs with different ghost cell numbers, grid size and number of threads to determine a range for optimal number of ghost cells.

5 Conclusion

5.1 What Has Been Done

In this project, we studied the original C++ and C code and understood the advantages in using structured data arrays instead of arrays of data structure. Based on that, we designed and implemented our domain decomposition mechanism and implemented it for parallel computing.

For the serial code, we made some improvement for the memory traffic management. First, we increased ghost cell size to reduce the synchronization frequency. Then we switched the role of u and v in the odd/even `compute_step` function to avoid memory copy (idea and implementation from Prof. Bindel, we updated our code after implementing OpenMP).

For the parallel part, we kept the data structure for original domain and made minimal copy of it to feed to each thread. And in the synchronization and update parts, we tried to move as little memory as possible yet maintaining a regular computation pattern.

Currently, we get good performance boost from parallelization for small number of threads used (8 for most cases). The overhead costs start to show up for larger number of threads, mainly 24, used which makes the code less stable. We started with about 1.5 times performance increase each time we doubled the number of threads used. The ratio decreases, then reverses at around 10, which indicates where the overhead costs starts to dominate the computing time.

5.2 What We Propose to Do

Given more time, we want to fine tune the code and make the parallel code better by improving the inner computing loop. Here are some ideas we will try:

- Fine tune the number of ghost cells for different problem size and number of threads used. What is a good percentage for a given subdomain size. The paper *Explicity Shallow Water Simulations on GPUs: Guidelines and Best Practices* suggests ghost cell number on the order of 10 to optimize the performance.
- Look into function running time again to find the new bottleneck.
- Use `_mm_malloc()` instead to preallocate space for sub-domain data storage. The aligned memory block will help Intel compiler to further optimize our inner loop. However, we do not want to do this for each frame but once for the entire simulation. This can be done by changing the overall structure of code but should give us more speedup.
- The same for launching OpenMP thread team. We want to do it once in the beginning of program and end it at the very end.

- Since we are doing a MPI-like parallelization, we do not need to keep the main grid data at all but implicitly keeping them on each subgrid. The communication can then be done between adjacent processors. This may help us reduce the wait time.
- Lastly, we didn't have time to offload to Phi Board but it will be interesting to see how things scale on that. Moreover, Phi Board's have AVX-512 and we can get further vectorization boost for the inner loops.

6 Appendix

Our version of stepper.c (changed from C version of the shallow water code)

```
#include "step
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <assert.h>
#include <stdbool.h>
#include <omp.h>
//ldoc on
/**
 * ## Implementation
 *
 * ### Structure allocation
 */

central2d_t* central2d_init(float w, float h, int nx, int ny,
                            int nfield, flux_t flux, speed_t speed,
                            float cfl)
{
    // This should be an automatic update but I don't want to define global variable, change thi
    // The number of ghost celss should be 4*iter
    // where iter is the number of iteration before sync in central2d_step()
    int ng = 16; // # of ghost cells ( 4*iter for laziness and safety)

    central2d_t* sim = (central2d_t*) malloc(sizeof(central2d_t));
    sim->nx = nx; // dimension size in x
    sim->ny = ny;
    sim->ng = ng; // number of ghost cells
    sim->nfield = nfield; // each vector has three components
    sim->dx = w/nx; // Grid size in x
    sim->dy = h/ny;
    sim->flux = flux; // flux ???
    sim->speed = speed; // speed ???
    sim->cfl = cfl; // CFL prefix coefficient

    int nx_all = nx + 2*ng; // ghost cells on each side to avoid sync
    int ny_all = ny + 2*ng;
    int nc = nx_all * ny_all; // entire space
    int N  = nfield * nc; // how many entries for each vector
    sim->u  = (float*) malloc((4*N + 6*nx_all)* sizeof(float)); // allocate all space,not quite
    sim->v  = sim->u +  N; // storage space for half step grid
    sim->f  = sim->u + 2*N;
    sim->g  = sim->u + 3*N;
    sim->scratch = sim->u + 4*N;
```

```
    return sim;
}


void central2d_free(central2d_t* sim)
{
    free(sim->u);
    free(sim);
}


int central2d_offset(central2d_t* sim, int k, int ix, int iy)
{
    int nx = sim->nx, ny = sim->ny, ng = sim->ng;
    int nx_all = nx + 2*ng;
    int ny_all = ny + 2*ng;
    return (k*ny_all+(ng+iy))*nx_all+(ng+ix);
}


/**
 * ### Boundary conditions
 *
 * In finite volume methods, boundary conditions are typically applied by
 * setting appropriate values in ghost cells.  For our framework, we will
 * apply periodic boundary conditions; that is, waves that exit one side
 * of the domain will enter from the other side.
 *
 * We apply the conditions by assuming that the cells with coordinates
 * `nghost <= ix <= nx+nghost` and `nghost <= iy <= ny+nghost` are
 * "canonical", and setting the values for all other cells `(ix,iy)`
 * to the corresponding canonical values `(ix+p*nx,iy+q*ny)` for some
 * integers `p` and `q`.
 */

static inline
void copy_subgrid(float* restrict dst,
                  const float* restrict src,
                  int nx, int ny, int stride)
{
    // from src, copy nx * ny date subblock to dst.
    for (int iy = 0; iy < ny; ++iy)
        for (int ix = 0; ix < nx; ++ix)
            dst[iy*stride+ix] = src[iy*stride+ix]; // Variable stride is used to accomodate the
}
```

```
void central2d_periodic(float* restrict u,
                        int nx, int ny, int ng, int nfield)
{
    // Stride and number per field
    int s = nx + 2*ng; // the dimension size in x when ghost cells are present
    int field_stride = (ny+2*ng)*s; // the step size in y to cross the one subdomain

    // Offsets of left, right, top, and bottom data blocks and ghost blocks
    int l = nx,   lg = 0;
    int r = ng,   rg = nx+ng;
    int b = ny*s, bg = 0;
    int t = ng*s, tg = (nx+ng)*s; // should it be tg = (ny+ng)*s ? It doesn't matter for now bec

    // Copy data into ghost cells on each side
    for (int k = 0; k < nfield; ++k) {
        float* uk = u + k*field_stride;
        copy_subgrid(uk+lg, uk+l, ng, ny+2*ng, s); // for periodic condition update
        copy_subgrid(uk+rg, uk+r, ng, ny+2*ng, s);
        copy_subgrid(uk+tg, uk+t, nx+2*ng, ng, s);
        copy_subgrid(uk+bg, uk+b, nx+2*ng, ng, s);
    }
}


/**
 * ### Derivatives with limiters
 *
 * In order to advance the time step, we also need to estimate
 * derivatives of the fluxes and the solution values at each cell.
 * In order to maintain stability, we apply a limiter here.
 *
 * The minmod limiter *looks* like it should be expensive to computer,
 * since superficially it seems to require a number of branches.
 * We do something a little tricky, getting rid of the condition
 * on the sign of the arguments using the `copysign` instruction.
 * If the compiler does the "right" thing with `max` and `min`
 * for floating point arguments (translating them to branch-free
 * intrinsic operations), this implementation should be relatively fast.
 */


// Branch-free computation of minmod of two numbers times 2s
static inline
float xmin2s(float s, float a, float b) {
    float sa = copysignf(s, a);
    float sb = copysignf(s, b);
    float abs_a = fabsf(a);
```

```
    float abs_b = fabsf(b);
    float min_abs = (abs_a < abs_b ? abs_a : abs_b);
    return (sa+sb) * min_abs;
}


// Limited combined slope estimate
static inline
float limdiff(float um, float u0, float up) {
    const float theta = 2.0;
    const float quarter = 0.25;
    float du1 = u0-um;   // Difference to left
    float du2 = up-u0;   // Difference to right
    float duc = up-um;   // Twice centered difference
    return xmin2s( quarter, xmin2s(theta, du1, du2), duc );
}


// Compute limited derivs
static inline
void limited_deriv1(float* restrict du,
                    const float* restrict u,
                    int ncell)
{
    for (int i = 0; i < ncell; ++i)
        du[i] = limdiff(u[i-1], u[i], u[i+1]);
}


// Compute limited derivs across stride
static inline
void limited_derivk(float* restrict du,
                    const float* restrict u,
                    int ncell, int stride)
{
    assert(stride > 0);
    for (int i = 0; i < ncell; ++i)
        du[i] = limdiff(u[i-stride], u[i], u[i+stride]);
}


/**
 * ### Advancing a time step
 *
 * Take one step of the numerical scheme.  This consists of two pieces:
 * a first-order corrector computed at a half time step, which is used
 * to obtain new $F$ and $G$ values; and a corrector step that computes
```

```
 * the solution at the full step.  For full details, we refer to the
 * [Jiang and Tadmor paper][jt].
 *
 * The `compute_step` function takes two arguments: the `io` flag
 * which is the time step modulo 2 (0 if even, 1 if odd); and the `dt`
 * flag, which actually determines the time step length.  We need
 * to know the even-vs-odd distinction because the Jiang-Tadmor
 * scheme alternates between a primary grid (on even steps) and a
 * staggered grid (on odd steps).  This means that the data at $(i,j)$
 * in an even step and the data at $(i,j)$ in an odd step represent
 * values at different locations in space, offset by half a space step
 * in each direction.  Every other step, we shift things back by one
 * mesh cell in each direction, essentially resetting to the primary
 * indexing scheme.
 *
 * We're slightly tricky in the corrector in that we write
 * $$
 *   v(i,j) = (s(i+1,j) + s(i,j)) - (d(i+1,j)-d(i,j))
 * $$
 * where $s(i,j)$ comprises the $u$ and $x$-derivative terms in the
 * update formula, and $d(i,j)$ the $y$-derivative terms.  This cuts
 * the arithmetic cost a little (not that it's that big to start).
 * It also makes it more obvious that we only need four rows worth
 * of scratch space.
 */


// Predictor half-step
static
void central2d_predict(float* restrict v,
                       float* restrict scratch,
                       const float* restrict u,
                       const float* restrict f,
                       const float* restrict g,
                       float dtcdx2, float dtcdy2,
                       int nx, int ny, int nfield)
{
    float* restrict fx = scratch;
    float* restrict gy = scratch+nx;
    for (int k = 0; k < nfield; ++k) {
        for (int iy = 1; iy < ny-1; ++iy) {
            int offset = (k*ny+iy)*nx+1;
            limited_deriv1(fx+1, f+offset, nx-2);
            limited_derivk(gy+1, g+offset, nx-2, nx);
            for (int ix = 1; ix < nx-1; ++ix) {
                int offset = (k*ny+iy)*nx+ix;
                v[offset] = u[offset] - dtcdx2 * fx[ix] - dtcdy2 * gy[ix];
```

```
            }
        }
    }
}


// Corrector
static
void central2d_correct_sd(float* restrict s,
                          float* restrict d,
                          const float* restrict ux,
                          const float* restrict uy,
                          const float* restrict u,
                          const float* restrict f,
                          const float* restrict g,
                          float dtcdx2, float dtcdy2,
                          int xlo, int xhi)
{
    for (int ix = xlo; ix < xhi; ++ix)
        s[ix] =
            0.2500f * (u [ix] + u [ix+1]) +
            0.0625f * (ux[ix] - ux[ix+1]) +
            dtcdx2  * (f [ix] - f [ix+1]);
    for (int ix = xlo; ix < xhi; ++ix)
        d[ix] =
            0.0625f * (uy[ix] + uy[ix+1]) +
            dtcdy2  * (g [ix] + g [ix+1]);
}


// Corrector
static
void central2d_correct(float* restrict v,
                       float* restrict scratch,
                       const float* restrict u,
                       const float* restrict f,
                       const float* restrict g,
                       float dtcdx2, float dtcdy2,
                       int xlo, int xhi, int ylo, int yhi,
                       int nx, int ny, int nfield)
{
    assert(0 <= xlo && xlo < xhi && xhi <= nx);
    assert(0 <= ylo && ylo < yhi && yhi <= ny);

    float* restrict ux = scratch;
    float* restrict uy = scratch +   nx;
    float* restrict s0 = scratch + 2*nx;
```

```
    float* restrict d0 = scratch + 3*nx;
    float* restrict s1 = scratch + 4*nx;
    float* restrict d1 = scratch + 5*nx;

    for (int k = 0; k < nfield; ++k) {

        float*       restrict vk = v + k*ny*nx;
        const float* restrict uk = u + k*ny*nx;
        const float* restrict fk = f + k*ny*nx;
        const float* restrict gk = g + k*ny*nx;

        limited_deriv1(ux+1, uk+ylo*nx+1, nx-2);
        limited_derivk(uy+1, uk+ylo*nx+1, nx-2, nx);
        central2d_correct_sd(s1, d1, ux, uy,
                             uk + ylo*nx, fk + ylo*nx, gk + ylo*nx,
                             dtcdx2, dtcdy2, xlo, xhi);

        for (int iy = ylo; iy < yhi; ++iy) {

            float* tmp;
            tmp = s0; s0 = s1; s1 = tmp;
            tmp = d0; d0 = d1; d1 = tmp;

            limited_deriv1(ux+1, uk+(iy+1)*nx+1, nx-2);
            limited_derivk(uy+1, uk+(iy+1)*nx+1, nx-2, nx);
            central2d_correct_sd(s1, d1, ux, uy,
                                 uk + (iy+1)*nx, fk + (iy+1)*nx, gk + (iy+1)*nx,
                                 dtcdx2, dtcdy2, xlo, xhi);

            for (int ix = xlo; ix < xhi; ++ix)
                vk[iy*nx+ix] = (s1[ix]+s0[ix])-(d1[ix]-d0[ix]);
        }
    }
}


static
void central2d_step(float* restrict u, float* restrict v,
                    float* restrict scratch,
                    float* restrict f,
                    float* restrict g,
                    int io, int nx, int ny, int ng,
                    int nfield, flux_t flux, speed_t speed,
                    float dt, float dx, float dy)
{
    int nx_all = nx + 2*ng;
    int ny_all = ny + 2*ng;
```

```
    float dtcdx2 = 0.5 * dt / dx;
    float dtcdy2 = 0.5 * dt / dy;

    flux(f, g, u, nx_all * ny_all, nx_all * ny_all);

    central2d_predict(v, scratch, u, f, g, dtcdx2, dtcdy2,
                      nx_all, ny_all, nfield);

    // Flux values of f and g at half step
    for (int iy = 1; iy < ny_all-1; ++iy) {
        int jj = iy*nx_all+1;
        flux(f+jj, g+jj, v+jj, nx_all-2, nx_all * ny_all);
    }
    //flux(f, g, v, nx_all * ny_all, nx_all * ny_all);

    central2d_correct(v + io*(nx_all+1), scratch, u, f, g, dtcdx2, dtcdy2,
                      //1-io, nx+2*ng-io,
                      //1-io, ny+2*ng-io,
                      ng-io, nx+ng-io,
                      ng-io, ny+ng-io,
                      nx_all, ny_all, nfield);

    // Copy from v storage back to main grid
    //memcpy(u, v, nfield*ny_all*nx_all*sizeof(float)); // copy everything instead, do not updat
    //memcpy(u+(ng   )*nx_all+ng,
           //v+(ng-io)*nx_all+ng-io,
           //(nfield*ny_all-ng) * nx_all * sizeof(float));
}

// My subdomain copy function
void copy_subdomain(float ** u_s, float ** v_s,
                float ** f_s, float ** g_s, float ** scratch_s,
                int *ny_sub, float* restrict u, int nx, int ny, int ng,
                int nfield, int index, int ndomain){
        int new_ny; // The size of subdomain
        int subsize = ceil( ny / ndomain);
        if(index < ndomain-1) new_ny = subsize;
        else new_ny = ny - (ndomain-1)*subsize;
        (*ny_sub) = new_ny;

        int nx_all = nx + 2*ng;
        int ny_all = new_ny + 2*ng;
        int nc = nx_all * ny_all;
        int N = nfield * nc;
        int start_index = index*subsize*nx_all;
        int Nc = nx_all * (ny + 2*ng);
```

```
        // Maybe use _mm_malloc for aligned memory block, suggestion for later
        float* u_new = (float*) malloc((4*N + 6*nx_all)*sizeof(float));
        *u_s = (float*)u_new;
        *v_s = (float*)u_new + N;
        *f_s = (float*)u_new + 2*N;
        *g_s = (float*)u_new + 3*N;
        *scratch_s = (float*)u_new + 4*N;

        for (int k = 0; k < nfield; k++){
                memcpy(*u_s + nc*k, u + start_index + Nc*k, nc*sizeof(float));
                //memcpy(*v_s + nc*k, u + start_index + Nc*k, nc*sizeof(float));
                //memcpy(*f_s + nc*k, u + start_index + Nc*k, nc*sizeof(float));
                //memcpy(*g_s + nc*k, u + start_index + Nc*k, nc*sizeof(float));
        }
}

void sync_subdomain(float* restrict u_s, float* restrict u,
                int ny_sub, int nx, int ny, int ng,
                int nfield, int index, int ndomain){
        int nx_all = nx + 2*ng;
        int ny_all_sub = ny_sub + 2*ng;
        int nc = nx_all * ny_all_sub;
        int Nc = nx_all * (ny + 2*ng);
        int subsize = ceil(ny / ndomain);
        int start_index = (index*subsize + ng)*nx_all;
        int sub_start = ng*nx_all;
        //printf("For processor %d/%d, the values are ny_sub = %d, start_index = %d, sub_start =
        for (int k = 0; k < nfield ; k++){
                memcpy(u+start_index+Nc*k, u_s+sub_start+nc*k, nx_all*ny_sub*sizeof(float));
        } // Copy the real data part back to the main grid.
}
void update_subdomain(float* restrict u_s, float* restrict u,
                int ny_sub, int nx, int ny, int ng,
                int nfield, int index, int ndomain){
        int s = nx + 2*ng;
        int field_stride = (ny+2*ng)*s;
        int sub_field_stride = (ny_sub+2*ng)*s;

        int l = 0, lg = 0;
        int r = nx+ng, rg = nx+ng;
        int b = (ny_sub+ng)*s, bg = (ny_sub+ng)*s;
        int t = 0, tg = 0; // I think this is correct?

        int subsize = ceil(ny / ndomain);
        int start_index = (index*subsize)*s; // no ng because of the other index system we have
        //printf("For processor %d/%d, the values are ny_sub = %d, start_index = %d\n", index, r
        for (int k = 0; k < nfield; k++){
```

```
                float* uk = u + k*field_stride + start_index;
                float* u_sk = u_s + k*sub_field_stride;
                copy_subgrid(u_sk+lg, uk+l, ng, ny_sub + 2*ng, s);
                copy_subgrid(u_sk+rg, uk+r, ng, ny_sub + 2*ng, s);
                copy_subgrid(u_sk+tg, uk+t, nx + 2*ng, ng, s);
                copy_subgrid(u_sk+bg, uk+b, nx + 2*ng, ng, s);
        }
}
/**
 * ### Advance a fixed time
 *
 * The `run` method advances from time 0 (initial conditions) to time
 * `tfinal`.  Note that `run` can be called repeatedly; for example,
 * we might want to advance for a period of time, write out a picture,
 * advance more, and write another picture.  In this sense, `tfinal`
 * should be interpreted as an offset from the time represented by
 * the simulator at the start of the call, rather than as an absolute time.
 *
 * We always take an even number of steps so that the solution
 * at the end lives on the main grid instead of the staggered grid.
 */

static
int central2d_xrun(float* restrict u, float* restrict v,
                   float* restrict scratch,
                   float* restrict f,
                   float* restrict g,
                   int nx, int ny, int ng,
                   int nfield, flux_t flux, speed_t speed,
                   float tfinal, float dx, float dy, float cfl)
{
    // OMP session should start here
    // sub-domain parallel

    int nstep = 0;
    int nx_all = nx + 2*ng;
    int ny_all = ny + 2*ng;
    bool done = false;
    float t = 0;

    // Initialize the new subdomain vectors here.
#ifdef _OPENMP
    int num_threads_used = 4;
    omp_set_num_threads(num_threads_used);

    float* u_sub[num_threads_used];
    float* v_sub[num_threads_used];
```

```
    float* f_sub[num_threads_used];
    float* g_sub[num_threads_used];
    float* scratch_sub[num_threads_used];
    int ny_sub[num_threads_used];
    for (int index = 0; index < num_threads_used; index++){
            copy_subdomain( &u_sub[index], &v_sub[index], &f_sub[index], &g_sub[index], &scratch
    }
    float dt;
#pragma omp parallel shared(dt,dt)
    while (!done) {
//#pragma omp single
        float cxy[2] = {1.0e-15f, 1.0e-15f};
        central2d_periodic(u, nx, ny, ng, nfield);
        speed(cxy, u, nx_all * ny_all, nx_all * ny_all);
        //float dt = cfl / fmaxf(cxy[0]/dx, cxy[1]/dy);
        dt = cfl / fmaxf(cxy[0]/dx, cxy[1]/dy);
        // For loops
        int it;
        int iter = 4;
        int idx = omp_get_thread_num();
//#pragma omp barrier
        update_subdomain(u_sub[idx], u, ny_sub[idx], nx, ny, ng, nfield, idx, num_threads_used);
        for(it = 0; it < iter; it ++){
            if (t + 2*dt >= tfinal) {
                dt = (tfinal-t)/2;
                done = true;
            }
            //central2d_step(u_sub[idx], v_sub[idx], scratch_sub[idx],
            //               f_sub[idx], g_sub[idx],
            //               0, nx, ny_sub[idx], ng,
            //               nfield, flux, speed,
            //               dt, dx, dy);
            //central2d_step(u_sub[idx], v_sub[idx], scratch_sub[idx],
            //               f_sub[idx], g_sub[idx],
            //               1, nx, ny_sub[idx], ng,
            //               nfield, flux, speed,
            //               dt, dx, dy);
            //central2d_step(u_sub[idx], v_sub[idx], scratch_sub[idx],
            //               f_sub[idx], g_sub[idx],
            //               0, nx, ny_sub[idx], ng,
            //               nfield, flux, speed,
            //               dt, dx, dy);
            //central2d_step(v_sub[idx], u_sub[idx], scratch_sub[idx],
            //               f_sub[idx], g_sub[idx],
            //               1, nx, ny_sub[idx], ng,
            //               nfield, flux, speed,
            //               dt, dx, dy);
```

```
            int ng_eff = 4 * (iter-1-it);
            central2d_step(u_sub[idx], v_sub[idx], scratch_sub[idx],
                           f_sub[idx], g_sub[idx],
                           0, nx+4+2*ng_eff, ny_sub[idx]+4+2*ng_eff, ng-2-ng_eff,
                           nfield, flux, speed,
                           dt, dx, dy);
            central2d_step(v_sub[idx], u_sub[idx], scratch_sub[idx],
                           f_sub[idx], g_sub[idx],
                           1, nx+2*ng_eff, ny_sub[idx]+2*ng_eff, ng-ng_eff,
                           nfield, flux, speed,
                           dt, dx, dy);
#pragma omp single
            t += 2*dt;
            nstep += 2;
        }
        sync_subdomain(u_sub[idx], u, ny_sub[idx], nx, ny, ng, nfield, idx, num_threads_used);
#pragma omp barrier
    }
    // Free the subdomain vectors.
    for (int index = 0; index < num_threads_used; index++){
            free(u_sub[index]);
    }
#else
    while (!done) {
        float cxy[2] = {1.0e-15f, 1.0e-15f};
        central2d_periodic(u, nx, ny, ng, nfield); // Apply periodic boundary condition
        speed(cxy, u, nx_all * ny_all, nx_all * ny_all);
        float dt = cfl / fmaxf(cxy[0]/dx, cxy[1]/dy);

        // For loops
        int it;
        int iter = 3;
        for(it = 0; it < iter; it ++){
            if (t + 2*dt >= tfinal) {
                dt = (tfinal-t)/2;
                done = true;
            }
            //central2d_step(u, v, scratch, f, g,
            //               0, nx, ny, ng,
            //               nfield, flux, speed,
            //               dt, dx, dy);
            ////central2d_periodic(u, nx, ny, ng, nfield);
            //central2d_step(u, v, scratch, f, g,
            //               1, nx, ny, ng,
            //               nfield, flux, speed,
            //               dt, dx, dy);
            int ng_eff = 4 * (iter-1-it);
```

```
            central2d_step(u_sub[idx], v_sub[idx], scratch_sub[idx],
                           f_sub[idx], g_sub[idx],
                           0, nx+4+2*ng_eff, ny_sub[idx]+4+2*ng_eff, ng-2-ng_eff,
                           nfield, flux, speed,
                           dt, dx, dy);
            central2d_step(v_sub[idx], u_sub[idx], scratch_sub[idx],
                           f_sub[idx], g_sub[idx],
                           1, nx+2*ng_eff, ny_sub[idx]+2*ng_eff, ng-ng_eff,
                           nfield, flux, speed,
                           dt, dx, dy);
            t += 2*dt;
            nstep += 2;
        }
    }
#endif
    return nstep;
}


int central2d_run(central2d_t* sim, float tfinal)
{
    return central2d_xrun(sim->u, sim->v, sim->scratch,
                          sim->f, sim->g,
                          sim->nx, sim->ny, sim->ng,
                          sim->nfield, sim->flux, sim->speed,
                          tfinal, sim->dx, sim->dy, sim->cfl);
}

// My own function (not in use)
//central2d_t* central2d_copy_subdomain(central2d_t* sim, int index, int ndomain)
//{
//        int new_ny; // The size of subdomain.
//        int subsize = ceil(sim->ny / ndomain);
//        if(index < ndomain - 1) new_ny = subsize;
//        else new_ny = sim->ny - (ndomain - 1) * subsize;
//
//        central2d_t* copied_sim = (central2d_t*) malloc(sizeof(central2d_t));
//        copied_sim->nx = sim->nx; // nx should be the same
//        copied_sim->ny = new_ny;
//        copied_sim->ng = sim->ng; // Number of ghost cells must be the same for boundary condi
//        copied_sim->nfield = sim->nfield;
//        copied_sim->dx = sim->dx; // Grid size in x
//        copied_sim->dy = sim->dy;
//        copied_sim->flux = sim->flux;
//        copied_sim->speed = sim->speed;
//        copied_sim->cfl = sim->cfl; // CFL prefix coefficient
//
```

```
//        int nx_all = copied_sim->nx + 2*copied_sim->ng; // ghost cells on each side to avoid s
//        int ny_all = copied_sim->ny + 2*copied_sim->ng;
//        int nc = nx_all * ny_all; // entire domain subspace including ghost cells on each side
//        int N  = copied_sim->nfield * nc; // how many entries for each vector
//        copied_sim->u  = (float*) malloc((4*N + 6*nx_all)* sizeof(float)); // new space for u
//        copied_sim->v  = copied_sim->u + N;
//        copied_sim->f  = copied_sim->u + 2*N;
//        copied_sim->g  = copied_sim->u + 3*N;
//        copied_sim->scratch = copied_sim->u + 4*N;
//        // Stride and number per field
//        int start_index = index*subsize*nx_all; // Starting from the index subdomain with the
//        int Nc = nx_all * (sim->ny + 2*sim->ng);
//        for (int k = 0; k < copied_sim->nfield; k++){
//            memcpy(copied_sim->u + nc*k, sim->u+start_index + Nc*k, nc);
//            memcpy(copied_sim->v + nc*k, sim->v+start_index + Nc*k, nc);
//            memcpy(copied_sim->f + nc*k, sim->f+start_index + Nc*k, nc);
//            memcpy(copied_sim->g + nc*k, sim->g+start_index + Nc*k, nc);
//        } // Copy all the memory for subdomain
//        return copied_sim;
//}
//
//void sync_subdomain(float* restrict u, float* restrict u_sub, int nx, int Ny, int ny, int ng,
//        int nx_all = nx + 2*ng;
//        int ny_all = ny + 2*ng;
//        int nc = nx_all * ny_all;
//        int Nc = nx_all * (Ny + 2*ng);
//        int subsize = ceil(nx / ndomain); // This should be ny, but for this special case, nx=
//        int start_index = (index*subsize + ng)*nx_all;
//        int sub_start = ng*nx;
//        for (int k = 0; k < nfield; k++){
//                memcpy(u + start_index + Nc*k, u_sub + sub_start + nc*k, nx_all*ny);
//        } // Copy the real data part back to the main grid.
//}
```

Output from Amplxe-cl demonstrating the CPU times spent in major functions where the grid size is 400, running on 4 threads

Function	Module	CPU Time
Time:Communication (MPI)	Spin Time:Other	Overhead Time:Creation (OpenMP)	Overhead Ti
central2d_step	lshallow	1.125s
0s	0s	0s	0s
shallow2d_flux	lshallow	0.714s
0s	0s	0s	0s
central2d_correct	lshallow	0.540s
0s	0s	0s	0s
__intel_ssse3_rep_memcpy	lshallow	0.522s
0s	0s	0s	0s
__kmp_wait_template<kmp_flag_64>	libiomp5.so	0.492s
0s	0s	0s	0s
[Outside any known module]	[Unknown]	0.253s
0s	0s	0s	0s
limited_deriv1	lshallow	0.248s
0s	0s	0s	0s
xmin2s	lshallow	0.222s
0s	0s	0s	0s
shallow2d_speed	lshallow	0.140s
0s	0s	0s	0s
limited_derivk	lshallow	0.101s
0s	0s	0s	0s
central2d_correct_sd	lshallow	0.099s
0s	0s	0s	0s
__kmp_wait_template<kmp_flag_64>	libiomp5.so	0.097s
0s	0s	0s	0s
xmin2s	lshallow	0.095s
0s	0s	0s	0s
xmin2s	lshallow	0.070s
0s	0s	0s	0s
limdiff	lshallow	0.064s
0s	0s	0s	0s
central2d_periodic	lshallow	0.037s
0s	0s	0s	0s
copy_subgrid	lshallow	0.026s
0s	0s	0s	0s
central2d_correct_sd	lshallow	0.024s
0s	0s	0s	0s
run_sim	lshallow	0.022s
0s	0s	0s	0s
limdiff	lshallow	0.020s
0s	0s	0s	0s
kmp_basic_flag<unsigned long long>::notdone_check	libiomp5.so	0.017s
0s	0s	0s	0s
luaV_execute	lshallow	0.015s
0s	0s	0s	0s
__kmp_x86_pause	libiomp5.so	0.013s
0s	0s	0s	0s
__kmp_yield	libiomp5.so	0.012s

Figure 13: *Function timing for Grid of size 400, running on 4 threads*