CS5220: Shallow Water Project

Junyoung Lim (jl3248), Cheng Perng Phoo(cp598), Robert Stephany (rrs254)

November 3, 2020

1 Introduction

In this project, we parallelized a finite volume solver for the shallow water equations on a rectangular grid with periodic boundary conditions. We also explored ways to optimize and tune the parallel solver. Our implementation is based on a finite volume solver for 2D hyperbolic PDEs via a high-resolution finite difference scheme by Jiang and Tadmor. Our approach is built upon two central themes: sub-domain partitioning and parallelization using OpenMP. Based on the performance analysis, our final approach achieved an almost three-fold speedup over a tuned serial implementation. In particular, we observed a twofold speedup on two processors (See 4.1). Our implementation can be found at: `https://github.com/cpphoo/shallow-water`.

2 Sub-domain Partitioning

In this section, we focus on the high-level implementation logic. We defer details on OpenMP parallelization to the next section (section 3). Throughout this report, we will let U_G denote the global grid of cells upon which we want to simulate the shallow water equations. U_G is partitioned into N sub-grids $\{U_i\}_{i=1}^{N}$. A team of N threads solves the shallow water equations on the subgrids (each thread gets a subgrid).

2.1 Maintaining Sub-grids Consistency Via a Global Grid U_G

The underlying PDE solver updates the water height and momentum in a cell using the corresponding values in each of the four cells which border it. This means that, for a thread to update the cells on the border of its subdomain, it must be able to access must access cells outside of its sub-grid. Fortunately, however, this also means that each thread only needs access to the cells which are adjacent to the boundary of its sub-grid. To ensure that each thread has access to the information it needs to update its boundary cells, we pad each U_i with a few layers of "ghost" cells. The ghost cells essentially hold copies of the cells (from other sub-grids) that the thread needs to update its sub-grid. We periodically update the ghost cells using the global grid U_g, which acts as a shared repository through which the threads can communicate.

When the threads update their "ghost" cells, they first push the first and last few rows and columns of their sub-grid to their corresponding locations in the global grid (we do this because the other threads will need access to these cells when updating their ghost cells). Once this finishes, we apply periodic boundary conditions to U_G. After that, each thread updates the momentum and water height in its "ghost" cells the values in the corresponding cells in the global grid (which, at this point, have been updated by the other threads). This order of operation ensures that the periodic BCs are applied and that each thread's "ghost" cells are consistent enough with the cells (in other sub-grids) that they correspond to for the thread to correctly update its sub-grid.

2.2 Determining dt for Finite Difference

Once we have applied the boundary conditions are applied and each thread has updated its "ghost cells", we compute dt. Determining dt in our parallel implementation is trickier than in the serial one. The serial implementation calculates dt by finding the maximum wave speed across U_G. In our parallel implementation, however, U_G is not updated after each time step. Thus, in our parallel implementation, each thread finds the maximum wave velocity in its sub-grid U_i. It then uses this maximum to determine dt_i, which would be the value of dt necessary to ensure stability on U_i. Once each thread has determined its dt_i, we compute dt to be $\min\{dt_i\}_{i=1}^N$ using a shared buffer. We do this to ensure that each thread uses the same time step and that this time step will give stability on each subgrid. This approach is equivalent to if a single thread computed the global maximum wave velocity, used that maximum to compute dt, and then broadcast dt to the other threads. Therefore, the value of dt that our approach calculates is the same dt that the serial code would have calculated.

2.3 Partitioning the Domain

We set the number of partitions N to be the number of threads running p. For simplicity, we decided to split the grid data row-wise. This allows us to exploit the fact that the grid data (or matrices) follows row-major ordering, thereby improving cache usage when the threads read from and write to U_G

3 OpenMP Parallelization

Sub-domain partitioning involves handling N disjoint partitions that can be simulated in parallel. To enable parallelism, we used a shared memory model. We implemented this model using OpenMP. In this section, we elaborate a few key implementations that enables parallelization of sub-domain partitioning.

3.1 Initialization of Each Thread

Once we determine the number of threads p, we initialize each U_i by making a deep copy of partition of the original grid data. After this, we use the original grid data as our global grid U_G. Before the simulation starts, it is crucial to ensure that all threads finish up their U_i initialization. This is implemented using a barrier. See below for a pseudocode of our thread initialization:

```
// ldriver_parallel.c
// n_rows = p
// n_cols = 1
#pragma omp parallel num_threads(n_rows*n_cols)
{
    // use omp_get_thread_num() to compute the sub grid location
    // call central2d_init(...)
    // initialize the U_i for each thread

    // wait for all threads to set up their local arrays.
    #pragma omp barrier
    ...
}
```

3.2 Computing dt via a Shared Buffer

To update each cell, we need to compute dt for the finite difference. To speed up the computation, we attempted to parallelize computation as described in section 2 which requires a shared memory buffer. As such, we let the first-arriving thread to create the shared buffer with a single directive and store its local dt in said buffer. For the rest of the threads, they would update the buffer if their local dt is smaller than the value stored in the buffer. We establish a critical section when updating the buffer and set a barrier to wait for all threads to finish updating the buffer before proceeding. See below for a pseudocode of our implementation:

```c
// inside central2d_xrun(...) of stepper_parallel.c
    ...
    // calculate dt_local using the data on our partition.

    // Initialize the shared buffer, because the buffer may contain 0.
    #pragma omp single
    {
      shared_buffer[0] = dt_local;
    }

    // Now, each thread writes its data to shared_buffer one by one.
    #pragma omp critical
    {
      shared_buffer[0] = fmin(dt_local, shared_buffer[0]);
    }
    #pragma omp barrier
    ...
```

3.3 Respecting the Boundary Conditions

After each sub-grid are updated, we copy U_i (on each thread) to the global grid U_G so that we could apply the periodic boundary conditions on the global grid. To make sure that U_G are completely updated, we again establish a barrier as shown as follows:

```c
// inside central2d_xrun(...) of stepper_parallel.c
    ...
    // copy boundary of U to corresponding entries of U global using
   central2d_local_to_global(...)
    #pragma omp barrier
    ...
```

3.4 Checking for Correctness and Generating Visualization

The solution check and visualization generation are performed on the global grid U_G.

4 Performance Analysis

In this section, we assume p is the number of threads and each thread is uniquely associated to a processor.

4.1 Strong Scaling

We performed strong scaling on Graphite. In particular, we set the problem size to be $nx = ny = 1000$ and ran our implementation for $p = 1, 2, 3, \ldots, 10$. We computed the speedup as $T_{serial}/T_{parallel}$ where T_{serial} was benchmarked using the initial code released to us. We plotted the speedup against the number of threads in figure 1. Our strong scaling plot demonstrated a close to linear scaling for the number of threads less than or equal to 5 but the speedup tapered off as we increased the number of threads due to inparallelizable portion of the code. Our finding aligned with Amdahl's Law, i.e, there always exists some code that is inherently serial and hard to parallelize and those serial work upper bound the speedup of parallel code.

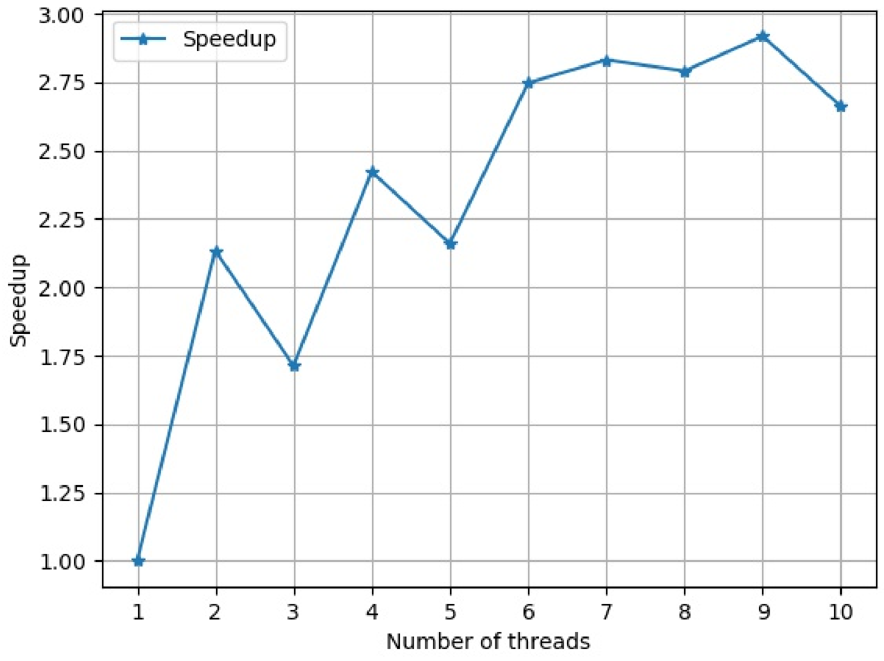


Figure 1: Speedup for Strong Scaling Study

4.2 Weak Scaling

To run weak scaling, we had to scale the problem size with respect to the number of threads such that the amount of work per processor is identical for each thread. However, this would be be hard to achieve if we wanted to maintain a square grid data. For instance, suppose we conducted weak scaling on a single thread on a grid of size $h \times h$, then to run weak scaling with 2 threads, we have to run on a grid of size $2h \times h$ or $h \times 2h$ which was no longer a square grid. As such, we conducted two variants of weak scaling:

1. Maintaining the square grid
   To make sure that we could conduct weak scaling on a square grid data, we made a compromise by setting the grid's height and width to be $\lfloor \sqrt{p} \rfloor \times 500$. We plot the scaled speedup in figure 2. The scaled speedup hovered around 2 for $p \geq 2$ (the trend line did show that the scaled speedup increased linearly with respect to p but the effect of p is negligible for small p). Our finding was consistent with Gustafson's Law, i.e, the scaled speedup is $O(p)$.

2. Keeping the amount of work per processor identical

   To ensure similar workload for each processor, we conducted weak scaling on grid data of size $500 \times 500p$ (violating the square grid assumption) so that each thread would operate on a grid data of size 500×500. We plotted the scaled speedup in figure 3. Again, we found that the scaled speedup hovered around 2 for $p \geq 2$ with linear dependence with the number of threads. Our finding was again consistent with Gustafson's Law.

5 Optimization Attempts, Potential Future Improvements

In this section, we discuss some of the optimizations that we attempted, and discuss changes that we believe could improve our implementation but that we did not have time to implement oursevles.

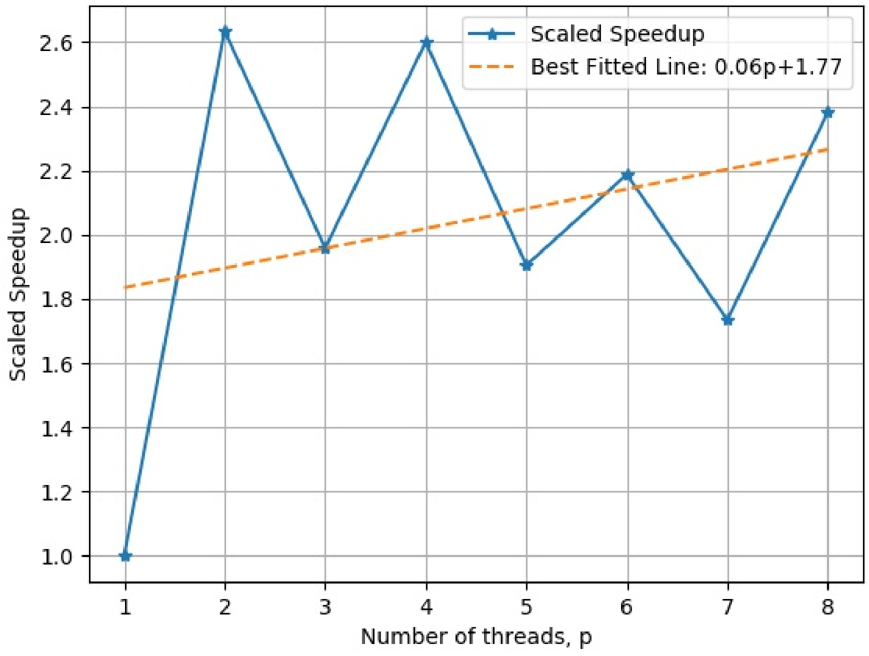


Figure 2: Speedup for Weak Scaling (Square Grid)

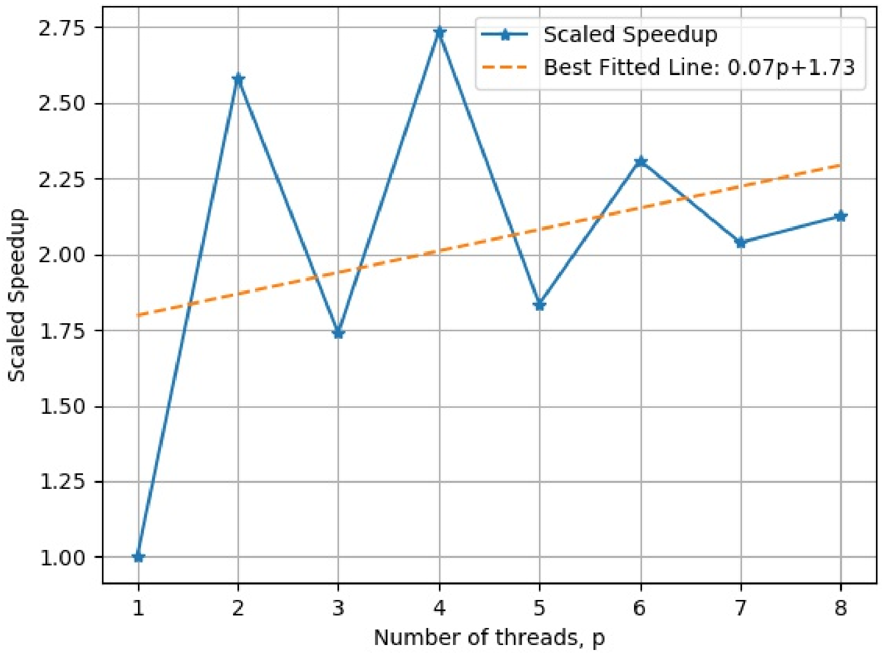


Figure 3: Speedup for Weak Scaling (Constant Workload for Processor)

5.1 Naively Inserting #pragma omp parallel

We started the project by profiling the serial code provided using gprof (see figure 4) and tried parallelizing a few functions using OpenMP. In particular, we tried adding `#pragma omp parallel for` in three functions: `central2d_correct_sd`, `shallow_2dv_flux` and `shallow_2dv_speed` but we ended up with an implementation that were significantly slower than the serial code.

5.2 Parallelizing Periodic BCs

Our parallel implementation performs most computations in parallel. However, some steps of our process happen sequentially. In particular, the Periodic BC's on U_G are applied by a single processor. We chose this approach initially because it was easy to implement. In theory, however, this process can happen in parallel. We tried rewriting `central2d_periodic` with `parallel for` directives. This change failed to improve runtime measurably. We believe that this is because applying the BCs represents a tiny fraction of the overall runtime. Given these results, we decided to use the serial version of `central2d_periodic`. We chose to do this because we knew that the serial version worked, would be easier to maintain, and because doing so did not appear to reduce performance.

5.3 Nested Parallelization

We also tried adding parallel regions within the predictor/corrector functions of our parallel implementation (these were, therefore, nested parallel regions). The idea here was that since the code spent most of its runtime in the predictor/corrector methods, and since those methods boil down to big for loops, we might be able to improve performance by having each thread spawn a new team of threads (which we will call 'sub-threads') which could run a thread's loops in parallel. To do this, we tried adding a `pragma omp parallel for` directive within some of these functions. This approach failed to improve runtime. This result makes sense if we consider what's going on.

First, let's consider the case when the number of threads is much less than the number of CPU cores. In this case, our setup is very similar to the serial case. As discussed in the previous sub-section, adding parallel for loops to the predictor-corrector methods in the serial implementation failed to improve runtime (probably because the cost of spawning a team of threads outweighed the benefit in parallelization). Thus, it makes sense that sub-threads in this case also failed to improve runtime.

Now let's consider the case when the number of threads is close to the number of CPU cores. In this case, there isn't much opportunity for further parallelization. A thread can spawn a team of sub-threads, but the number of CPU cores stays fixed. Thus, if there are N cores available, then only N sub-threads can run in parallel.

Given this analysis, it is not surprising that this change failed to improve runtime. After seeing these results, we abandoned this approach and instead focused on trying to use MPI with OpenMP (which is the topic of the next section).

5.4 Using OpenMP With MPI

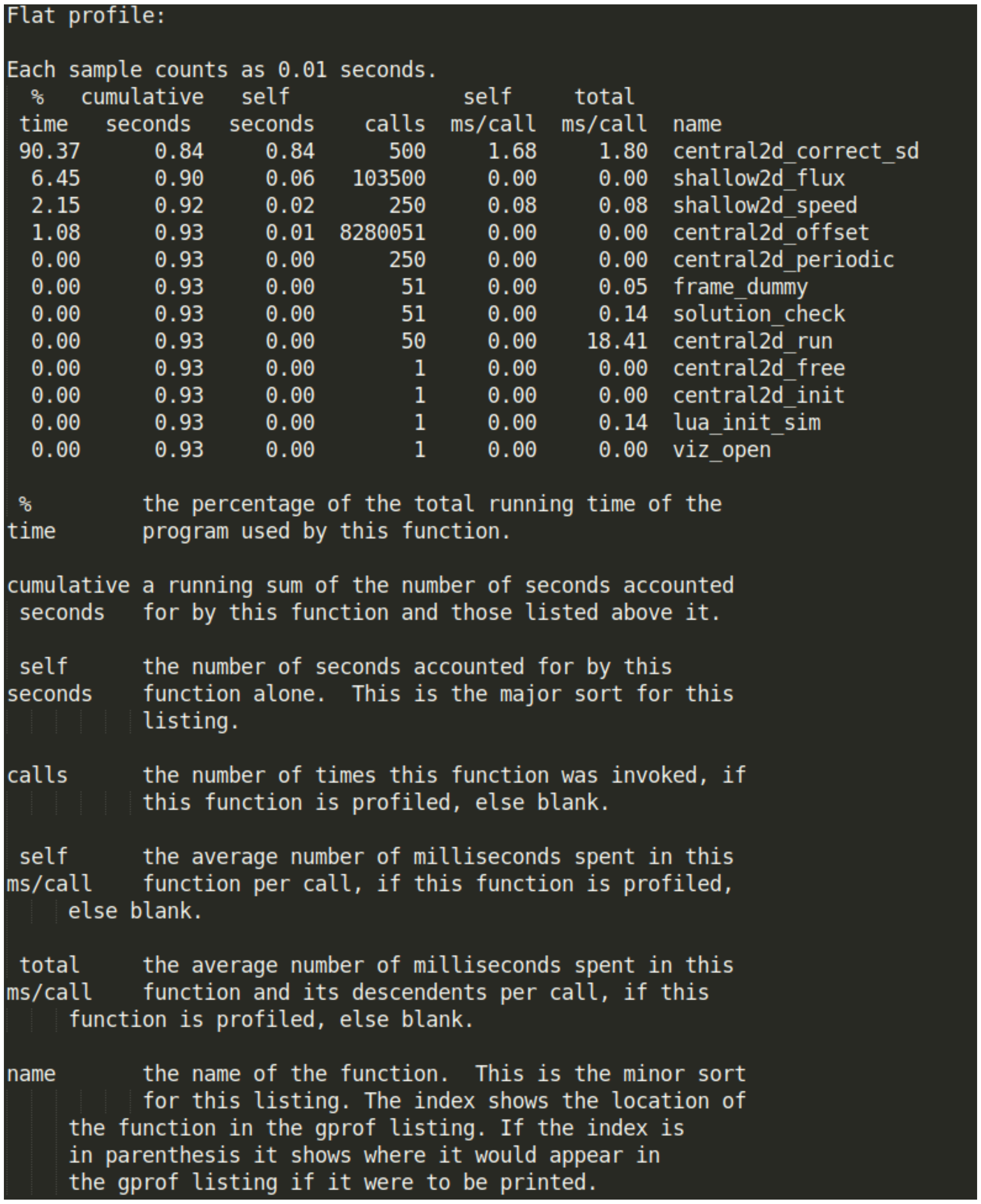
After we got our OpenMP based approach running, we started working on a hybrid OpenMP and MPI strategy. Via this strategy, we would have used MPI to partition the domain (which in our opinion, is a problem that is better suited for a distributed memory setup) and OpenMP to further parallelize the computations on each piece of the partition (possibly with a second level of sub-grids). This approach would allow us to run larger global grids. One downside to the implementation described above was that it relies on the existence of a global grid that can fit into memory. If the global grid is too large, it may not be possible to store the entire thing in memory simultaneously. Our approach with MPI promised to alleviate this bottleneck by eliminating the global grid altogether.

In our MPI based approach, each processor would only hold its local grid. To make this work, we needed to rewrite our implementation so that it did not rely on a global grid. As described in previous sections, our implementation used the global grid as a repository through which the threads could communicate, apply global periodic boundary conditions, run the solution check, and simplify the visualizer code. Switching to MPI, therefore, required us to implement these features without a global grid.

Some of these features were natural to implement in MPI. For example, to communicate between processors, we could have the processors send their boundary data directly to one another rather than using the global grid as a buffer.

Others, however, were difficult to implement in MPI. In particular, the visualizer code posed an interesting challenge. Since there is only one visualizer file, data needs to be written to it as if it were being written row-for-row from a global grid. In our approach, a single row of the global grid could have been distributed over multiple processors. Therefore, getting the processors to write to the file as if it were written row-for-row from a global grid presented a complicated coordination problem.

Unfortunately, we were not able to get these modifications finished before the original two-week project deadline. As such, we stuck with our original OpenMP based approach. If we had more time to work on this project, we likely would have focused on implementing this hybrid approach.



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 90.37      0.84     0.84      500     1.68     1.80  central2d_correct_sd
  6.45      0.90     0.06   103500     0.00     0.00  shallow2d_flux
  2.15      0.92     0.02      250     0.08     0.08  shallow2d_speed
  1.08      0.93     0.01  8280051     0.00     0.00  central2d_offset
  0.00      0.93     0.00      250     0.00     0.00  central2d_periodic
  0.00      0.93     0.00       51     0.00     0.05  frame_dummy
  0.00      0.93     0.00       51     0.00     0.14  solution_check
  0.00      0.93     0.00       50     0.00    18.41  central2d_run
  0.00      0.93     0.00        1     0.00     0.00  central2d_free
  0.00      0.93     0.00        1     0.00     0.00  central2d_init
  0.00      0.93     0.00        1     0.00     0.14  lua_init_sim
  0.00      0.93     0.00        1     0.00     0.00  viz_open

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
     else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
     function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
     the function in the gprof listing. If the index is
     in parenthesis it shows where it would appear in
     the gprof listing if it were to be printed.
```

Figure 4: Output from profiling the serial code using gprof.

6 Compilation Details

Although we used different environments (Mac, Ubuntu, Arch) for local testing, profiling and debugging, the performance analysis was been done on Graphite. In this section, we discuss the compilation configurations we used.

We used the GCC 5.5.0 compiler provided by Graphite. The four flags that we used for the compilation are the following:

- `-O3` — Performs more aggressive optimization than `-O2` used by default.
- `-ffast-math` — Enables the compiler to reorder operations that may not actually be associative.
- `-fopenmp` — Enables the OpenMP directive `#pragma omp` in our C code for shared memory parallel processing.
- `-march=native` — Optimizes the code for the specific machine architecture.

7 Conclusion

To optimize and tune the implementation of the finite volume solver, we have applied two different strategies: sub-domain partitioning that splits the grid data into sub-grids to ease computation and using OpenMP to parallelize the computations (each thread handling a sub-grid). During this process, we overcame two difficulties: maintaining sub-grid consistency and synchronization of dt among the sub-grids.

Our performance analyses suggest that our optimization has doubled the performance with just two processors. The experiment conducted with strong scaling aligns with Amdahl's Law (that inherently serial part of the code sets the upper bound of the speedup of parallel code), and the experiments conducted with weak scaling aligns with Gustafson's Law (that the scaled speedup is linear with the number of processors).

Besides the aforementioned approaches, there were two additional attempts that were not as successful. First, a hybrid OpenMP and MPI strategy would have allowed us to run larger global grids, yet this approach could not be implemented due to difficulties in rewriting our implementation that does not depend on the global grid and resolving a complicated coordination problem during visualization. Second, naive insertion of `#pragma omp parallel` directive had a significantly worse performance than the serial implementation.