# CS5220: Shallow Water Project

Cheng Perng Phoo, Robert Stephany, Junyoung Lim

November 3, 2020

## 1 Introduction

In this project, we have tried various ways to optimize and tune the implementation of a finite volume solver for 2D hyperbolic PDEs via a high-resolution finite difference scheme (TODO: cite a reference to Jiang and Tadmor). Our approaches are built upon two central themes: sub-domain partitioning and parallelization using OpenMP. Based on the performance analysis, the best tuning we have achieved has (TODO: one-two lines summary of the performance result).

## 2 Discussion: Compilation Details

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

We used the GCC x.x.x (TODO: @Cheng can you specify any details about the compiler version here if there is any?) 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.

## 3 Discussion: Sub-domain Partitioning

### 3.1 Overview

The first approach is to partition the grid data, $U_G$, into $N$ sub-grids, $U_i$, each with its own start and end indices for both $x$ (horizontal) and $y$ (vertical) direction. In this report and our implementation, we will often refer to $U_i$ as a local grid, and $U_G$ as the global grid. Assume that each thread will handle each $U_i$ in a parallel manner. The overview below will overlook any parallelization aspect of the logic since more details about parallelization through OpenMP will come in the next section.

In the underlying CFD method, each cell is updated using the cells that directly neighbor it. In particular, this means that each thread needs to access cells of $U_G$ which lie outside of $U_i$ to update $U_i$ (specifically the cells which form the boundary of $U_i$). However, because information propagates through the simulation at a finite rate, each thread only needs to have access to the cells that directly neighbor its local grid. To enable this, each $U_i$ is padded with a few layers of "Ghost" cells, which hold a few layers of cells from other threads' local grids. These ghost cells are periodically updated to ensure that each thread can accurately update its local grid. As in Dr. Bindel's original code, the global grid is also padded with a few layers of ghost cells, but those are used to implement periodic boundary conditions.

In our implementation, the global grid $U_G$ is used for two purposes: to implement boundary conditions, and to act as a repository through which the threads can communicate with one another. When the threads need to update the ghost cells in their local grid, they first push the outer few layers of their local grid to their corresponding locations in the global grid (these are the cells from $U_i$ that other threads need to access to update their local grids). Once this has finished, we apply periodic boundary conditions to $U_G$. This order of operation ensures that the ghost cells of $U_G$ can be easily pulled into those of each $U_i$. Then we can safely apply boundary conditions to each $U_i$ by copying over the relevant cells from $U_G$ into the ghost cells of $U_i$. ghost cells from $U_G$ to $U_i$.

Thus, in our approach, the global grid exists, but only a tiny fraction of it is updated at each time step. We decided to keep the global grid for this purpose, and because it allows us to easily write the state of the system at the end of each time frame for visualization purposes. This isn't the most memory-efficient approach, but it doesn't require much communication between the processors and was straight forward to implement in a shared memory environment.

Once boundary conditions are applied, we compute the maximum wave speed in the $x$ and $y$ direction and determine $dt$. Unlike the original work, in which $dt$ is computed directly using $U_G$, we need to compute the maximum $x$ and $y$ velocities from each local grid and determine $dt_i$ for each $U_i$. Once all the computations are done, we determine $dt$ to be $\min\{dt_i\}_{i=1}^{N}$.

The rest of the main loop logic is the same as the original work, except that we need to make sure that copy the boundary of each local grid to the corresponding location in the global grid. Then proceed to the next iteration of the loop only after all the boundaries are copied over.

## 3.2 Implementation Details

Only some of the noteworthy code snippets, which will help understanding the implementation details, are discussed in this section. The interested reader is encouraged to see our full implementation on our Github repository, which includes myriad comments for clear exposition.

In order to apply the idea of sub-domain partitioning, we need to determine $N$, the number of partitions to create. To generalize our logic, we dynamically determine the number of partitions at runtime by detecting $p$, the number of threads running, and let $N = p$.

```
// ldriver_parallel.c
// First get the number of threads from the environment
// If the number of threads is null, then set the number of threads to 1
char* s = getenv("OMP_NUM_THREADS");

int num_threads = 0;
if (s != NULL) {
    num_threads = atoi(s);
}

if (num_threads == 0) {
    num_threads = 1;
}

printf("Number of threads: %d\n", num_threads);
const int n_rows = num_threads;
const int n_cols = 1;
```

n_row and n_col, are used to calculate two variables, xlow_local and ylow_local. xlow_local specifies which row of $U_G$ the bottom row of $U_i$ corresponds to. ylow_local specifies which column of $U_G$ the left column of $U_i$ corresponds to. These variables are used to compute the location (nx_local and ny_local) in $U_G$ that corresponds to each cell of $U_i$. Note that n_row $= N$ and n_col $= 1$, because we are exploiting the fact that matrices follow row major ordering in our code. This approach will, therefore, improve cache usage when threads read from and write to $U_g$. We initialize each $U_i$ as shown below.

```
// ldriver_parallel.c
float* U_local = sim_local->U;
float* U = sim -> U;
for(int k = 0; k < 3; ++k) {  // 3 = nfield
  for(int iy = 0; iy < ny_local; ++iy) {
    for(int ix = 0; ix < nx_local; ++ix) {
      U_local[central2d_offset(sim_local, k, ix, iy)]
        = U[central2offset(sim, k, xlow_local + ix, ylow_local + iy)];
    }
  }
}
```

stepper_parallel.c contains the main logic described in the overview. In addition to the provided functions, we introduce three new functions, each with the following functionalities:

- central2d_local_BC — Used to fetch $U_i$'s ghost cells from $U_g$.
- central2d_local_to_global — Copies the boundary of $U_i$ back to $U_G$.
- central2d_U_to_global_U — Copies the canonical cells of $U_i$ back to corresponding locations in $U_G$.

## 4 Discussion: OpenMP Parallelization

The aforementioned sub-domain partitioning involves handling $N$ disjoint partitions, each of which can be handled in parallel. #pragma omp directives are used to ensure proper setup of parallel processing of each thread.

Once we figure out $p$, the number of threads, we can initialize each $U_i$ and its corresponding location in the global grid $U_G$ from the thread number assuming row-wise partitioning. It's necessary to set the barrier before running a simulation so that all threads finish up their $U_i$ initialization.

```
// ldriver_parallel.c
#pragma omp parallel num_threads(n_rows*n_cols)
{
    // use omp_get_thread_num() to compute the sub grid location
    // call central2d_init(...)
    // initialize U_i, as shown in Section 3

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

Once all threads are ready with their initialization phase, run the simulation for each frame. By the time it needs to compute $dt$ from the maximum $x$ and $y$ velocities as described above, we can parallelize the operation by letting the first-arriving thread to create a shared buffer (with a single directive). Then each thread $T_i$ arrives and writes the minimum between $dt_i$ and the value already stored in the shared buffer. Wait until all threads are completed, and move on.

```
// 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
    ...
```

Before exiting the main loop, we copy the boundary of $U_i$ back to $U_G$. This needs to be done because at the start of the next time step, we're going to apply periodic boundary conditions, and that will only give the result we want if each thread has written its boundary cells into $U_G$.

```
// 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
    ...
```

The barrier directive is included so that threads do not start a new time step until every thread has written its boundary data to $U_G$. Once the main while loop in `xrun` completes, each thread writes its $U_i$ into $U_G$. We included another `barrier` directive after this step so that `xrun` does not return until every thread has finished writing to $U_G$. This is done so that, at the end of a time frame, $U_G$ stores the global state of the simulation. We can then use $U_G$ to run solution checks and print out to the visualizer. A consequence of this is that every thread returns from `xrun` at the same time. Therefore, we can use the time measurement from any one of the cells to evaluate the elapsed time. The solution check and visualizer steps can be done independently, and are, therefore, included in a sections directive. One section checks the correctness of our solution, and the other writes $U_G$ to the visualizer filer.

```
// ldriver_parallel.c
#pragma omp parallel num_threads(n_rows*n_cols)
{
    ...
    #pragma omp barrier

    for (int i = 0; i < frames; ++i) {
        // measure time elapsed from central2d_run(...)

        #pragma omp sections {
            #pragma omp section {
              // run out diagnostic on U_i by running solution_check(...)
            }
            #pragma omp section {
              //  write a frame of U to memory by running viz_frame(...);
            }
        }
    }
    ...
}
```

# 5 Performance Analysis

Keep the same notation as above, where $p$ is the number of threads, and assume each thread is uniquely associated to a processor.

## 5.1 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 compute 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 increased the number of threads. 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.

## 5.2 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 will be hard to achieve if we want to maintain a square grid data. For instance, suppose we conduct 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 is 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 aligned 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 $500 \times 500$. 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 yet again consistent with Gustafson's Law.

# 6 Attempts That Didn't Work

## 6.1 Using OpenMP with MPI

Originally, we planned to use a hybrid OpenMP and MPI approach. In this approach, we would have used MPI to partition the domain (which is a, in our opinion, a problem that is better suited for a distributed memory setup) and OpenMP to further parallelize the computations on each piece of the partition. This approach would allow us to run larger global grids. One downside to the implementation described above is 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 forging 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 relied on a global grid. As described in previous sections, our implementation used the global grid as a repository through which the threads can communicate, to apply global periodic boundary conditions, to run the solution check, and to simply 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.

## 6.2 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 is significantly worse than the serial code.

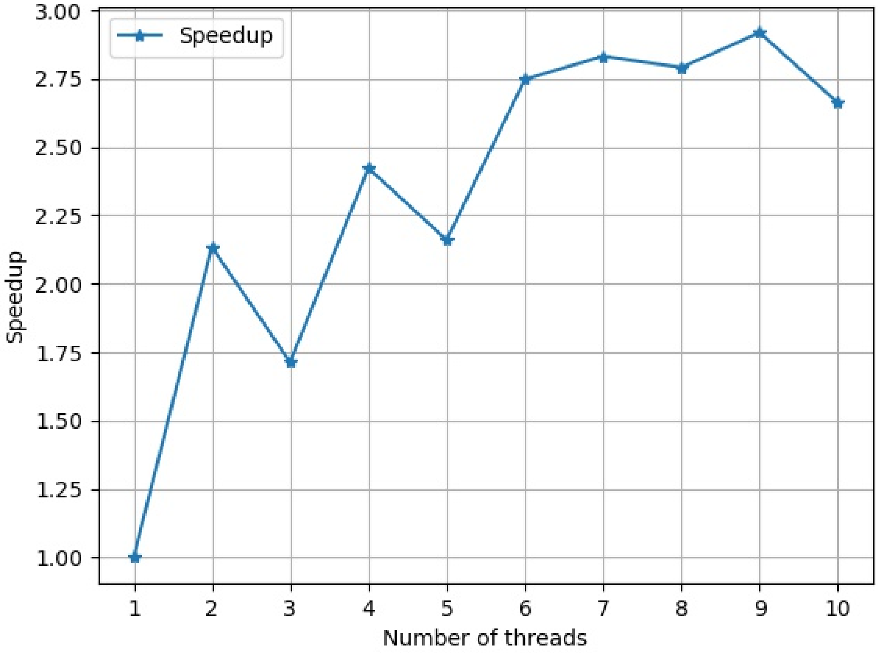

Figure 1: Speedup for Strong Scaling Study

TODO: add Output from Gprof diagram.

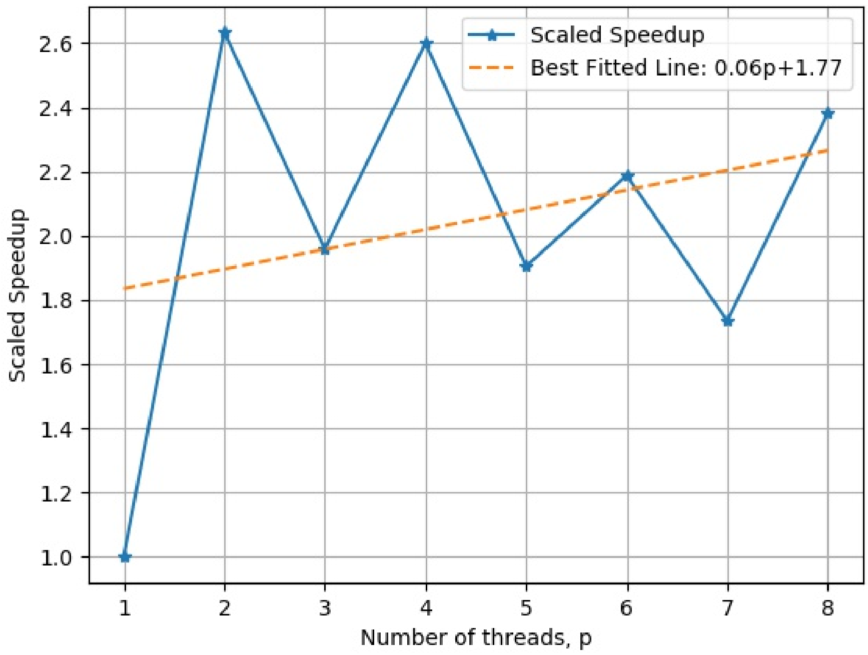

Figure 2: Speedup for Weak Scaling (Square Grid)

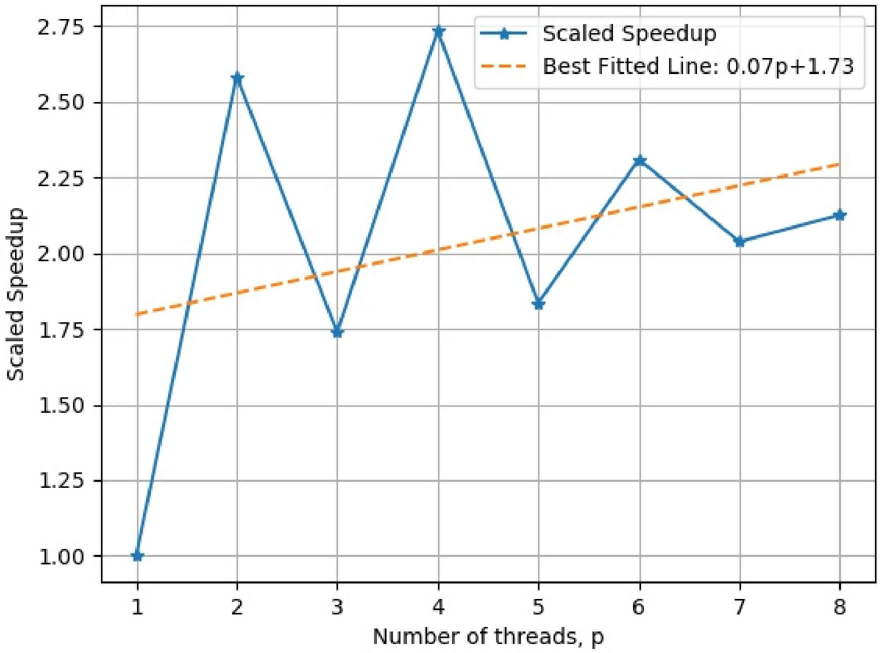

Figure 3: Speedup for Weak Scaling (Constant Workload for Processor)

# 7 Conclusion

TODO: write a conclusion.