CS5220: Shallow Water Project

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

November 3, 2020

1 Introduction

In this project, we had 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 by Jiang and Tadmor. Our approaches were built upon two central themes: sub-domain partitioning and parallelization using OpenMP. Based on the performance analysis, the best tuning we had achieved a twofold speedup with only two processors (See 4.1). Our implementation can be found at: `https://github.com/cpphoo/shallow-water`.

2 Sub-domain Partitioning

To start, assume the grid data was partitioned into N sub-grids $\{U_i\}_{i=1}^N$. For this section, we focused on the high level implementation logic. We assumed that each sub-grids U_i would be handled by a single thread in a parallel manner and each thread had a deep copy of the sub-grid). We defer details on OpenMP parallelization to the next section (section 3).

2.1 Maintaining Sub-grids Consistency Via a Global Grid U_G

Given that the underlying CFD method updated each cell based on its adjacent cell, each thread needed to access cells that were outside of its respective grid U_i for updates (i.e. the boundary of U_i). Fortunately, each thread did not need access to all cells in its neighboring grids but rather a few cells that were adjacent to the boundary of its local sub-grid for updates. To ensure that each thread had access to the information needed to update the boundary cells, we padded each U_i with a few layers of "Ghost" cells that held information of cells needed for updates from other threads' sub-grids. We ensured the consistency of "Ghost" cells between different sub-grids by maintaining a global grid U_G - a common repository through which the threads could communicate with one another.

When the threads needed to update the "Ghost" cells in their local grid, they first pushed the outer few layers of their local grid to their corresponding locations in the global grid (these were the cells from U_i that other threads needed to access to update their local grids). Once this was finished, we applied periodic boundary conditions to U_G and copied the values back to the individual sub-grids. The order of operation ensured that the "Ghost" cells in each local sub-grids could be consistent and the boundary conditions of the original PDE were respected.

2.2 Determining dt for Finite Difference

Once the boundary conditions were applied and the changes were propagated back to the individual sub-grids, we computed the maximum wave speed in the x and y direction and determine dt. Different from the original work in which dt was computed directly based on the whole grid, we determined dt by determining dt_i for each U_i (based on the $x-$, $y-$ velocity) and set dt to be $\min\{dt_i\}_{i=1}^N$.

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 allowed us to exploit the fact that the grid data (or matrices) followed row major ordering which improved cache usage when the threads read from and wrote to U_g

3 OpenMP Parallelization

Sub-domain partitioning involved handling N disjoint partitions that, in principle, could be processed in parallel. To enable parallelism, we employ a shared memory model, OpenMP. In this section, we elaborated a few key implementations that enabled parallelization of sub-domain partitioning.

3.1 Initialization of Each Thread

Once we figured out the number of threads p, we could initialize each U_i by making a deep copy of partition of the original grid data. After this, we would use the original grid data as our global grid U_G. Before the simulation was started, it was crucial to ensure that all threads finish up their U_i initialization. This was 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 needed to compute dt for the finite difference. To speedup the computation, we attempted to parallelize computation as described in section 2 which required 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 established 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:

```
// 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 were updated, we copied 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 was completely updated, we again establish a barrier as shown as follows:

```
// 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 was performed on the global grid U_G.

4 Performance Analysis

In this section, we assumed p was the number of threads and each thread was 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.

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:

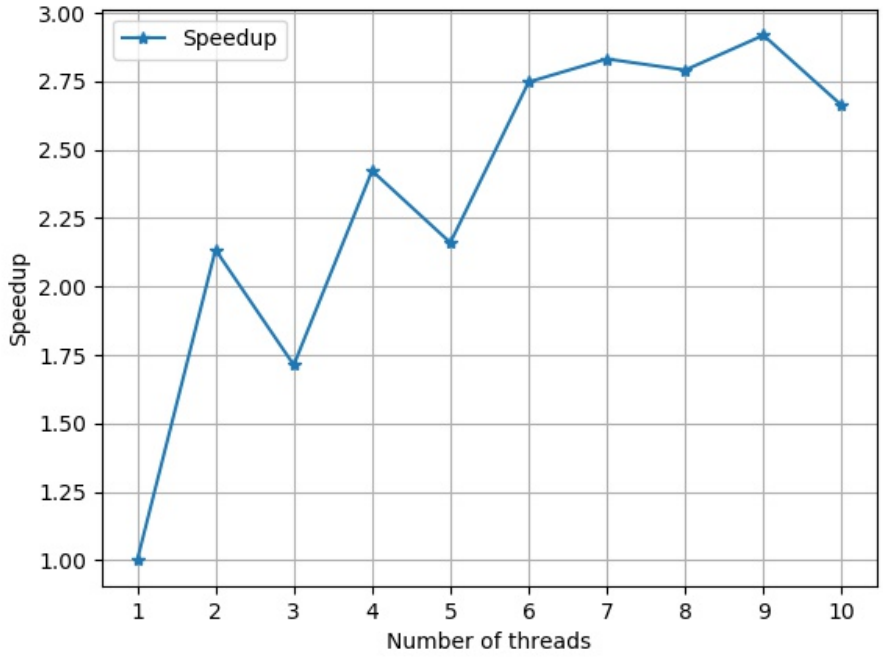

Figure 1: Speedup for Strong Scaling Study

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 Failed Attempts

In this section, we elaborated on our failed attempts.

5.1 Using OpenMP with MPI

Originally, we planned to use 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. 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

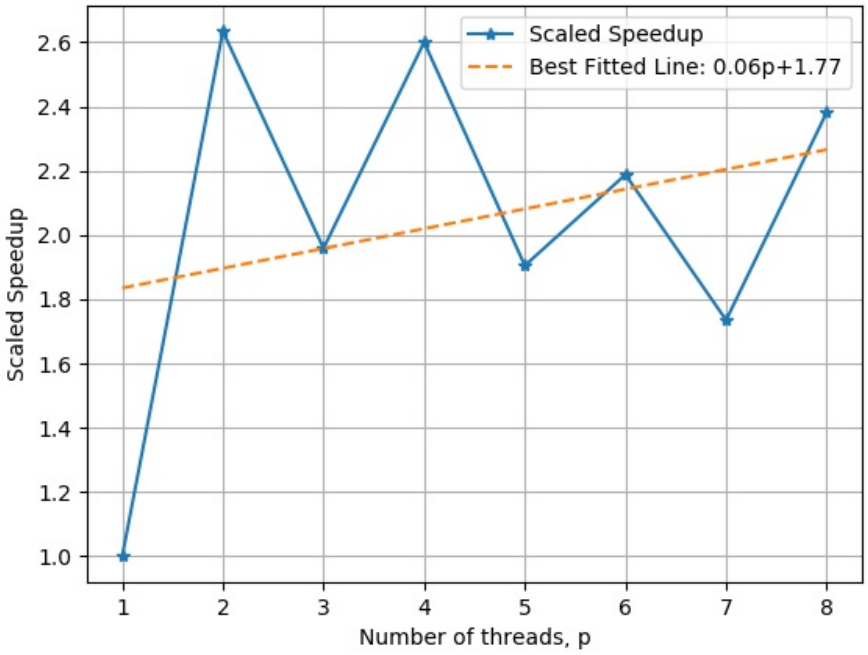

Figure 2: Speedup for Weak Scaling (Square Grid)

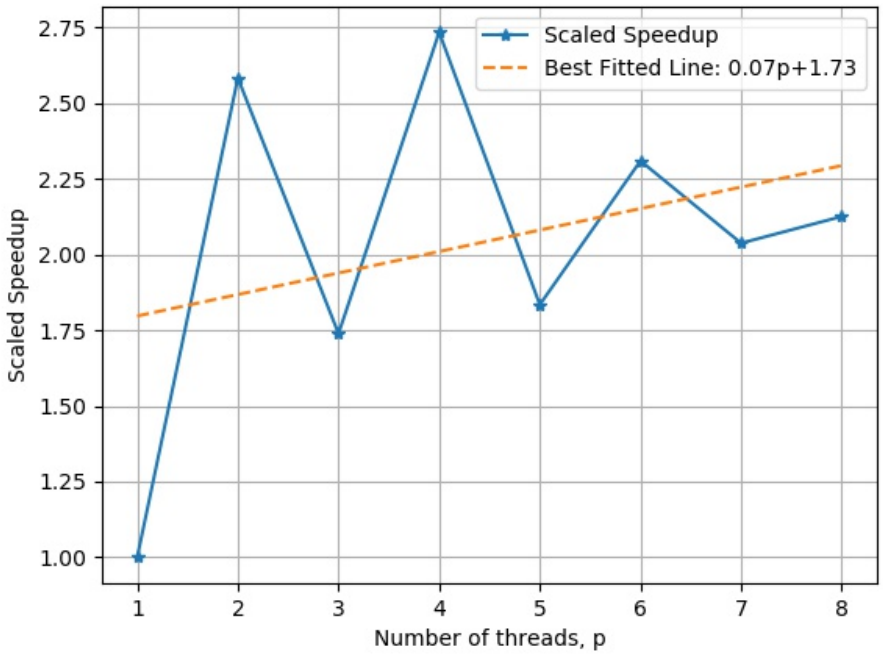

Figure 3: Speedup for Weak Scaling (Constant Workload for Processor)

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 could communicate, to apply global periodic boundary conditions, to run the solution check, and to 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.

5.2 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 were significantly slower than the serial code.

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; through sub-domain partitioning, our implementation uses disjoint sub-grids to carry on the computation, each of which is handled parallelly via OpenMP directives. During this process, there were two technical difficulties to overcome: maintaining sub-grid consistency and synchronization of dt among the sub-grids.

Our performance analysis results suggest that the best 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

```
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.

could not be implemented due to an additional overhead of 1) rewriting our implementation that does not depend on the global grid and 2) resolving a complicated coordination problem during visualization. Second, naive insertion of `#pragma omp parallel` directive had a significantly worse performance than the serial implementation.