CS 5220 Project 2 – Team 7 Mid Report

Taejoon Song, Michael Whittaker, Wensi Wu
{ts693, mjw397, ww382}@cornell.edu

October 19, 2015

1 Introduction

Structured grid computation, the fifth of the famous thirteen dwarfs, is a ubiquitous computational pattern in high performance computing. Structured grid computations show up in a variety of contexts including cellular automata and fluid dynamics simulations. In this paper, we present the development of a set of optimized shallow water simulators.

In §2, we discuss a simple verification system we developed to ensure the correctness of our simulators. In §3, we explain how we timed and evaluated our simulators. In §4, we present how we modified the build system to increase developer productivity. In §5 and §6, we describe how we vectorized and parallelized our simulators. In §7 and §8, we evaluate our simulators and discuss future work.

2 Verification

Shallow water simulation involves a lot of numerical computation, and optimizing the simulation involves modifying and rearranging the code that performs these computations in complex and intricate ways. Given the subtlety and intricacy of the code, it is very easy to accidentally introduce bugs into optimized code. The released code includes a `solution_check` function which verifies certain invariants of the simulation. For example, it checks that the volume of water is conserved and that there are no negative water heights. This invariant checking catches *some* but not *all* bugs.

In order to catch all bugs and guarantee the correctness of our code, we developed our own simple verification system. In order to verify the correctness of a simulator, we check its output against the reference simulator provided in the release code. More concretely, assume we want to check the correctness of an optimized `OptSim` simulator. We instantiate both `OptSim` and the reference simulator `RefSim`.

```
typedef Central2D   <Shallow2D,    MinMod   <Shallow2D   ::real>> RefSim;
typedef Central2DOpt<Shallow2DOpt, MinModOpt<Shallow2DOpt::real>> OptSim;
RefSim ref_sim;
OptSim opt_sim;
```

Then, we simulate `opt_sim` and `ref_sim` in lockstep and use a function `validate` to check that the two simulators have identical grids up to rounding error.

```
for (int i = 0; i < frames; ++i) {
    opt_sim.run(ftime);
    ref_sim.run(ftime);
    validate(ref_sim, sim);
}
```

To ensure our verification system is itself correct, we developed an intentionally buggy `BuggySim` simulator and a copy of the reference simulator `CopySim`. Our system correctly reports that `BuggySim` is incorrect while `CopySim` is correct.

3 Timing

3.1 Profiling

In order to find the bottlenecks of our simulators, we first profiled our code using Intel's VTune Amplifier, as shown in Figure 1.

```
amplxe-cl -collect advanced-hotspots ./shallow
amplxe-cl -report hotspots -source-object function=<NAME>

amplxe-cl -report hotspots -r r001ah/ > all
amplxe-cl -report hotspots -source-object function="Central2D
<Shallow2D, MinMod<float>>::compute_step" > compute_step
```

Figure 1: VTune Amplifier Command

3.2 Initial Profile Result

From Figure 2, we concluded that `limited_derivs`, `compute_step`, and `compute_fg_speeds` take the longest time (see `profile.sh` for more information). Among these results, the `limited_derivs` function was the worst bottleneck.

```
Function                       Module        CPU Time
-------------------------------------------  --------
limited_derivs                 shallow         1.378s
compute_step                   shallow         0.640s
compute_fg_speeds              shallow         0.219s
_IO_fwrite                     libc-2.12.so    0.019s
_IO_file_xsputn                libc-2.12.so    0.015s
[Outside any known module]     [Unknown]       0.014s
run                            shallow         0.006s
write_frame                    shallow         0.006s
solution_check                 shallow         0.004s
offset                         shallow         0.002s
do_lookup_x                    ld-2.12.so      0.001s
operator[]                     shallow         0.001s
```

Figure 2: Initial Profile Result

3.3 Initial Timing Result

To visualize our simulator performance, we wrote a simple script to generate timing plots (see `plotter.sh` code). The x-axis of the plots indicates the number of cells per side of a simulator grid, nx. We swept nx from 100 to 500. The y-axis shows the number of simulated cells per second. To compute the number of simulated cells per second, we scaled the total execution proportionally to nx^{-3}. Our script also generates plots that simply show the total execution time vs. nx.

Initial timing result can be found at Figure 3 and Figure 4 .

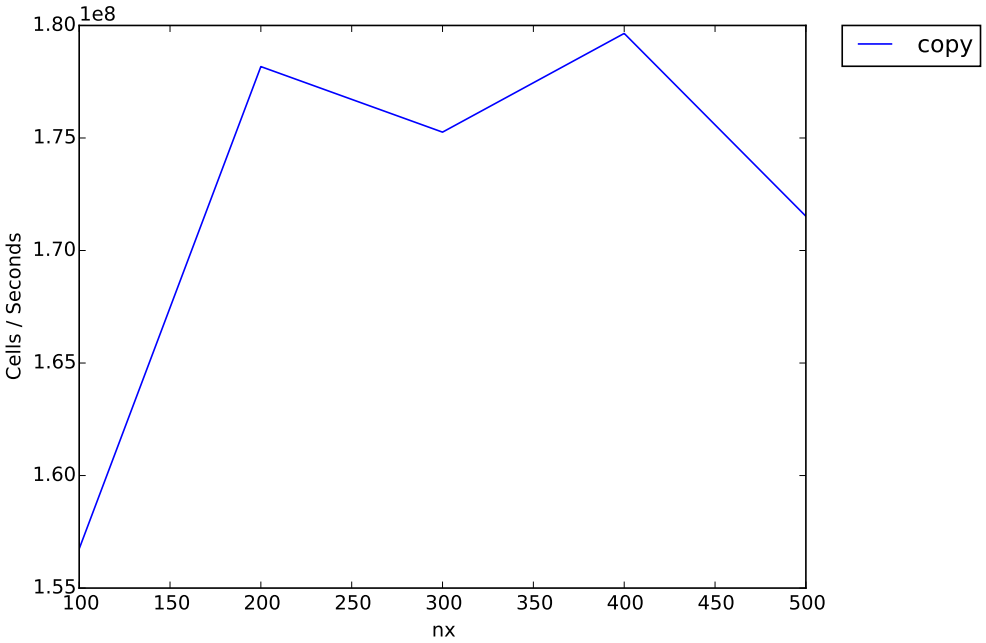

Figure 3: Initial Timing Result (*cells/seconds* vs. *nx*)

4 Improved Build System

The development of an efficient shallow water simulator is a complicated task that involves a lot of rapid prototyping, and the efficiency with which we can develop simulators is limited in part by the overhead of creating, modifying, and running simulators. In order to reduce this overhead, we improved and streamlined the build system.

The original `water` build system did not accommodate an easy way to run multiple versions of a shallow water simulator. Instead, we were forced to modify the type definitions at the top of `driver.cc` before compilation which is slow and error-prone. To alleviate this burden, we modified `driver.cc` to instantiate one of a set of simulators based on a macro which is provided at the command line during compilation. For example, assume we have two simulators `SimA` and `SimB`. To build `driver.cc` to use `SimA`, we run a command similar to `icpc -DVERSION_a -o shallow_a driver.cc`. Similarly, to build `driver.cc` to use `SimB`, we run something similar to `icpc -DVERSION_b -o shallow_b driver.cc`. Moreover, we abstract these commands using the Makefile so that we only have to run `make shallow_a` or `make shallow_b`.

This improved build system makes it much easier to develop and evaluate many versions of simulators simultaneously.

5 Vectorization

Vectorization is a critical part of optimization; typically, vectorized code can run many times faster than its un-vectorized counterpart. The reference shallow simulator implemented in `central2d.h` is almost entirely un-vectorized. For example, consider the loop in Figure 5

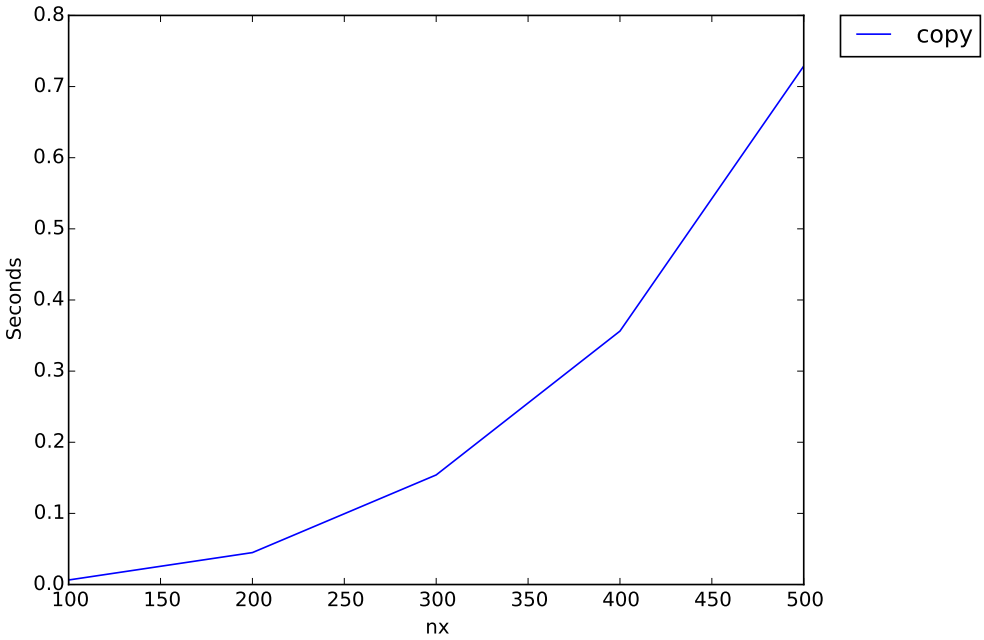

Figure 4: Initial Timing Result (*seconds* vs. *nx*)

which was taken from `central2d.h`.

```
// Copy from v storage back to main grid
for (int j = nghost; j < ny+nghost; ++j){
    for (int i = nghost; i < nx+nghost; ++i){
        u(i,j) = v(i-io,j-io);
    }
}
```

Figure 5: A loop from `central2d.h`.

The loop copies values from the `v` vector into the `u` vector; this is clearly a vectorizable operation. However, consider the vectorization report in Figure 6.

There are two important things to notice. First, the compiler assumes there are multiple flow and anti dependencies in the loop which prevent it from being vectorized. This is caused in part by the fact that the reference simulator represents solution vectors as `std::vector`s of `std::array`s which are not amenable to vectorization. Second, the vectorization report mentions the dependencies involve `_M_elems`: a rather cryptic name which is, from what we can gather, the name of a function internal to the implementation of `std::vector` or `std::array`. This is caused entirely by the use of vectors and arrays.

We solved both these problems in our vectorized simulator implemented by `central2d_vec.h` and `shallow2d_vec.h`. Our vectorized simulator replaces the `std::vector` of `std::array`s with plain dynamically allocated arrays that logically represent a struct of arrays, as shown in Figure 7. This solves the first problem because operations on the flat arrays are much easier for the compiler to vectorize. It solves the second problem by removing the use of

```
LOOP BEGIN at central2d.h(362,5)
   #15344: loop was not vectorized: vector dep prevents vectorization
   #15346: vector dep: assumed FLOW dep between _M_elems line 364 and _M_elems line 364
   #15346: vector dep: assumed ANTI dep between _M_elems line 364 and _M_elems line 364

   LOOP BEGIN at central2d.h(363,9)
      #15344: loop was not vectorized: vector dep prevents vectorization
      #15346: vector dep: assumed FLOW dep between _M_elems line 364 and _M_elems line 364
      #15346: vector dep: assumed ANTI dep between _M_elems line 364 and _M_elems line 364
   LOOP END
LOOP END
```

Figure 6: The vectorization report of the loop in Figure 5.

`std::vector` and `std::array` entirely.

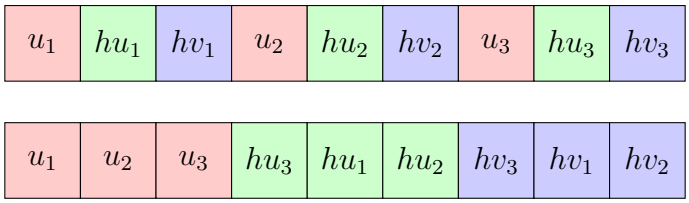

Figure 7: (top) The memory layout of an array of structs as used in the reference simulator. (bottom) The memory layout of a struct of arrays as used in the vectorized simulator.

In addition to using flat arrays, we are also in the process of modifying loop orderings, using `#pragma ivdep` annotations, and generally rearranging code in order to increase vectorization. We have already vectorized a fair number of the loops, but some remain to be vectorized.

6 Parallelization

In this assignment, we used OpenMP to parallelize our code in hopes of improving the computational performance. OpenMP stands for Open Multi-threads Programming; it is an industry standard API of C/C++ for shared memory parallel programming. The way OpenMP works is first decomposing the work into smaller chunks, and then assigning the tasks to different threads such that multiple threads can share work in parallel. When the work is done, the threads will synchronize implicitly by reading and writing shared variables.

Although OpenMP is a very powerful tool to use to increase speedup, we found that using OpenMP will actually worsen the performance if it is not implemented appropriately. Two of the possible ways that contribute to the cause are load imbalance overhead and parallel overhead. Load imbalance overhead is when the threads are performing unequal amount of work in the shared region. The faster thread will need to wait for all the other threads to finish the work before they can synchronize the information; when the threads are not doing any work/idling, it accumulates synchronization overhead. Since our code was not vectorized yet, using OpenMP would not help us to achieve our goal.

Parallel overhead is the accumulated time that takes to start threads, distribute tasks to threads, and etc. Using Vtune to analyze the runtime of our codes, it appears that the time

for the processor to compute information in most of our for-loops was in the range of microseconds. As a result, using `#pragma omp parallel for` would not be beneficial because it takes much more time to distribute work to threads than to compute the code in serial.

7 Evaluation

We evaluate our different simulators using our own timing plots. Timing result with vectorization can be found at Figure 8 and Figure 9. In Figure 8, we plot cells/seconds vs. nx. We used the cube of cells. As discussed in Piazza (Referring to Prof. Bindel's answer), there are two important factors: the time between frames, and the time step. Total time is proportional to nx^3. In Figure 9, we can clearly see timing is proportional to nx^3. Note that "copy" is our initial copy result from the very first scratch, and "vec" indicates our vectorization result.

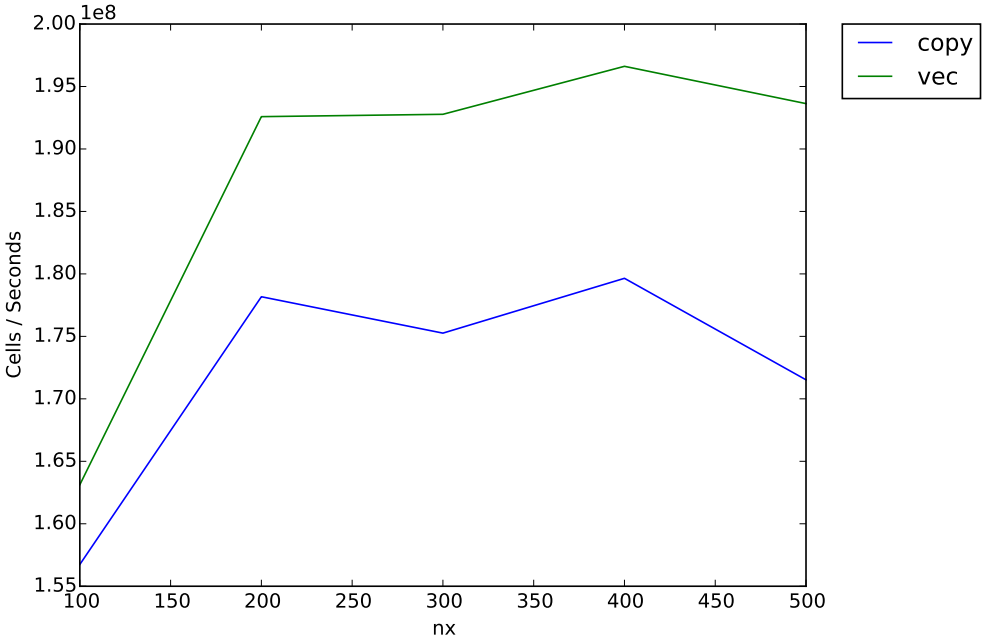

Figure 8: Vector Timing Result (*cells/seconds* vs. nx)

7.1 What Worked

We swept nx from 100 to 500 to see how the performance works given nx. First, like we have mentioned above, we could see timing is proportional to nx^3. In terms of performance improvement, we could achieve quite good improvement by just doing vectorization. If you see Figure 8, vectorization did more jobs per second.

7.2 What did not work

We tried OpenMP to make it parallel. Like we said at earlier section, OpenMP is a powerful tool to use to increase performance, but we noticed that it actually decreases the performance.

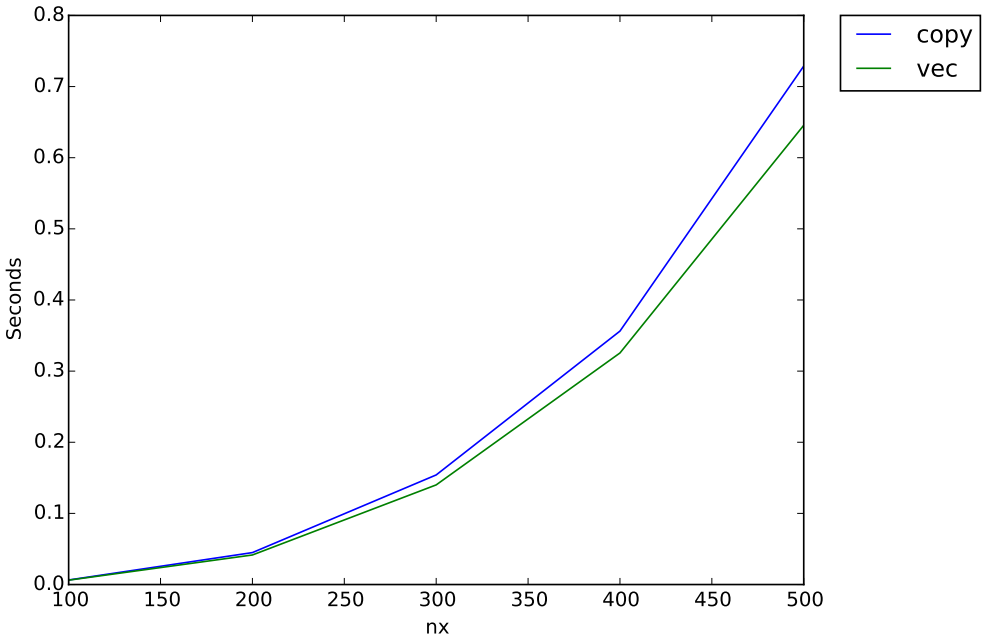

Figure 9: Vector Timing Result (*seconds* vs. *nx*)

Since it exceeded wall clock time, we could not even get the result yet. Hopefully, after the feedback, we can work on parallel part again.

8 Future Work

There are many optimizations that we plan on implementing before the final submission.

- **Vectorization.** We will complete the vectorization of our vectorized simulator.
- **Work Minimization.** The reference simulator performs some redundant computation. For example, the function `compute_fg_speeds` computes the flux for every ghost cell and its corresponding canonical cell. The number of ghost cells is small, but there may still be some benefits to removing this redundancy.
- **Compile Time Sizing.** The compiler is unable to vectorize certain loops because the bounds on the loops are unknown at compile time. Many of the loop bounds involve `nx`, `ny`, `nx_all`, or `ny_all` which are provided to the simulator at compile time. We plan on converting our simulators to take in these values as compile time template parameters. This will allow the compiler to perform more aggressive optimizations.
- **Blocking.** We plan on performing some sort of blocking to improve the memory locality of the simulator.
- **Ghost Celling and Domain Decomposing.** The reference simulator doesn't have much opportunity for naive parallelization. However, after we block our computation and introduce ghost cells between blocks, we will be able to parallelize our code at a much coarser level.