CS 5220 Project 2 – Team 7 Final Report

Taejoon Song, Michael Whittaker, Wensi Wu
{ts693, mjw297, ww382}@cornell.edu

October 29, 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, §7, and §6, we describe how we vectorized, parallelized, and domain decomposed our simulators. In §8, we evaluate our simulators.

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 .

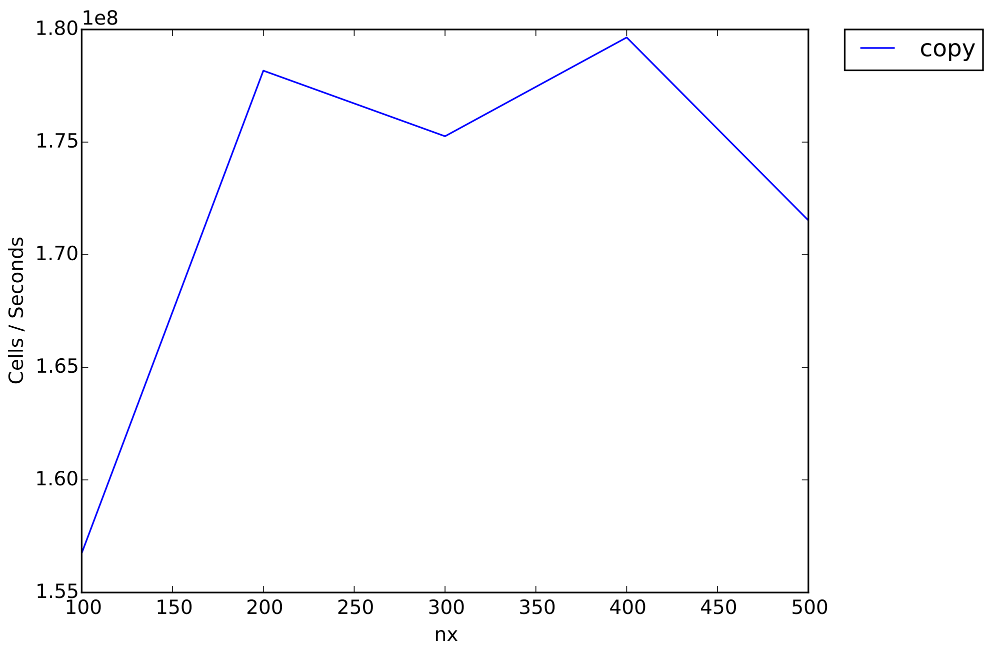

Figure 3: Initial Timing Result (*cells*/*seconds* vs. *nx*)

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`.

We also modified our build system to make it easy to enable and disable the verification presented in §2. We enable verification when debugging and developing our applications. However, since verification involves executing two simulators and repeatedly checking for grid equivalence, it is expensive. Thus, when we time and evaluate our simulators, we disable verification.

Our build system also involves commands for profiling simulators, running multiple simulators at once, running a simulator with various values of *nx*, etc.

Overall, the improved build system makes it much easier to develop and evaluate many

Taejoon Song, Michael Whittaker, Wensi Wu
{ts693, mjw297, ww382}@cornell.edu

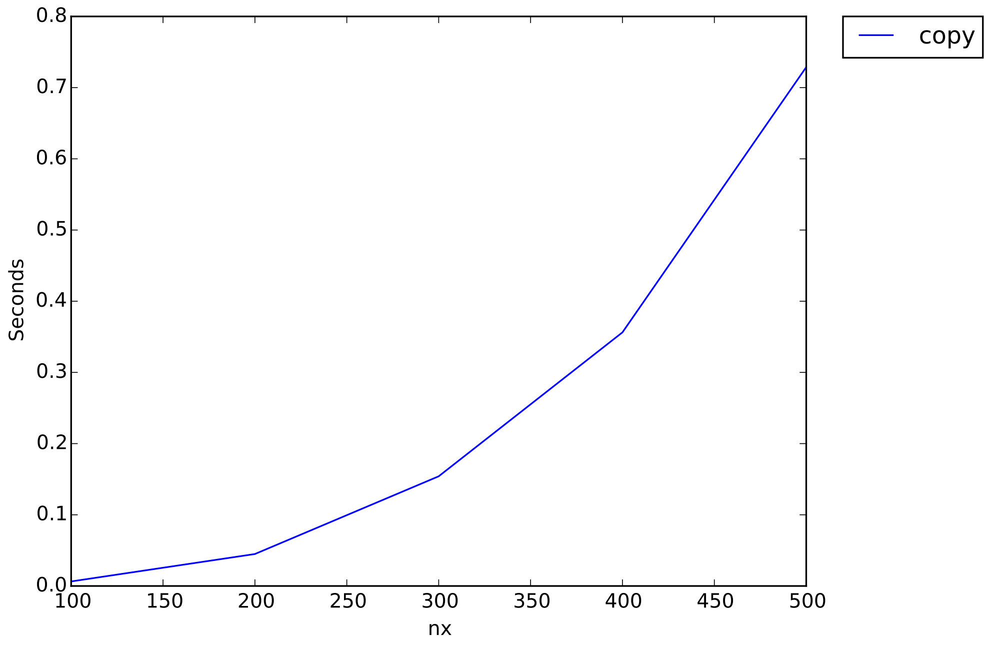

Figure 4: Initial Timing Result (*seconds* vs. *nx*)

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 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

```
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.

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

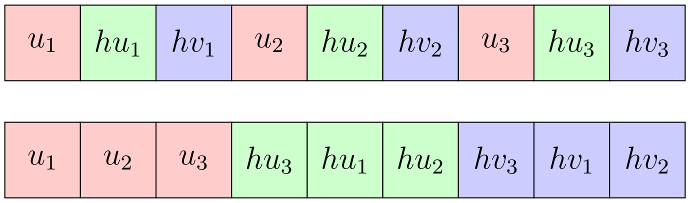
u_1 hu_1 hv_1 u_2 hu_2 hv_2 u_3 hu_3 hv_3
u_1 u_2 u_3 hu_3 hu_1 hu_2 hv_3 hv_1 hv_2

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 modified loop orderings, used `#pragma ivdep` annotations, and generally rearranged code in order to increase vectorization. Ultimately, we were able to vectorize almost all critical loops in our simulators.

6 Domain Decomposition

Our shallow water simulators represent a region of water as a two-dimensional grid of water heights. The simulation proceeds in several steps, and during each step, values from one grid are transformed into values of another grid. Symbolically, if one step of the simulation transforms values from a grid A to a grid B using a function f, then the following formula is used to populate B.

$$\forall i, j.\, B_{ij} = f(A_{ij})$$

A naive implementation of this pattern iterates over all values of i and j transforming values from A to B. This implementation is analogous to a naive implementation of matrix multiplication and suffers from the same problems. More specifically, the naive implementation has poor cache locality leading to poor performance on large grids.

To improve cache locality, we implemented blocking, or domain decomposition, in which we operate on grids one block at a time. Domain decomposition not only increases cache locality; it also allows for very easy parallelization. This is discussed in detail in §7.

The details of our implementation are not discussed here; they can be found in `central2d_block.h`.

7 Parallelization using OpenMP

In the initial stage of the project, we used Open Multi-threads Programming (OpenMP) to parallelize our code and improve its performance. OpenMP is an industry standard API of C/C++ for shared memory parallel programming. OpenMP first decomposes work into smaller chunks, and then assigns the chunks to different threads such that multiple threads can perform work in parallel. When the work is done, the threads synchronize implicitly.

Although OpenMP is a very powerful tool that can greatly improve the performance of code, we found that using OpenMP alone without any blocking or vectorization actually worsened performance. Two of the reasons were load imbalance overhead and parallel overhead. Load imbalance occurs when threads perform unequal amounts of work. Faster threads need to wait for the other slower threads to finish their work before they can synchronize.

Parallel overhead is the accumulated time that it takes to start threads, distribute tasks to threads, etc. Using Vtune to analyze the runtime of our code, 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` was not beneficial because the time to distribute work to threads was larger than the time to perform the loop serially.

However, after we performed domain decomposition, as described in §6, we were able to optimize the performance of our simulators by a factor of 6-10 using a `#pragma omp parallel for collapse(2)` clause as shown in Figure 8. The `run_block` function decomposes a simulator's grid into smaller sub-domains wrapped with a ring of ghost cells. The `#pragma omp parallel for collapse(2)` clause parallelizes the loop and allows multiple threads to perform the simulation at the granularity of blocks. This minimizes work imbalance and makes parallel overhead marginal.

```
#pragma omp parallel for collapse(2)
for (int by = 0; by < BY; ++by) {
    for (int bx = 0; bx < BX; ++bx) {
        run_block(io,dt,bx,by);
    }
}
```

Figure 8: A parallelized loop from `central2d_opt.h`.

8 Evaluation

We used two metrics to measure the performance of our simulators. First, we measured the number of cells our simulators can process in one second. Second, we measured the total execution time of our simulator. The cell throughput metric is desirable because it is not dependent on the size of the underlying grid being simulated. However, the cells throughput isn't easily interpreted. Similarly, total execution time is easy to interpret, yet it varies with the size of the underlying grid. In this section, we evaluate our simulators using these two metrics.

We developed three shallow water simulators.

1. `vec`. Our `vec` simulator is a vectorized version of the initial unoptimized compiler. It includes all the optimizations discussed in §5.
2. `block`. Our `block` simulator introduces blocking in addition to vectorization. It includes all the optimizations described in §5 and §6.
3. `opt`. Our `opt` simulator is a fully optimized simulator that includes vectorization, blocking, and parallelization. It includes all the optimizations described in §5, §6, and §7.

The cell throughput and total execution time of these three simulators and the release simulator for various values of nx are given in Figure 9 and Figure 10. Our `vec` simulator performs better than the release simulator for most values of nx, though surprisingly it performs worse for large grids. Our `block` simulator expectedly performs better than both the release and `vec` simulator especially for large values of nx. Our `opt` simulator performs significantly better than almost all other simulators.

Based on our qualitative and quantitative evaluations, we see that our `opt` simulator is able to spread computation very well over multiple threads. The `opt` simulator completes 4 times faster on average and up to 6 times faster than the reference simulator.

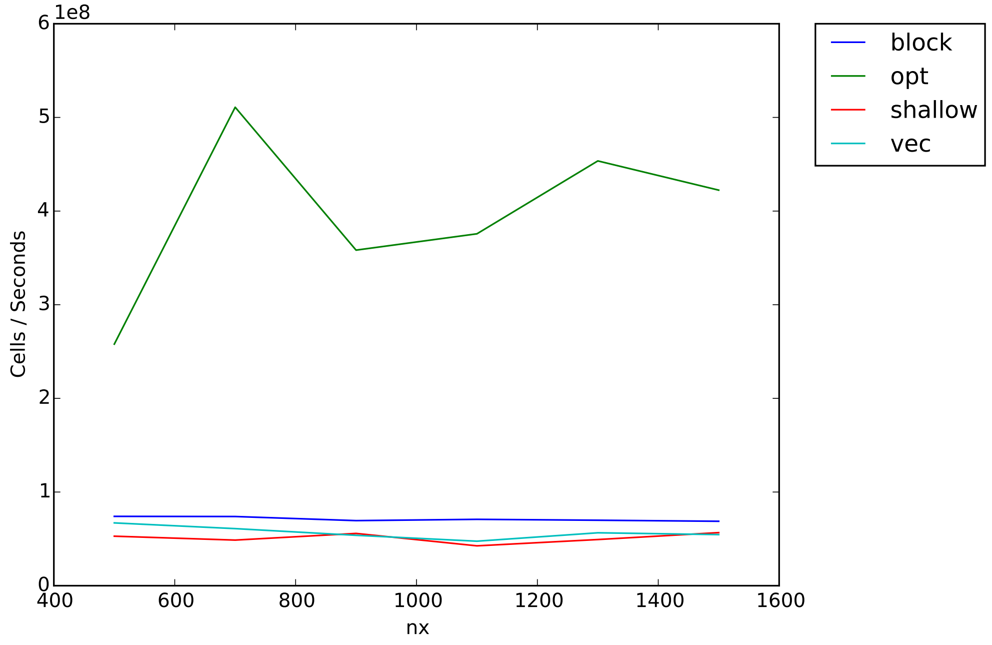

Figure 9: Final Timing Result (*cells/seconds* vs. *nx*)

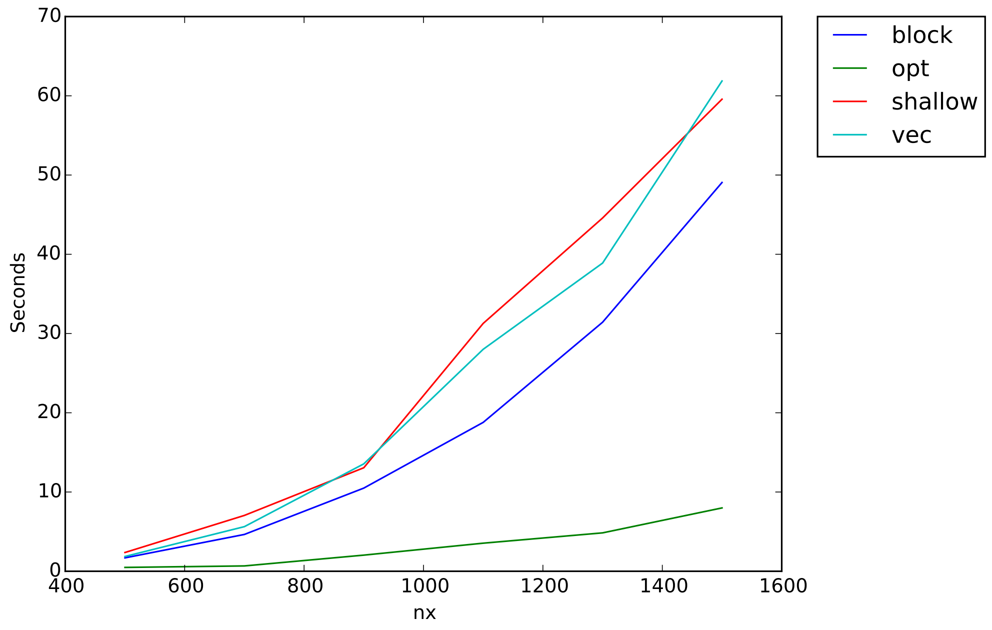

Figure 10: Final Timing Result (*seconds* vs. *nx*)