Project 2 - Shallow Wave Equation

Team 15 – Amiraj Dhawan, David Eckman, Xiangyu Zhang

1 Introduction

Our objective was to analyze a coded version of the shallow wave equation—a physical model of the dynamics of water over time. Our implementation assumed a square domain with periodic boundary conditions. We profiled the C++ serial version and created a parallelized version using OpenMP. We tested the performance of the parallelized version in weak- and strong-scaling experiments but did get around to profiling it. We did however come up with a number of ideas for how our parallel implementation could be further improved.

2 Profiling

We used amplxe to profile the serial implementation. The amplxe hotspots report showed that the following three functions took the most CPU time:

Module	CPU Time (s)
`Central2D<Shallow2D, MinMod<float>>::limited_derivs`	1.137
`Central2D<Shallow2D, MinMod<float>>::compute_step`	0.550
`Central2D<Shallow2D, MinMod<float>>::compute_fg_speeds`	0.197

The function `limited_derivs` calculates a limiter version of the derivatives of the fluxes and solution values at each cell, calling the function `limdiff` four times, which in turn calls the function `xmin` twice. The function `compute_step` updates the properties of each cell using the physics model and a staggered grid scheme. The function `compute_fg_speeds` calculates the maximum wave speeds in the x and y directions over the entire domain. All three functions require double `for` loops (in x and y) to cover all cells in the domain. This suggests that they are computationally intensive yet may also be easily parallelized across threads if the domain can be decomposed into subdomains.

We also studied how the CPU times of the three functions scale with respect to the modeling parameters: the numbers of cells along an edge of the domain and the numbers of time frames. Timing results are shown in Figures 1 and 2.

Figure 1 shows a cubic relationship between the number of cells per side of the domain (n) and the CPU times across all three functions. This result matches the observation that the work per time step increases by a factor of n^2 while the number of time steps increases by a factor of n, giving a total factor of n^3. Figure 2 shows a linear (perhaps sublinear) relationship between the number of frames (F) and the times.

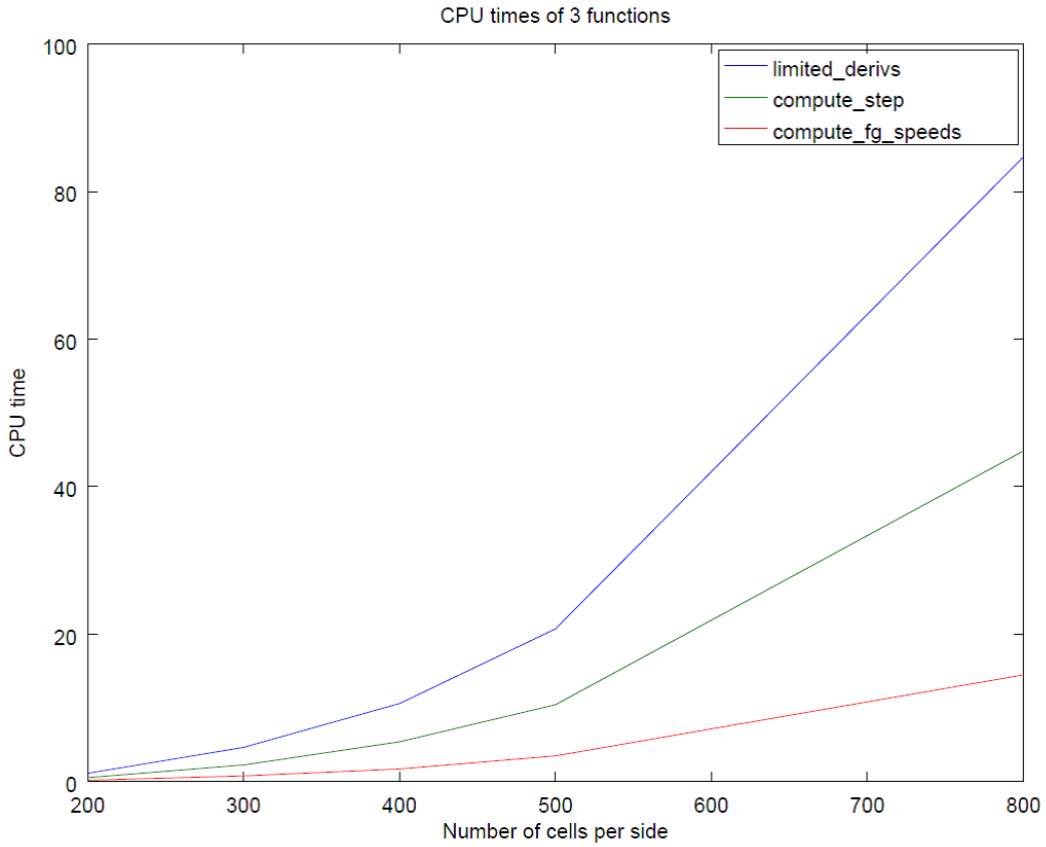

Figure 1: CPU time (seconds) vs number of cells per side.

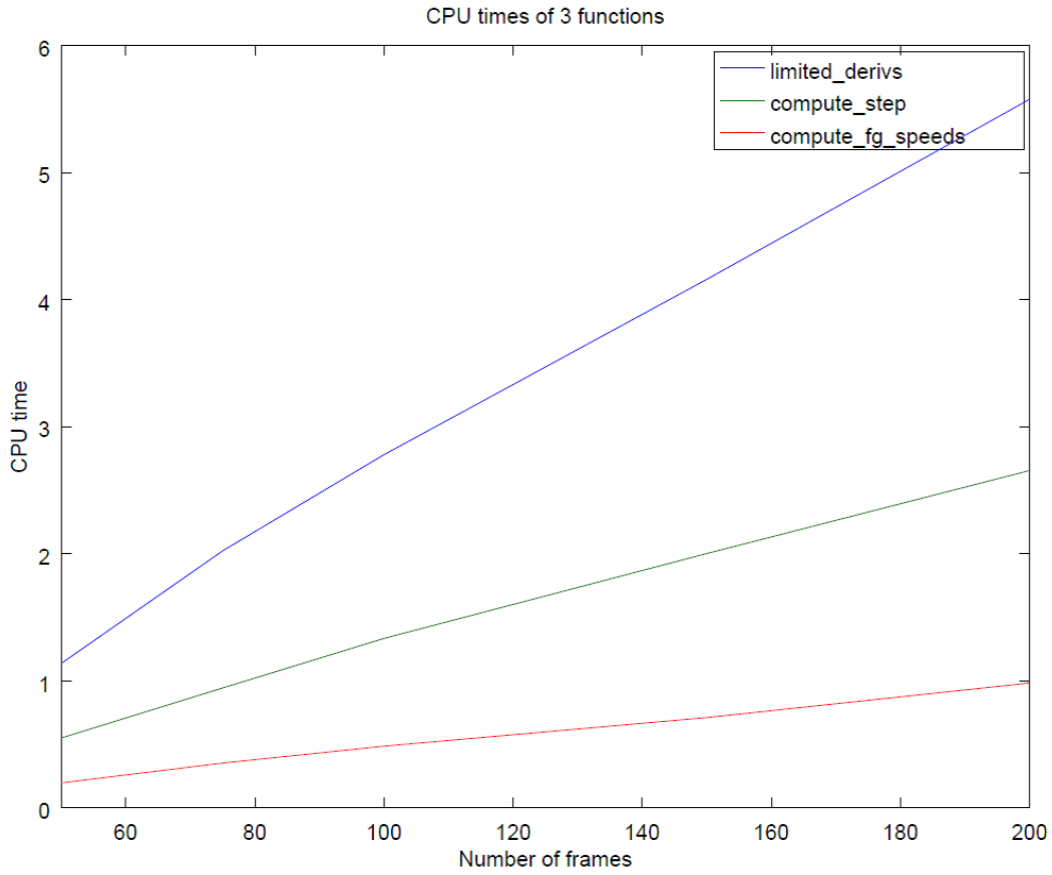

Figure 2: CPU time (seconds) vs number of frames.

3 Serial Tuning

Max Function

From the ICC vectorization reports, we saw that the compiler was unable to vectorize the `max` functions used throughout `central2d.h`. We therefore replaced the `max` functions with conditional variable declarations. For example, we replaced the lines

```
cx = max(cx, cell_cx);
cy = max(cy, cell_cy);
```

in `central2d.h` with

```
cx = (cell_cx > cx ? cell_cx : cx);
cy = (cell_cy > cy ? cell_cy : cy);
```

Because branch prediction is highly successful, the compiler is able streamline how it performs this operation and we noticed a modest speedup of $\approx 10\%$.

Vectorization vs Locality

We observed that the variables `u`, `f`, `g` containing the water level and fluxes at each cell were stored as an array of structures. This arrangement provides good locality because values for a particular cell are stored close to one another in memory. However this arrangement has poor vectorization because functions which pass over the entire domain do not have unit stride access patterns. We considered storing this data as a structure of arrays so that the structure's elements are arrays of all the relevant data that can be easily be passed over using a double `for` loop. However because it was challenging to reformat this data structure in the C++ code, we did not end up changing it in our code. We did observe that this was more easily done in the C implementation that was posted.

Pragmas

One of the other teams suggested we use pragmas to avoid some of the dependency issues noted in the vectorization report. In particular, they advised using `#pragma ivdep` (which asserts that there are no loop-carried dependencies) and `#pragma simd` (which enforces vectorization). We have yet to explore where these should be used in the serial implementation.

4 Parallelization

Domain Decomposition

Our main idea for parallelization was to divide the domain into subdomains for which different threads would be responsible. Each subdomain would be surrounded by layers of ghost cells which would allow each thread to perform several time steps' worth of updates before requiring communication with other threads. The prototypical example discussed in class involved dividing the domain into square subdomains, copying the subdomains

and surrounding ghost cells to private memory, performing updates, and copying the subdomains back to the shared domain.

Rather than copy data between each processor subdomain and the shared domain, we created an expanded shared domain that incorporated layers of ghost cells around each subdomain. This reconfiguration of the shared domain is illustrated in Figure 3 for the case of $p = 9$ threads. Note that the dimensions of the original domain were `nx` by `ny` and the dimensions of the expanded domain are `nx_all` by `ny_all` which are equal to `nx` (respectively `ny`) plus two times the number of ghost cell layers times the number of domains per edge. We believe that this domain decomposition strategy will be less time intensive than the one proposed in class because it removes the phase of copying data between shared and private memory.

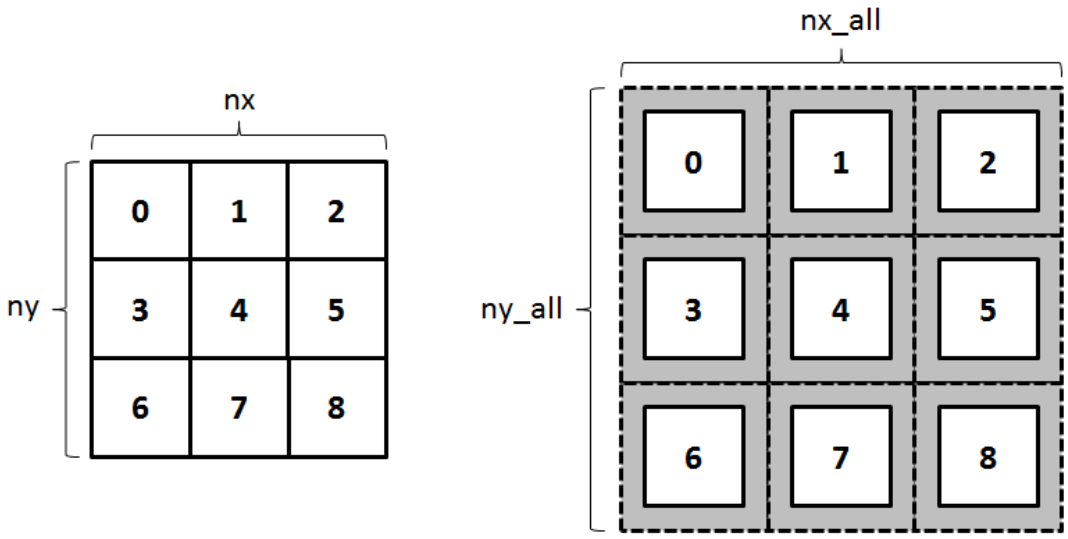

Figure 3: Original shared domain (left) and the expanded shared domain (right). White areas are original domain data and gray shaded areas are ghost cells.

After we implemented this domain decomposition scheme, we realized that instead of storing the entire expanded shared domain in row major form (as we do), it would be more efficient to store the expanded shared domain in row major form by subdomains where each subdomain is in turn stored in row major form. This would improve locality

in terms of cache because each thread will be using contiguous memory in its updates. Because this restructuring would involve rewriting our accessor functions to match this memory reconfiguration, we have not yet implemented this idea.

Although our code is configured to handle any number of layers of ghost cells that is a multiple of three, we have not experimented beyond the default of three layers. Therefore we have not run into the problem of how the updates performed by `compute_step` depend on the maximum wave speeds over the entire domain (`cx` and `cy`). Our current implementation stores stores `cx` and `cy` as vectors of length `nodomains*nodomains` wherein each element is the maximum wave speed within a subdomain. It also uses barriers to enforce that the max wave speeds are recalculated after each update step.

Accessor Functions

We rewrote many of the accessor functions used throughout the code to access the new expanded domain. For example, we created a new accessor `offset` that was used in the `init` function to map the initial data directly into the corresponding subdomains in the expanded shared domain. We also created another accessor `offsetfull` for accessing any element of the expanded shared domain. This accessor was used in the `apply_periodic` function and needed wrap-around capability, much like the given `ioffset` accessor. Another new accessor was `offsetg` that was used in the `compute_fg_speeds` and `compute_step` functions to access all cells in a subdomain, including ghost cells. The last accessor we wrote was `offsetres` which mapped from the expanded shared domain to the original domain. It was used for making the simulation animations.

Parallelizing the `run` Function

Our new `run` function is shown below:

```
template <class Physics, class Limiter>
void Central2D<Physics, Limiter>::run(real tfinal)
{
    bool done = false;
    real t = 0;
    real dt;
    real cx[nodomains*nodomains], cy[nodomains*nodomains];
```

```
    #pragma omp parallel num_threads(nodomains*nodomains)
    while (!done) {
        int tno = omp_get_thread_num();
        #pragma omp single
        apply_periodic();
        compute_fg_speeds(cx[tno], cy[tno], tno);
        limited_derivs(tno);
        #pragma omp barrier
        {
            #pragma omp single
            {
                real cxmax=cx[0], cymax=cy[0];
                for (int i=1; i<nodomains*nodomains; i++) {
                    cxmax = (cxmax>cx[i])?cxmax:cx[i];
                    cymax = (cymax>cy[i])?cymax:cy[i];
                }

                dt = cfl / (cxmax/dx > cymax/dy ? cxmax/dx : cymax/dy);

                if (t + 2*dt >= tfinal) {
                    dt = (tfinal-t)/2;
                    done = true;
                }
            }
        }
        compute_step(0, dt, tno);
        compute_step(1, dt, tno);
        #pragma omp barrier
        #pragma omp single
        t += 2*dt;
    }
}
```

We assume that the number of threads is a perfect square so that the shared domain can be decomposed into square subdomains. In the `#pragma omp parallel` statement, we initialize a team of `tno` threads, one for each subdomain.

We then use `#pragma omp single` to assign one thread the job of copying new data into the ghost cells using `apply_periodic`. We chose to have a single thread perform `apply_periodic` because of the risk of corrupting the data when individual threads copy and write data into their ghost cells. By having one thread do all left-right copying followed by all up-down copying, this issue is avoided. The `#pragma omp single` statment

has an implicit barrier condition so we can be assured that no threads will rush ahead and begin computing wave speeds.

After `compute_fg_speeds` and `limited_derivs`, we added a `#pragma omp barrier` because we need all of the threads to receive the maximum wave speeds before they advance to `compute_step`. Once all threads reach the barrier, another `#pragma omp single` enforced that a single thread passes over the vectors `cx` and `cy` and calculates the maximums `cxmax` and `cymax`.

After the maximum is calculated, all of the threads proceed with their odd and even update steps. Then we have another barrier condition after which a single thread updates the time counter `t`.

Parallelism within Functions

We also gave serious consideration to parallelizing the functions within the `run` function. More specifically, we tested using `#pragma omp for` within some of the functions but found that the communication overhead was too expensive.

5 Weak- and Strong-Scaling Studies

For the strong-scaling experiment, we held the problem size constant at $n = 1080$ cells per side. We tested domain decompositions of 2×2, 3×3, 4×4, 5×5, 6×6, and 8×8 and chose $n = 1080$ because it divided evenly into these dimensions. The speedup results (serial time/parallel time) are shown in Figure 4 and show a curious trend of decreasing speedups between $p = 4$ and $p = 16$ threads and increasing speedups between $p = 16$ and $p = 64$. We had expected to see increasing speedups over the whole range although with diminishing returns as communication costs became more expensive. One possible explanation for our observed results is the inherent variability in timing output; taking more observations for a fixed p would lead to tighter estimates of the speedups. Another explanation is that the observed trend in timings may have to do with the architecture of the cores on the cluster and how threads are assigned.

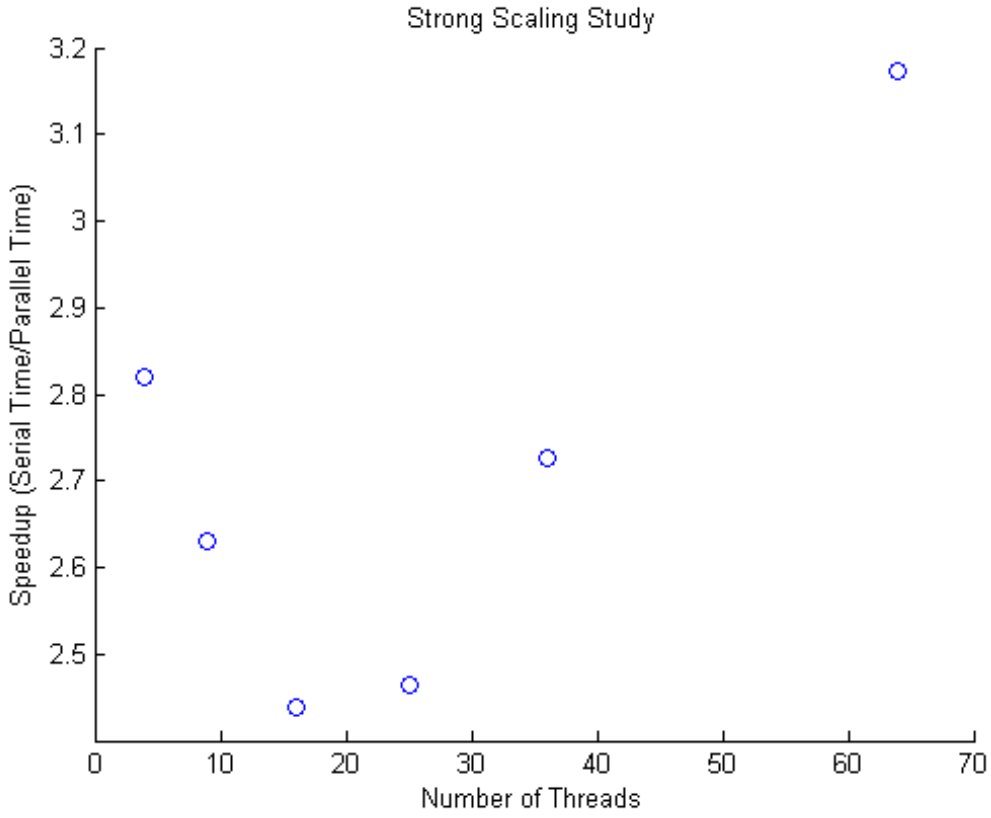

Figure 4: Strong scaling experiment with $n = 1080$ held constant.

For the weak-scaling experiment, we held the work per processor fixed by scaling n and p so that n^3/p was roughly constant. The speedup results, shown in Figure 5 show diminishing speedups. This trend reflects the communication overhead involved with increased numbers of threads.

6 Tuning

We had a hard time debugging the parallel version (including a particularly awful "bad malloc" error!) and ran out of time to fully profile and tune the code. We did however change the order of the for loops from our first implementation and ensured that whenever we iterated over the board, the access was always row wise to ensure more cache hits. This resulted in a speed-up of roughly 30%. All the nested for loops iterate over y dimension

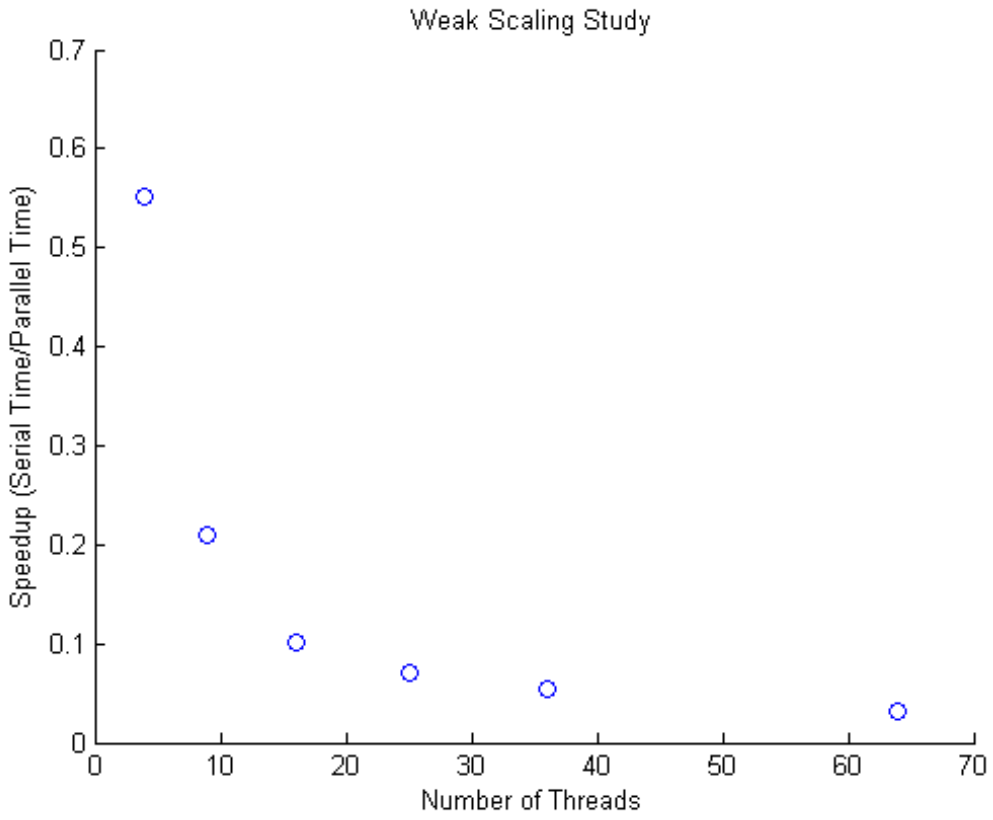

Figure 5: Weak scaling experiment with n^3/p held constant. Baseline of $n = 500$ for serial version.

in the outer for loop and the inner loop iterates over the x dimension. The board is stored in row major format (along the x dimension) in the vector `u_` data structure.

As was mentioned before, one idea that we wanted to test was to store the data in row major form within subdomains and store the subdomains in row major form, similar to the matrix multiplication project. We also would have like to experiment with different numbers of ghost cell layers to see the tradeoff between reducing communication and increasing computation. Another technique we did not attempt was offloading to the Phi boards, but we heard that other groups had been able to do this.

7 Summary

We profiled the serial version and identified the three most time-consuming functions. We tested using parallel for loops within these functions but found the overhead to be

too expensive. We found that the best form of parallelism for this problem was instead to divide the domain into subdomains. We rewrote all of the accessor functions for our new shared domain and restructured the run function to deal with issues of parallelism. We conducted weak- and strong-scaling experiments with mixed results, but ran out of time to do any profiling and tuning of the parallel code.

References

1. https://github.com/dbindel/water

2. https://github.com/kenlimmj