Project 2 - Shallow Wave Equation

Team 15 – Amiraj Dhawan, David Eckman, Xiangyu Zhang

1 Motivation

Our objective was to analyze the performance of the shallow wave equation. The shallow wave equation models the dynamics of water over time. Our implementation featured a square domain with periodic boundary conditions. We profiled a serial version and attempted to create a tuned parallelized version.

2 Profiling

We used *amplxe* to profile the serial implementation. The results of 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 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, which suggests that they are computationally intensive yet may also be easily parallelized if the domain can be divided among threads.

2.1 Scaling Study

To observe how the timings of the three functions scale with respect to the modeling parameters, we recorded their CPU time over different numbers of cells in the domain and numbers of time frames. The timings are shown in Figures 1 and 2.

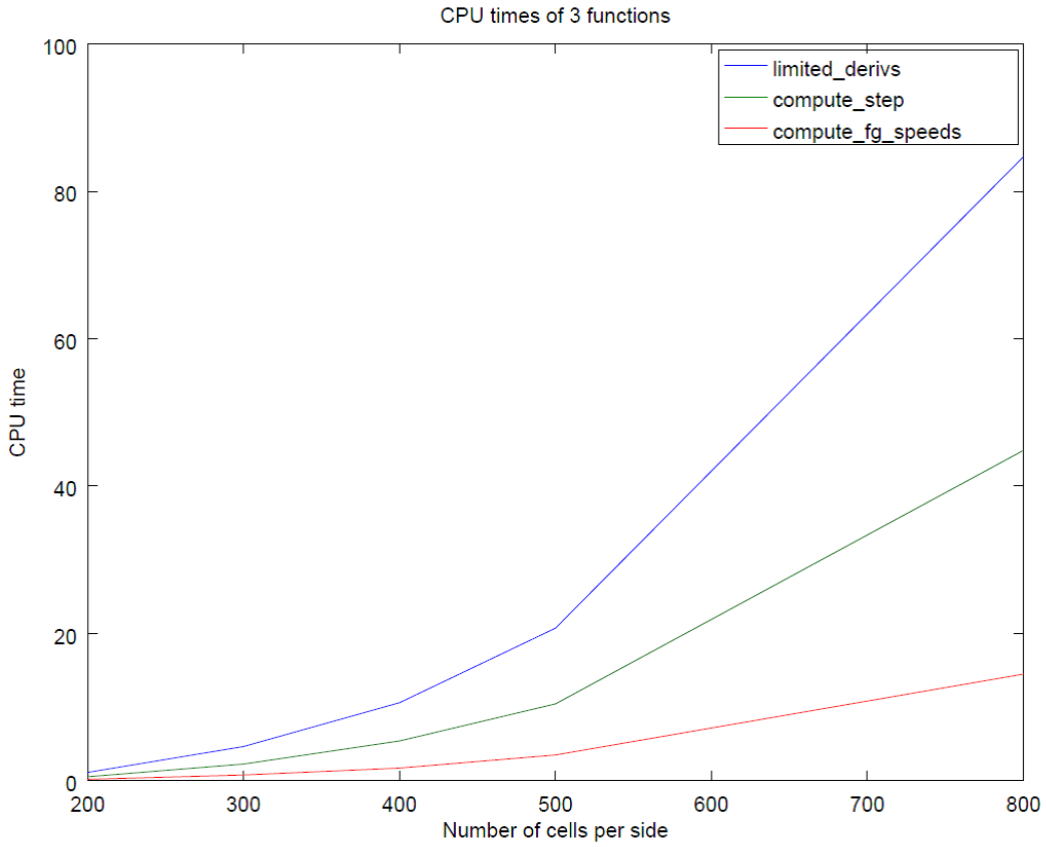

Figure 1: CPU time (seconds) vs number of cells per side.

Figure 1 shows a cubic relationship between the number of cells per side of the domain (n) and the CPU times, since the time appears to grow by a factor of eight as the dimension is doubled. This result matches the observations 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 roughly linear relationship between the number of frames (F) and the times.

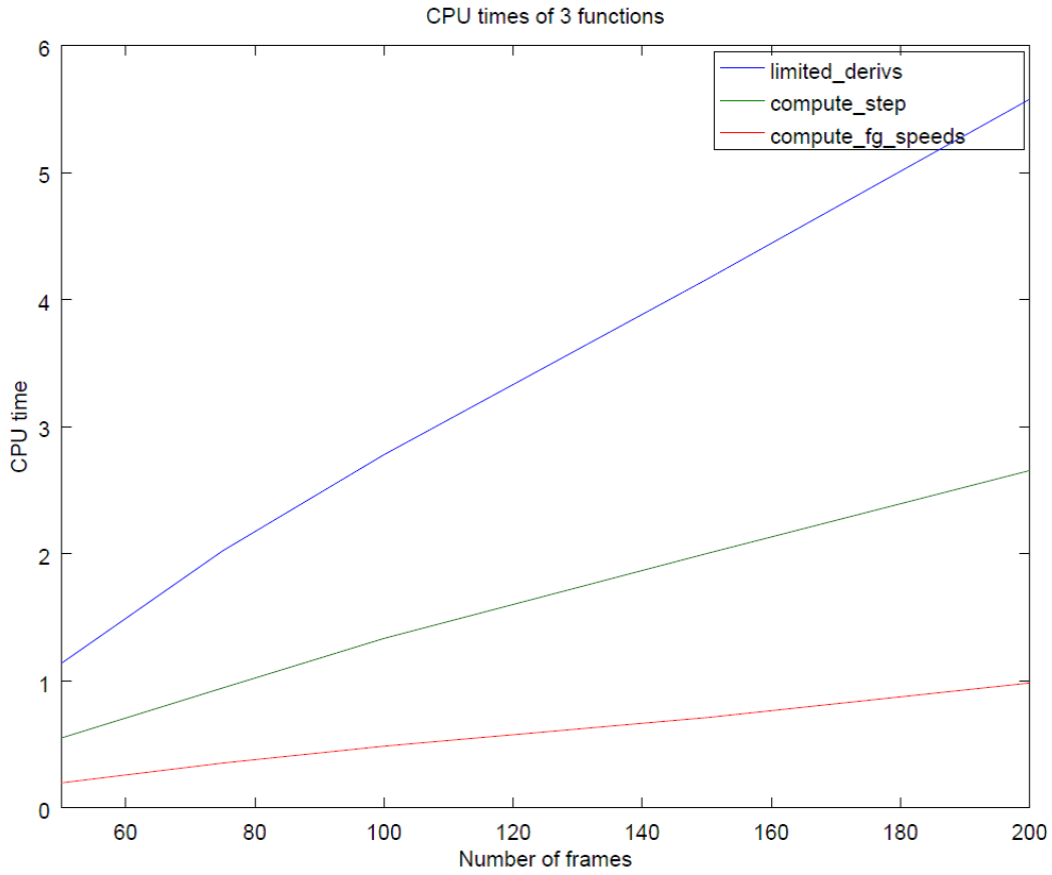

Figure 2: CPU time (seconds) vs number of frames.

3 Serial Tuning

3.1 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.

3.2 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. We have not yet made this change in our code.

4 Parallelization

4.1 Domain Decomposition

Our main idea for parallelization was to divide the domain into sub-domains for which different threads would be responsible. Each sub-domain 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. One issue with this approach, which we have not yet resolved, is that the updates depend on the maximum wave speed over the entire domain. Thus it will be difficult for threads to advance time in parallel without knowing this value, or at least a bound on it. With a working implementation of this scheme, we would then need to experiment to determine the number of threads and ghost cells that represent the best tradeoff between computation and communication time.

4.2 Parallelism within functions

Another approach we are experimenting with centers around the `run` function because it is called at every time frame. The `run` function is shown below:

```
template <class Physics, class Limiter>
 void Central2D<Physics, Limiter>::run(real tfinal)
 {
    bool done = false;
    real t = 0;
    while (!done) {
        real dt;
        for (int io = 0; io < 2; ++io) {
            real cx, cy;
            apply_periodic();
            compute_fg_speeds(cx, cy);
            limited_derivs();
            if (io == 0) {
                dt = cfl / std::max(cx/dx, cy/dy);
                if (t + 2*dt >= tfinal) {
                    dt = (tfinal-t)/2;
                    done = true;
                }
            }
            compute_step(io, dt);
            t += dt;
        }
    }
}
```

To parallelize the `run` function, we would declare a parallel environment outside the `while` loop using `#pragma omp parallel` which creates a team of threads. Because the `for` loop over `io` only makes two passes and the second pass requires data from the first pass, we would not attempt to parallelize this loop.

The `apply_periodic` function copies the values from other cells into the ghost cells. Because of the dependencies between copying left-right and then copying up-down, we would choose to use `#pragma omp for` with implicit barriers for the two sets of double for loops in the function.

We would again use `#pragma omp for` within the `compute_fg_speeds` function, but this time we would use the `nowait` argument to remove the implicit barrier since the next function (`limited_derivs`) does not depend on the output of `compute_fg_speeds`. In order to get the overall value of `cx`, determined by the maximum of the `cx` values from the double `for` loop in `compute_fg_speeds`, we would store all of the `cx` values in an array and then have the master thread calculate the maximum over the array.

We would also use `#pragma omp for` with implicit barriers for the double for loop in the function `limited_derivs`. Next, we would have the master thread calculate the maximum `cx` and perform the nested `if` statements in the `run` function. Lastly, we would use `#pragma omp for` with implicit barriers throught the function `compute_step`.

An issue with this parallelization method is that threads would be writing to the same data structures, although in different locations. These operations may not be protected, or if they were, may prove to be very expensive in terms of CPU time.

5 Tuning

We have not yet taken the opportunity to tune the parallel code since we have not yet debugged the domain decomposition scheme. Once we get a working parallel implementation, we will be able to set up both weak- and strong-scaling experiments.

6 Failed Methods and Future Ideas

We attempted to use the `restrict` keyword to further optimize the serial implementation but did not find any opportunities to use it on the pointers in the C++ code. We also have yet to look into opportunities to unroll the double `for` loop sections to help the compile vectorize them. Another technique we have not yet implemented is offloading to the Phi boards.