CS 5220 Assignment 2: Shallow Water Simulation

Unmukt Gupta, Eric Lee, Scott Wu

1 Introduction

The purpose of this assignment is to improve performance of the the shallow water simulation, utilizing multiple processor cores as well as the Xeon Phis to attain massive performance increases. The shallow water simulation is a grid based physics simulation where cells in a single time step are dependent on the 4 adjacent cells from the previous time step. To improve performance, we implement vectorization, domain decomposition and offloading as three separate tasks that we split among our three group members. While we had success with each task, combining them proved a bit harder. We started with Prof. Bindel's base C++ code and rewrote much of it into C. This was necessary, as we'll discuss later, for offloading to the Phis.

2 Changes to the Code Base

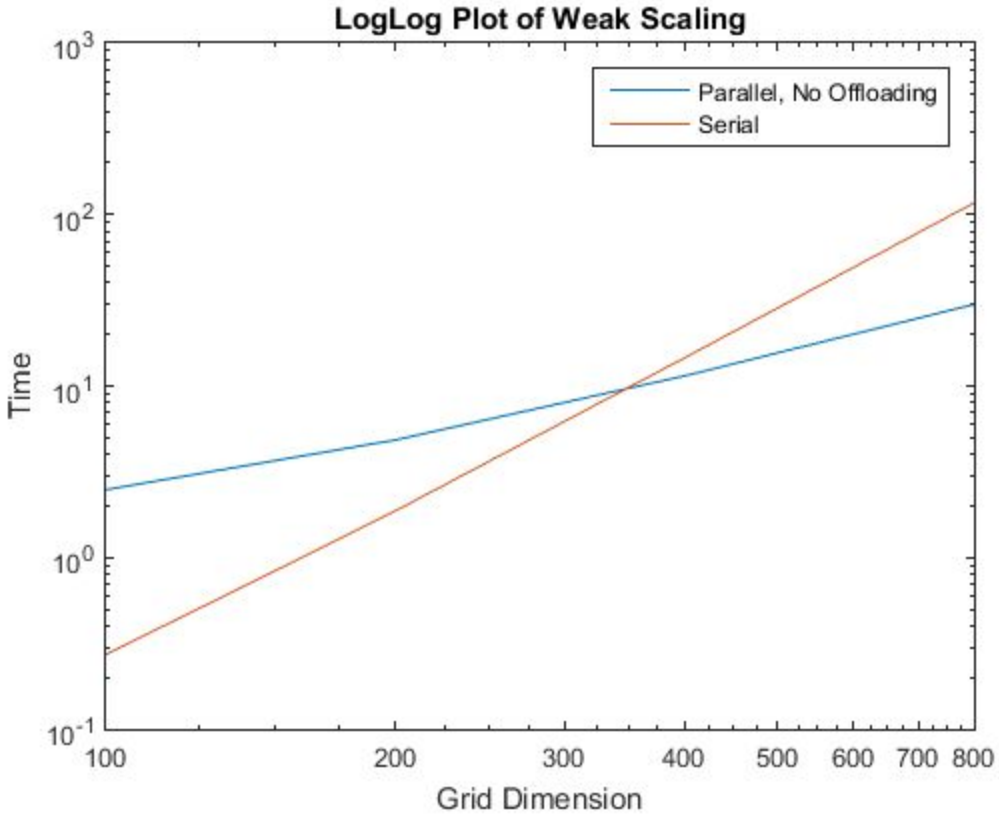
We tried two different schemes to improve the data structures in the code for performance. First, replaced the each vector of vectors with a triplet of arrays. We also tried replacing each vector of vectors with instead an array with element stride three. The former, although much messier, was predictably much faster than the latter due to its unit stride.

3 Domain Decomposition

In the same vein as the game of life example discussed in class, the high level idea of domain decomposition is assigning each processor a single block to work with. This block can be kept in cache and would be highly accessible to the processor. To enable processors to take multiple time steps, each block is padded with ghost cells. Since cells are dependent on adjacent cells, one layer of cells around the border are made obsolete after every step.

More precisely, we cut our grid into p blocks, where p is our number of threads, and then pad these blocks with ghost cells. In the OpenMP model, our grid is in shared memory while each processor holds its pth block in private memory. If we consider a "half step" as one single predictive + corrective step, then each processor will run on its own block, copy the results back to grid in shared memory, and copy the updated ghost cells from the grid in shared memory to continue running. The value of our grid at (i,j) depends on a halo of ghost cells three deep from the previous step. Thus, if each block has only three non-obsolete ghost cells, then we must synchronize on the full step. Moreover, by maintaining a halo of ghost cells for each domain, in accordance with the periodic boundary condition, also saves the computation cost of applying it, at each step, to the main grid.

Having only three ghost cells is far too much synchronization; the biggest challenge was finding a good balance between the number of ghost cells and the amount of synchronization. This was further complicated by the fact that our step size is dependent upon each step due to the limiter.



We decided to have 12 ghost cells, though for four full time steps before synchronization was necessary.

4 Vectorized Instructions

We both attempted auto-vectorization and manual vectorization. Automatic vectorization with through a #pragma omp simd generated code running slower than the code without the pragma. Presumably, this is because much of the arithmetic logic was too complicated for the compiler to vectorize properly.

Manual vectorization yielded far better results. The Phis supported AVX-512, which meant we could do up to 16 floating point operations at once. In any loop over the grid, we can load whole 16-element vectors into a register. Since we only use basic arithmetic operations, we could easily string all the calculations together, then writing back all 16 results. Ideally, we would get nearly 16 times the speedup, but problems with implementation have severely cut this number.

The first problem we encountered was unaligned memory operations. If an instruction attempts to load or store values from a non-multiple-of-16 memory location, a segfault is raised. In the

corrector step, we need to load adjacent cells, offset by 1. AVX-512 documents an unaligned load and store operation, but compiling gives an unsupported instruction error.

```
float * arr = new float[32];
...
_mm512_load_ps(arr); // Works
_mm512_load_ps(arr+1); // Segfault
```

5 Offloading

Automatic offloading is accomplished using a #pragma offload target(mic). While presumably an easy fix (at least according to intel), in reality, offloading was much harder. Offloading is done through a single pragma:

```
#pragma offload target(mic) in() inout() out()
{
	//do stuff
}
```

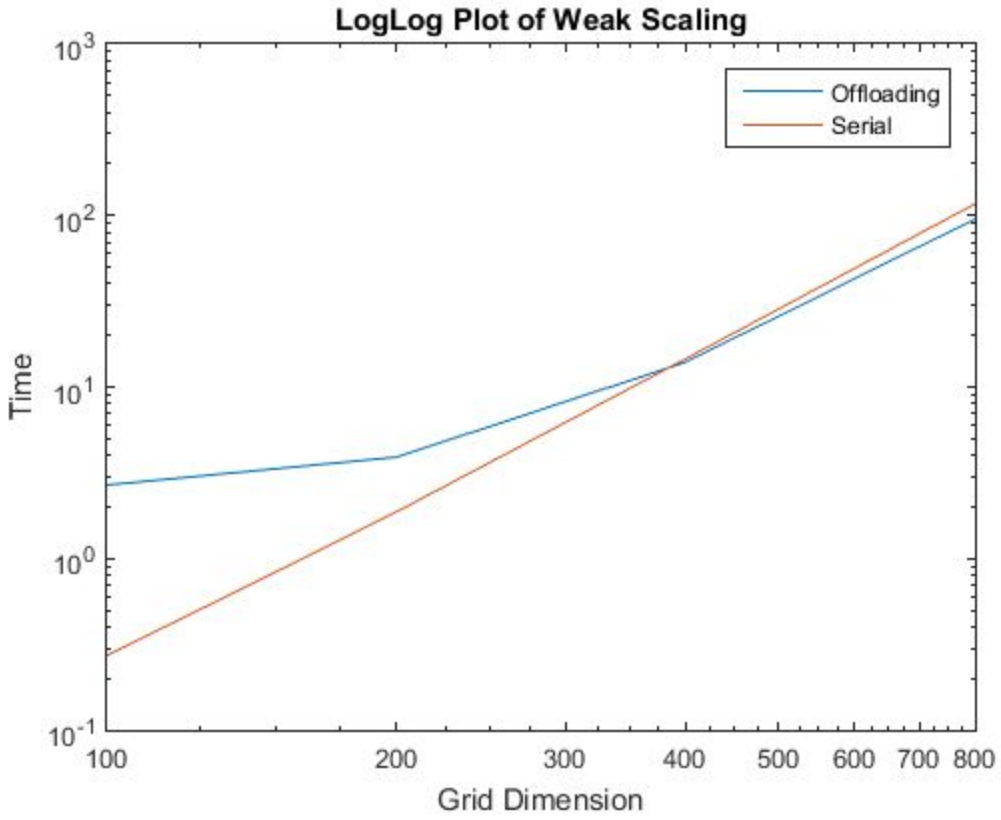
Where the arguments for in(), inout(), and out() are the values copied in, the values maintained in memory, and the values copied out respectively.

The lack of cohesive documentation about the intricacies of offloading led us to create smaller testbed for understanding offloading behavior. From this testbed, we learned a variety of things, some of which could be simply due to our lack of in-depth knowledge:

- ❏ Offloading vectors of non-primitives is not allowed (or at least, we failed to develop a hacky way of doing so). This necessitated a rewriting of the code.
- ❏ The overhead for offloading is roughly one second.
- ❏ Many compiler flags clashed with offloading flags e.g. we found that the -ipo flag threw warnings during compilation and actually decreased the performance of our code in a few cases.
- ❏ AVX-512 instructions aren't as mature as their older counterparts. Furthermore, as we found out later, one needs to explicitly identify AVX-512 as running on the Phis, and take steps to segment host code with mic code.
- ❏ ICC and ICPC behave somewhat differently for offloading. While ICC doesn't compile functions inside an offloaded section without a __declspec(target(mic)) declaration, ICPC does -and successfully compiles as well.
- ❏ Offloading behaves unpredictably with C++ classes. If one tries to offload arrays declared inside a class (either as private, public, or protected), depending on a number of conditions, one will get a segfault after the Phi fails to properly copy the array to its own memory. Moving the grid's many arrays outside the class declaration solved this problem.
    - ❏ We weren't able to identify why the cause of this behavior.

- ❏ We also removed the templating; it was causing the compiler to throw a wide variety of warnings

❏ Compiler optimizations will oddly optimize "out" offloading, thus leading to segfaults. We observed that zero compiler optimization allowed us to compile successfully, but O1 and O2 levels optimizations offloaded most functions to the phis (as they were already inside a phi kernel, this led to segfaults). An optimization level of O3 removed this problem for us, strangely enough



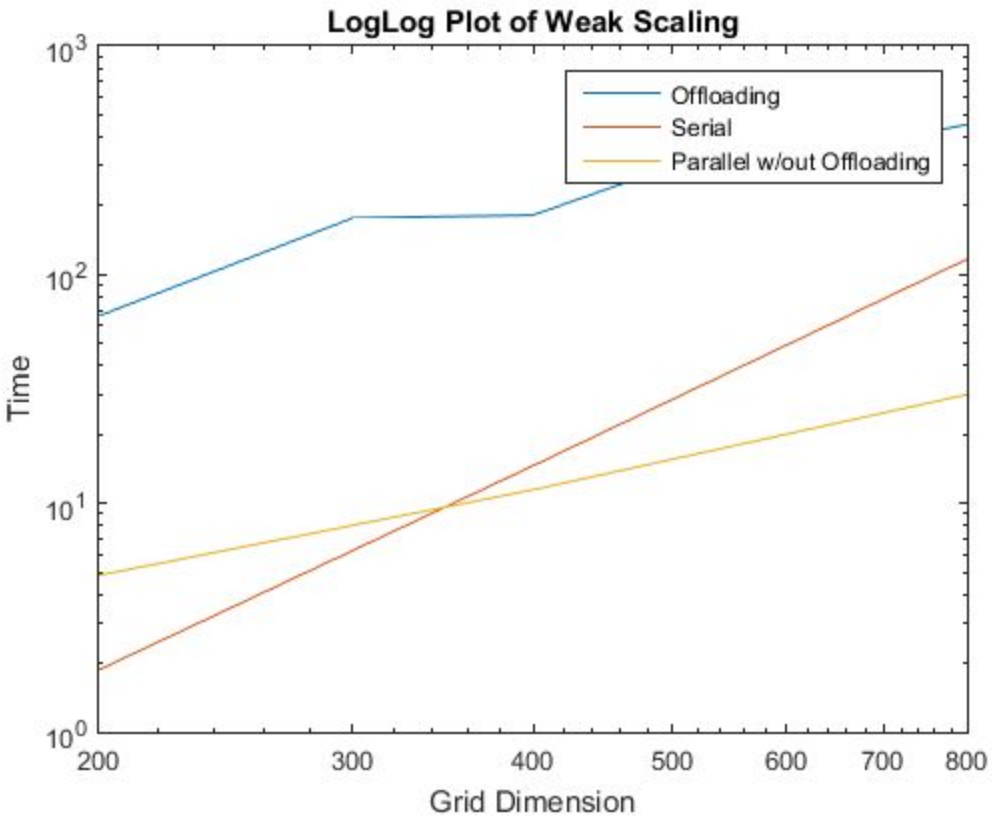
In the following graph, we have a loglog plot of a weak scaling study, including time taken to run our offloaded, parallel version as well as the C++ serial version. Note that offloading has a much higher overhead, but scales better than the serial version. The number of threads we used was the default (236).

6 OpenMP and Parallelization

To utilize the parallel potential of domain decomposition, we wrap a parallel pragma around the while loop of the code. This spawns a set of threads that we can use to assign to individual blocks. Work is split into domains by a parallel for over each block. First, we copy data from the shared grid to private arrays. Then we compute the smallest non-zero dt based on the maximum wave speeds. This is a critical section which ensures that all processors take the same timesteps over the same amount of time.

```
#pragma omp parallel shared(grid, dt, done)
  while (!done)
    #pragma omp for
    Copy from shared grid and compute timestep

    #pragma omp barrier

    #pragma omp for
    Compute speeds, derivatives, and take multiple steps

    #pragma omp barrier

    Copy results back to shared grid
```

7 Combining Everything



We roughly combined our three implementations to get a final one with domain decomposition and offloading (the blue line). It seems to scale by roughly the same amount as the non-offloaded parallel implementation, but slower. This is probably due to our inefficient combination; we offloaded at every single time frame, which certainly contributed many more memory operations than needed. Another pitfall lies in poor parallelization. Our algorithm contains two barrier points, which potentially costs lots of idle time. The OpenMP parallel for pragma also cannot

efficiently employ a job queue style workload. In hindsight, investigating other pragmas, such as master and task, could improve parallelization performance.

We managed a higher performance implementation with offloading, vectorization, and domain decomposition, but kept getting memory problems past a grid size of 400 by 400. We weren't able to identify the problem, but our final results follow.

100	0.1251030
200	0.1618872
300	0.2803910
400	0.5958390

8 Load Balancing

We decided to manually load balance by choosing grid sizes that mapped nicely to a large number of threads. We could have also written a load-balancing script to choose the number of threads, grid size, ghost cells, and other such parameters, but we didn't have the time.

9 Concluding Thoughts

Optimistically, using the Phis, we could have seen massive speed-ups with vectorization; having 512-bit vector registers means a pipeline of 16 single precision numbers. Combined with 60 processors, this would give us a theoretical "pipeline" of 960 single precision numbers, or up to 960 arithmetic operations per cpu cycle.

However, we believe OpenMP to be too simple of a parallel tool to provide significant speed-ups to tasks not embarrassingly parallel; too much is abstracted away from the developer at times, leading to poorly-performing parallel code. This certainly proved to be the case when we tried offloading.

On another note, if we didn't generate the simulation video, did we do the simulation at all? This existential question was not within the bounds of this project, because we weren't required to generate the simulation video. However, one fun thing to consider is parallel generation of the simulation video. Having a single processor write the simulation is certainly far too inefficient. A much better method would be having each processor write its own set of files corresponding to its private block of the domain, then stitching together the files in a post-processing phase.