CS 5220 Project 2: Stage 1

Elliot Cartee (evc34), Patrick Cao (pxc2), Ben Shulman (bgs53)

1 Introduction

For this project, our goal is to optimize a simulation of the shallow water equations with periodic boundary conditions. To accomplish this, we will use three main approaches:

- Use profiling and vectorization reports to determine which parts of the code would benefit most from tuning
- Tune the serial code to maximize performance on a single core
- Implement domain decomposition so that our highly tuned serial code can be run in parallel.

We will then do scaling studies for our tuned parallel code to see how much we improve over the standard serial code, and compare our results to performance models.

2 Profiling

In order to identify what areas of the original `C++` code could receive benefits from tuning, we began with the standard profiling tool **Intel VTune Amplifier XE**. Using the profiler to identify which functions were slowest, we found that in the basic dam break example, the majority of time was spent in the following functions:

function	time (seconds)
`limited_derivs`	1.323
`compute_step`	0.683
`compute_fg_speed`	0.236

We also found that no more than about 20 milliseconds was spent in any other function. Considering that these functions contain the vast majority of the computations, it is not surprising that they took the most time to complete.

We then profiled each of these functions individually to identify which parts were slowest. Unsurprisingly, we learned from profiling these functions that the steps which did real computation inside of `for` loops was the most expensive. Unfortunately we were unable to profile memory and cache accesses as the Haswell architecture on the chips we are using does not support much memory profiling via **VTune Amplifier**.

3 Vectorization

One of the key ways of improving performance is by maximizing instruction level parallelism through vectorization. In order to see what loops were and were not being vectorized, we began by looking at the vectorization reports generated by `ICC`. We found that none of the `C++` code was being vectorized due to assumed anti/flow dependencies in every `for` loop. Since we knew from profiling that the vast majority of time was spent in the functions `limited_derivs`, `compute_step`, and `compute_fg_speed`, we realized we could make large performance gains from vectorizing these functions as much as possible.

Our first attempt at improving vectorization was to resolve incorrectly assumed dependencies using `#pragma ivdep`. This pragma tells the compiler to discount any assumed dependencies and only consider proven dependencies. This improves vectorization slightly for the `C++` code, but much of the code still does not get vectorized. Unfortunately, this vectorization sometimes led to only minor improvements, and was sometimes actually worse (in which case the compiler did not vectorize).

In an attempt to make vectorization more efficient we turned to Intel's vectorization guides, and found that the layout of the original data structure is not ideal[1]. The problem is that the the vector u was originally stored as Vector of Arrays. Each array (u cell) is length 3, thus making the memory structure of the vector u be `[U0, U1, U2, U0, U1, U2, ...]`. Most of the computations in the slow parts of the code involve only a single dimension of u at a time rather than using all 3 dimensions of each vector at the same time. This suggests that this Vectors of Arrays structure (VOA) can be improved by separating each vector into 3 vectors, one for each dimension, which allows unit stride access to different cells when working in the same dimension.

We implemented this method for each vector and found that while vectorization increased, overall performance actually decreased. This suggests that even though the VOA style did not seem like the best structure the compiler was still able to optimize better with the original VOA memory structure than the new memory structure.

Due to our struggles with vectorization in `C++` and the rewriting of the simulation code into `C` by Professor Bindel we decided to switch over to optimizing the new `C` code as it gives us more control over memory alignment, restriction, vectorization, etc. which `C++` made difficult. Further we are generally more comfortable with `C`.

4 Domain Decomposition

Given the local nature of the shallow water physics, this simulation is an ideal candidate for domain decomposition. Because calculating the water height and momentum at each cell depends only on nearby cells, this problem can be broken into subdomains, which processors can compute independently for several steps before needing to synchronize. In

[1] https://software.intel.com/en-us/articles/memory-layout-transformations

order to make implementing domain decomposition easier, we made some basic simplifying assumptions:

- All simulations take place on a square grid of size $n \times n$.
- The number of processors p is a perfect square.
- n is divisible by $\sqrt{p}$

We will also use b to refer to the batch size, meaning that $2b$ is the number of timesteps computed by each processor between synchronizations. (Note that we compute $2b$ instead of b steps because of the grid offset in the Jiang-Tadmor central difference scheme being used).

For our implementation of domain decomposition, we maintained a global copy of the domain which is used to store the current state of the entire simulation. We then divide the domain into p equally sized subdomains, with one processor being responsible for each. There were two approaches we considered for allowing each processor to compute the subdomain it was responsible for. The first approach is to keep only one copy of the whole domain and each processor will read and write to this shared memory. The second approach would be to have each processor maintain a copy of its subdomain and periodically synchronize up with the global copy of the domain. The first approach requires less copying of memory back and forth. However, since the subdomains overlap with each other, it requires us to guarantee thread safety for the shared global domain. The second approach requires more copying of memory but it can be done without needing to worry about thread safety. In addition to decreasing implementation complexity, the second approach is able to use the cache more efficiently. Since each subdomain will now be stored in a contiguous region of memory, this allows us to make more efficient use of spatial locality.

Using this approach, we developed the following algorithm (written here in pseudocode):

```
1: Apply periodic boundary conditions to domain
2: Find max_speed in domain
3: dt ← 0.9 * cfl/max_speed
4: for each processor do
5:     subdomain ← relevant part of domain
6:     for i = 1,2,...,b do
7:         subdomain ← COMPUTE_STEP(0)
8:         subdomain ← COMPUTE_STEP(1)
9:     end for
10:    domain ← subdomain
11: end for
```

We note that we have had to decrease the size of the time step `dt` as we are now computing multiple time steps between each calculation of `dt`. While decreasing the time step increases the amount of computational time for simulations, decreasing the time step can lead to instability of the numerical method. Our current choice of 0.9 times the CFL condition of 0.45 is heuristic, and could very likely be improved.

5 Results

We have not yet done an in-depth performance analysis of our domain decomposition code. Initial rough estimates show that we do get a speedup of our own code as the number of processors increases but only to a certain point, at which the overhead makes it no longer worth it. However, this speedup is not linear in the number of processors (perhaps it is logarithmic?).

6 Future Work

While we were able to develop a working version of domain decomposition, we have quite a bit of further work to do, as well as areas to improve our current code:

- Profile our code to determine where most of the time is spent, and whether we are fully utilizing all of our parallel cores
- Use these profiling results to improve the performance of our code
- Try using one `omp parallel` pragma with barriers for synchronization (and possibly a master thread) instead of using multiple `omp parallel for` pragmas.
- Calculate the time step `dt` in a more intelligent way
- Offloading computation to the Phi accelerator boards rather than just using the Xeon E5 cores
- Completing weak and strong scaling studies and comparing these results to performance models