Water: Stage 1

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

Introduction

In this project we are optimizing a simulation of the shallow water equations. This simulations models how water flows. Boundary conditions are periodic (i.e. water flows out one side and into the other). Our goals are to profile the given code, parallelize and tune it to improve performance. We then will do scaling studies and performance models for our tuned code to see how much we improve over the standard serial code.

Profiling

In order to identify the parts of our code that could most benefit from tuning and parallelization we began by running a standard profiler tool: VTune Amplifier. We began by running the profiler over the entire run to identify functions which were slowest. We found that the functions that took up the vast majority of processing time were limited_derivs, compute_step, and compute_fg_speed. They took 1.323 seconds, 0.683 seconds, and 0.236 seconds respectively while no other function took more than 20 milliseconds. It is unsurprising that these functions took the most time as they are the functions which are doing most of the calculations at each step of the simulation.

We then individually profiled each of these functions individually to identify which parts of them were slowest. Unsurprisingly the results of profiling these functions told us 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.

Vectorization

To understand how the Intel compiler was managing (or rather: not managing) to vectorize the starter code we also looked at the vectorization reports that are generated by ICC. None of the code was being vectorized at all because of assumed anti/flow dependencies throughout every for loop. The combination of poor vectorization and knowing that the slowest functions in the code led us to first try to vectorize the main three functions: limited_derivs, compute_step, and compute_fg_speed.

The first thing we tried to do to improve vectorization was to resolve assumed dependencies that didn't truly exist 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 doesn’t get vectorized, or vectorization hsa only minor improvements, or is even worse (and thus doesn’t get vectorized).

To try and make vectorization more efficient we turned to Intel’s vectorization guides and found that the data storage structure used is not the best.[1] The problem is that the original storage structure of the data such as the vector u is it is a 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 for loops in the slow parts of the code do computations that involve only a single dimension of U at a time rather than computations that are based off of all 3 dimensions of each vector. This suggests that this Vectors of Arrays structure (VOA) can be improved by separated each vector into 3 vectors, one for each dimension, which allows unit stride access to different cells when accessing the same dimension.

We implemented this method for each vector and found that more parts of the code would vectorize but the overall cost increased. This suggests that though the VOA style did not seem like the best structure the compiler was still able to optimize better with 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.

Domain Decomposition

This type of simulation is ideal for domain decomposition. The shallow water simulation has locality when calculating the next timestep for each cell which means the problem is prime for being broken into subdomains which can be passed off to processors to be computed for several steps before synchronizing. In order to implement domain decomposition for this problem we made some assumptions to simplify the cases we needed to handle:

- The simulation is on a square grid.
- The number of subdomains (processors) is a perfect square
- The side length of the simulation is divisible by the square root of the number of subdomains.

Throughout our implementation we refer to p as the number of subdomains and in general will explain our implementation for $p = 4$. However, our implementation works for any p for which our previous assumptions hold. We also use b to refer to the batch size, each batch consist of $2b$ steps to ensure that we end on the true grid rather than the offset grid.

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

We keep a global copy of the domain that holds the state of the current universe. We then divide the domain into 4 equal subdomains. Each subdomain will be computed by a processor. There were two approaches we considered for allowing each processor to compute the subdomain it was responsible for. The first approach is to keep 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 lower implementation complexity, the second approach is able to use the cache more efficiently. Each subdomain will be copied into a region on contiguous memory. Since the subdomain will be stored in a contiguous block it will allow us to make more efficient use of spatial locality.

Using this approach gives us the following algorithm. First we apply the periodic boundary conditions to the global ghost cells. Note that we need $3b$ ghost cells and that each subdomain will need $3b$ ghost cells (that extend into the other subdomains). We then calculate the maximum dt such that the CFL criterion holds true. Because we may be calculating using this dt to calculate more than 2 steps we need to back-off our dt slightly to make sure we do not take too large a step and violate the CFL criterion. Currently we simply reduce dt by 10%. Next we copy memory from the global domain into each subdomain's memory (parallel step). For each subdomain we run the subdomain for a batch of size b as if it were its own domain (thus not requiring any changes to the simulation kernel). This step is done in parallel. After all subdomains have completed running we then copy memory from each subdomain back to the global domain to synchronize data. We then repeat, going back to periodic boundary condition application until we reach completion.

Results

We haven't yet done much 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 (to a certain point, at which the overhead makes it no longer worth it). The speed up is not linear in the number of processors (logarithmic perhaps?).

Future work

Now that we have a functional version of domain decomposition we plan on profiling it to see what parts of it are slow and where we are not fully utilizing our parallel cores. We also plan on working on offloading to the phi boards rather than just using the Xeon E5 cores. We also will be tuning our code in response to the results of our profiling. Finally we will be doing (weak and strong) scaling studies as well as creating performance models.