Shallow Water: Final Report

Group Number: 2
Members: Nimit Sohoni (nss66), Edward Tremel (ejt64), Ian Vermeulen (iyv2)

IMPORTANT NOTE: You will notice that the timestamp for this commit is after midnight. Please use whichever version you judge appropriate (we did have a submission right before midnight, along with a couple very soon after it); we were running some scripts on the cluster and it had slowed down significantly from around 11:40 onwards. However, they did finish so we have included the results.

Optimizations used or attempted:
Restructuring and directives to enable greater vectorization, informed by results from profiling.
Parallelization with OpenMP
Notes: The main code of our project is all located in the cpp/ subfolder. We toyed with the idea of making the parallelization changes to the released C code as well but we decided to focus on the C++ version, as we incorporated many of the ideas of the C serial code to optimize the C++ version anyway.

More details:

1. Vectorization
    a. Used compiler directives such as '#pragma ivdep' to encourage the compiler to vectorize loops and ignore possible dependencies that did not actually exist.
    b. Restructured the code to make better use of memory locality and vectorization.
        i. Changed the data structure from a vector of vecs to a 'flat' data structure
        ii. Also used floats instead of 'real' for increased speed
2. Parallelization
    a. Used OpenMP to create a parallelized main "run" loop
    b. Split the simulation grid into rectangular blocks based on the number of threads available.
        i. For an a-by-b grid with p threads, we set the block size to ceil(a/√p)-by-ceil(b/√p), based on the assumption that the grid is roughly square (so the threads should be distributed equally in the horizontal and vertical directions). If the grid is not square, or if p is not square, there may be more blocks than the number of processors, due to small rectangular blocks on the fringes of the grid. In this case, we assign multiple blocks to each processor sequentially in row-major order (e.g. processor 0 gets the blocks in position [0,0] and [0,1]). Although the extra blocks assigned to each processor can only be computed sequentially, each processor will usually have 3 or fewer blocks, so this should not affect performance too much.
        ii. Each block also has 4 layers of ghost cells, initialized to the values of cells in neighboring blocks, so that each thread can compute 2 time steps

before communication is necessary. This reduces communication overheads and also ensures that the grid being communicated between blocks is always the "normal" grid used in the even-numbered steps, rather than the "offset" grid used in the odd-numbered steps.

iii. The blocks are managed by a "wrapper" class with its own **u** and **v** arrays representing the "current" and "next" state of the grid, and each block is initialized at the beginning of a batch of 2 timesteps by copying data into thread-local memory (an instance of `Central2D`) from the appropriate section of the large **u** grid. When a block finishes its local timesteps, it copies its data out to the wrapper class's **v** grid. Since the **u** array is read-only and the **v** array is write-only, the blocks can start asynchronously without concern that one block's write will interfere with another block's read. At the end of a timestep batch we use a barrier to wait for all blocks to finish writing to **v**, and then all threads swap their pointers to **u** and **v** (so that the "next" grid becomes "current," and each block starts reading from the previous step's output at the start of the next step).

iv. We avoided using any synchronization to implement the periodic boundary condition by having blocks copy some data out to the ghost cells of the large **v** grid when they write out their results. Specifically, edge blocks copy their last 4 rows or columns of cells to ghost cells on the opposite edge, and corner blocks copy their corner cells to ghost cells on the opposite corner. This achieves the effect of the periodic boundary condition in the same step as writing out results from blocks, which both eliminates the need to call `apply_periodic` on the entire large grid and eliminates the need for a critical section or barrier to stop blocks from reading the grid while `apply_periodic` was being executed.

v. We also computed the values of *dt* in a manner that avoids synchronization overheads. At the end of each batch of 2 timesteps, each thread computes *dt* using the maximum *cx* and *cy* over its block, and writes that value into a shared array at the index corresponding to its thread number. Then at the beginning of a batch (after the barrier that ends the previous one), each process scans this array and uses the smallest value in it as *dt* for its next batch of 2 steps. This results in every process using the value of *dt* that would have been computed by maximizing *cx* and *cy* over the entire grid, without using synchronization to stop and compute it over the large grid.

Vectorization Results:

The following are results from our old report. Although we have significantly improved the vectorization since then, amplxe is down so we don't have stats.

When we initially ran the amplxe-cl profiling tool on the original code and default dam_break test case, we found that by far the bottleneck was the limited_derivs function. The generated report is below:

amplxe: Executing actions 50 % Generating a report	Function	Module	CPU Time	Spin Time	Overhead Time
Central2D<Shallow2D, MinMod<float>>::limited_derivs	shallow		1.148s	0s	0s
Central2D<Shallow2D, MinMod<float>>::compute_step	shallow		0.659s	0s	0s
Central2D<Shallow2D, MinMod<float>>::compute_fg_speeds	shallow		0.230s	0s	0s
[Outside any known module]	[Unknown]		0.021s	0s	0s
_IO_file_xsputn	libc-2.12.so		0.015s	0s	0s
_IO_fwrite	libc-2.12.so		0.010s	0s	0s
Central2D<Shallow2D, MinMod<float>>::solution_check	shallow		0.004s	0s	0s
SimViz<Central2D<Shallow2D, MinMod<float>>>::write_frame	shallow		0.004s	0s	0s
Central2D<Shallow2D, MinMod<float>>::Central2D	shallow		0.003s	0s	0s
Central2D<Shallow2D, MinMod<float>>::run	shallow		0.001s	0s	0s
std::array<float, (unsigned long)3>::operator[]	shallow		0.001s	0s	0s
Central2D<Shallow2D, MinMod<float>>::offset	shallow		0.001s	0s	0s
do_lookup_x	ld-2.12.so		0.001s	0s	0s

Initially, we simply added #pragma ivdep statements to several loops and in some cases changed the loop ordering to improve vectorization. However, this did not work for many of the loops because they contained non-vectorizable operations: the structures with the function and solution data were all in the form of vectors of vecs (where a vec was a data structure containing three reals). This made it unreasonable for the compiler to vectorize complicated operations relating to these, as the memory layout did not facilitate it. We realized that we could separate the three ‘components’ of the solution data and lay these out after one another instead of contiguously listing all three dimensions of the data at one particular grid point, as in professor Bindel’s C code, and similarly inferred from this code that it would be more efficient to loop on this ‘dimension’ as the outermost loop variable, rather than as the innermost. Therefore, we decided to change the memory layout in this manner, as a flat array of floats (we did not realize that the real datatype was defined to be float, and by the time we did had already changed it). We changed the offset functions to take the inputs (grid x position, grid y position, solution component) and convert that to an index in the appropriate array.

Unfortunately, the amplxe_cl analysis tool has not been working in the past few days, so we are not able to give a detailed profile of the vectorization. However, in the ipo_out report, we found that most of the computation-heavy loops were successfully vectorized. The only ones that weren’t were either parts of functions that only ran once and therefore were not a major part of the runtime (i.e. init, check_solution), loops that couldn’t be vectorized because they might have had early termination (also in init), and sometimes the second loop in compute_step which even after the #pragma ivdep directive sometimes says there is an assumed FLOW or ANTI dependence (sometimes I don’t seem to see this message in the report, which was strange). All the rest, including both loops in limited_derivs, the other compute_step, and the loops in compute_fg_speeds, were successfully vectorized.

Here is the amplxe-cl report after the initial changes we made (in the first stage of the project)

amplxe: Executing actions 50 % Generating a report	Function	Module	CPU Time	Spin Time	Overhead Time
Central2D<Shallow2D, MinMod<float>>::limited_derivs	shallow		1.149s	0s	0s

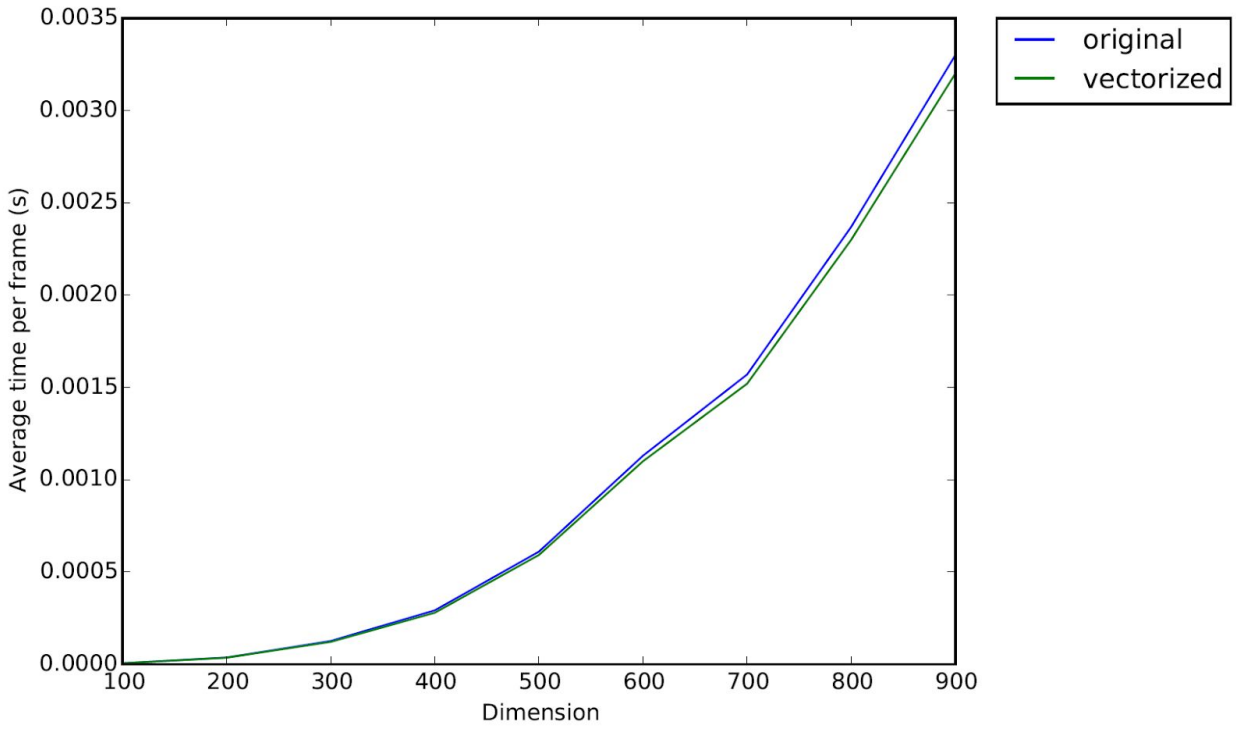
```
Central2D<Shallow2D, MinMod<float>>::compute_step           shallow            0.548s          0s              0s
Central2D<Shallow2D, MinMod<float>>::compute_fg_speeds      shallow            0.130s          0s              0s
[Outside any known module]                                  [Unknown]          0.017s          0s              0s
_IO_file_xsputn                                             libc-2.12.so       0.012s          0s              0s
_IO_fwrite                                                  libc-2.12.so       0.010s          0s              0s
Central2D<Shallow2D, MinMod<float>>::solution_check         shallow            0.005s          0s              0s
SimViz<Central2D<Shallow2D, MinMod<float>>>::write_frame  shallow              0.005s          0s              0s
Central2D<Shallow2D, MinMod<float>>::Central2D              shallow            0.002s          0s              0s
Central2D<Shallow2D, MinMod<float>>::run                    shallow            0.002s          0s              0s
std::array<float, (unsigned long)3>::operator[]             shallow            0.002s          0s              0s
Central2D<Shallow2D, MinMod<float>>::offset                 shallow            0.001s          0s              0s
do_lookup_x                                                 ld-2.12.so         0.001s          0s              0s
```

As can be seen, both the compute_step and compute_fg_speeds functions take about 0.1 seconds less time after vectorization is used. These numbers were not just coincidences or artifacts, but held up after repeated trials.

Table of overall speeds per frame vs. simulation grid size:

Grid size (dimension of one square side)	Average time per frame: with vectorization improvements (seconds)	Average time per frame: unoptimized original version (seconds)
100	5.17E-03	5.13E-03
200	3.55E-02	3.62E-02
300	1.21E-01	1.26E-01
400	2.79E-01	2.92E-01
500	5.92E-01	6.10E-01
600	1.10E+00	1.13E+00
700	1.52E+00	1.57E+00
800	2.30E+00	2.37E+00
900	3.20E+00	3.30E+00
1000	4.41E+00	4.54E+00

Chart of average speedup per frame in new code vs. simulation grid size:



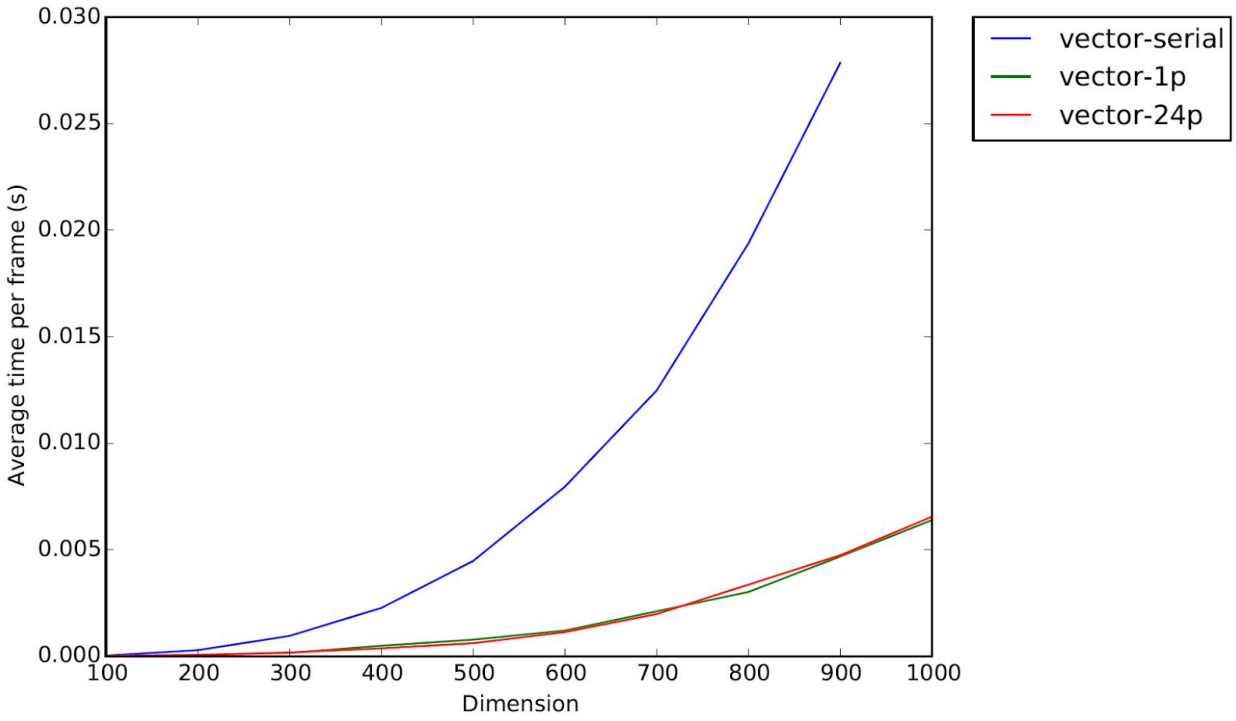
The #pragma ivdep directive was helpful in loops where vectorization was only being held back by the compiler's prior inability to assume that the memory locations being modified and the memory locations being read did not overlap. However, in many cases, such as the limited_derivs function, the loops still did not vectorize, because they would actually have become slower than the serial version due to memory access patterns that disrupted cache locality. Therefore, it would be necessary to change the structure of the code to enable access patterns making use of cache locality.

We are uncertain why the speedup was so marginal. We know that we did a get good amount of non-forced vectorization (check the cpp/ipo_out_vectorized_nonparallel.optrpt file). We are not sure where the bottleneck was after this, since as stated the amplxe-cl profiler is down.

The code with vectorization improvements only is in central2dVectorized.h.

Parallelization results:
We implemented the parallelization scheme described above in central2dwrapper.h, and tested its performance as both the number of processors and the size of the grid increased. The graph below shows the average computation time per frame as the grid size increases for varying numbers of processors. A timing report can be found in the cpp/timing subfolder. A plot was not included because the graph was extremely similar to the vector-1p / vector-24p timing plots below and therefore not terribly illuminating.

Combined:
We next combined the two approaches by vectorizing the parallel code. The graph below is a similar plot of average frame time as both problem size and number of processes changes, but the code being run on each processor is also vectorized. The line labeled “serial” represents running just the vectorized code in central2d.h, without central2dwrapper, while the line labeled “1p” represents running centrald2wrapper.h (and all the OpenMP code) with only a single processor available. This code is in the main (built by default by make) files, central2d.h and central2dwrapper.h. Timing plot:



More csv timing files can be found in the timing/ folder.