CS 5220 Shallow Water Project: Stage 1 Report

Group Members: Xinyi Wang, Markus Salasoo, Bangrui Chen
Group 12

1. Profiling

1.1 ICC compiler

In order to use icc to get the profiling report, we follow the instruction in [1] and get a detailed report ipo_out.optrpt. In the report, It checks the possibility that the icc compiler can vectorize each of the for loop. The report consists of the optimization report for driver.cc and central2d.h. Each optimization report again consists two parts: the first part is for the optimizable loops and the second part is for non-optimizable loops.

For the optimizable loops, we found that most of loops were not vectorized for two main reasons. The first one is that it cannot compute loop iteration count before executing the loop. The second reason is that vector dependence prevents vectorization. Two examples can be found on [2][3], which gives us an example of how to avoid these situations.

For the non-optimizable loops, the comment is nonstandard loop is not a vectorization candidate. Based on diagnostic report for icc compiler [4], there are four typical causes for this problem: 1. More than one exit point in the loop. A loop must have a single entry and a single exit point. Multiple exit points in a loop can generate this message. 2. This remark is also reported when C++ exception handling and OpenMP critical construct are used in a SIMD loop. 3. Third example demonstrates a case where compiler has no way to narrow down which function is passed as a function parameter. When this passed function is invoked inside the loop body, this vectorization diagnostic is generated. 4. Another case is documented which demonstrates how a loop body which is seen as a non-standard loop for vectorization can be made a suitable candidate for vectorization just by enabling the ANSI alias rule during compilation using compiler option -ansi-alias.

In summary, the report states that the icc compiler is not able to vectorize our existing code, but points out a direction for us to rewrite the code so that we can fully utilize the intel compiler.

1.2 VTune Amplifier

For the C++ version of the code, the VTune Amplifier tool gives insight into how our code performs. We use the tool in three steps.

1. Data Collection - run the `amplxe-cl` command with our compiled code. This will run slower than the standalone program, but this tool will record performance data valuable for identifying bottlenecks and store it in a new folder.

2. Generate Report - run the `amplxe-cl` command with the path to the new folder of performance data. The output of this tells us by method, where the heaviest computation occurs, ranked by CPU time.
3. Make Optimizations - use the report to identify areas with slow or inefficient computation and parallelize, vectorize, or re-write those sections with potential.

In section 2.2, we included some of the report output and how it led us to focus on optimizations in specific areas of the project.

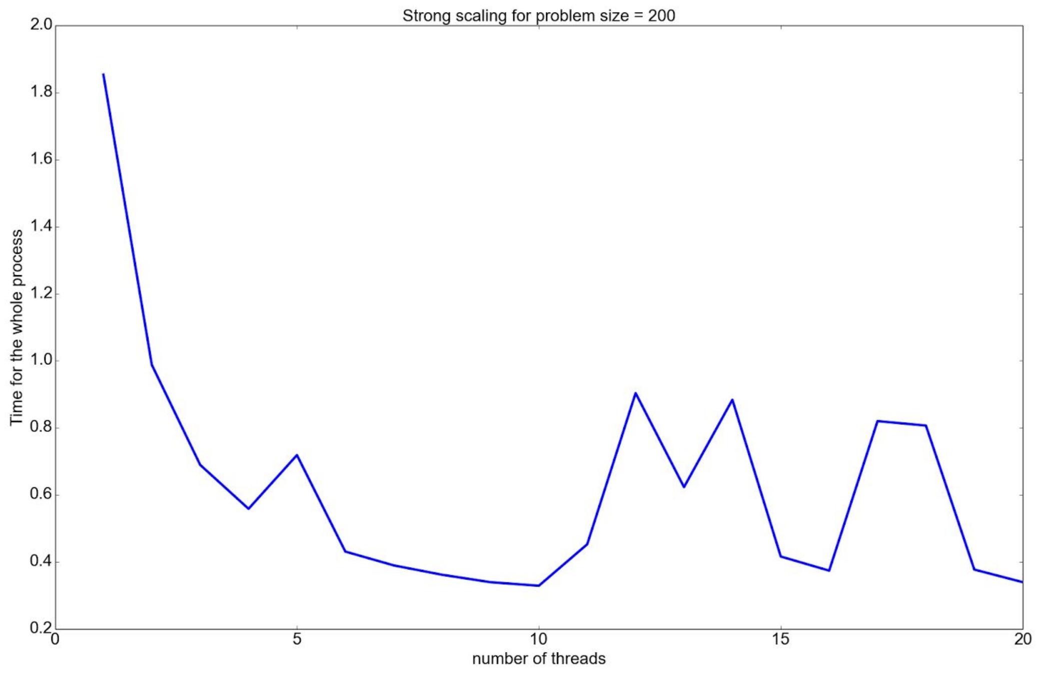
2. Parallelization

2.1 OpenMP

We use the openmp to parallel for loops in central2d.h file. The way we do it is the following:

```
#pragma omp parallel for
for (int iy = 0; iy < ny; ++iy)
    for (int ix = 0; ix < nx; ++ix)
        f(u(nghost+ix,nghost+iy), (ix+0.5)*dx, (iy+0.5)*dy);
```

We also use export OMP_NUM_THREADS=k to set the number of threads equals k in the pbs file. We did this for all the nested loops in central2d.h file and added omp_get_wtime at the beginning and end of the driver.cc file to compute the total amout of time we need in order to run the whoe process. The test result on the node is the following:



2.2 Grid Strategy / Tuning

In this shallow water simulation, we have a discretized grid representing our domain. Each cell/point is used in a calculation approximating a time-step with the Euler equations. There are two grids, but one is offset from the other by half step in between grid points. We plan to parallelize the computation with OpenMP as described in the previous section and here we describe how the algorithm will change to allow this.

There are three methods in central2d.h which have shown to take the longest time from our VTune Amplifier profiling report. The output is summarized with the top three functions:

Function	Module	CPU Time (s)
`Central2D<Shallow2D, MinMod<float>>::limited_derivs`	shallow	1.351
`Central2D<Shallow2D, MinMod<float>>::compute_step`	shallow	0.647
`Central2D<Shallow2D, MinMod<float>>::compute_fg_speeds`	shallow	0.227

`limited_derivs()`

Derivatives are calculated by finite difference methods, so any single point's future value is only dependent on the points adjacent to it. With many subdomains, it is possible to run these computations in parallel. Each subdomain will need to have its data offloaded, and that data needs to include an additional perimeter of "ghost cells." Points on the border of a subdomain are dependent on points in an adjacent subdomain. Communicating this data between parallel processes is inefficient. Instead, copy the necessary ghost cells into each subdomain data array before beginning the computation on that block. Only perform the finite difference computations on the non-ghost cells. Now we can compute a time-step for the entire domain more quickly.

After one timestep, we wait for each process to complete its computations on the subdomain so our entire domain is updated in memory. We can repeat for subsequent time-steps.

`compute_step()`

Blocking into subdomains will be applied here as well. The first loop computes a value for each grid entry (accumulating subtractions). The second, corrector loop does a computation with adjacent entries. The subdomains with ghost cells can handle this in parallel.

`compute_fg_speeds()`

This method computes flux values at each point and determines the maximum wave speed of the grid. This is parallelizable with the same subdomain blocking concept. The maximum value can be determined by computing a max on each subdomain, and then taking the maximum from those values.

It may be desirable to advance multiple timesteps with each subdomain to reduce the proportion of time spent synchronizing the master domain data structure. This may be possible by including more ghost cells (per subdomain) that reach farther into other subdomains. We will experiment with this after getting the base case working.

The second strategy we will experiment with is manually vectorizing some of the inner finite difference computation. From lecture and the previous project, we can see a speedup by manually specifying a loop to perform multiple updates simultaneously. In other words, performing the operation on a vector rather than a series of scalars. This could be applied inside each of the above loops.

It may be possible to apply SSE data structures and operation to these loops to gain an additional speedup. This may be accounted for with the vectorization flag at compile time, but we will verify this.

3. Tuning

The tuning aspect comes after implementing the grid blocking. We will need to choose subdomain sizes to take advantage of the hardware's cache sizes. Of course we will also benchmark the performance using block sizes around the theoretical optimum size because hardware is not 100% predictable.

Reference:

1. https://github.com/cornell-cs5220-f15/lecture/blob/master/2015-10-06/notes.md
2. https://software.intel.com/en-us/articles/fdiag15523
3. https://software.intel.com/en-us/articles/fdiag15344
4. https://software.intel.com/en-us/articles/cdiag15043
5. http://stackoverflow.com/questions/11095309/openmp-set-num-threads-is-not-working