CS 5220 Shallow Water Project: Final 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 instructions in [1] to 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, for example: **`assumed OUTPUT dependence between ** and **`** or **`assumed FLOW dependence between ** and **`**. 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. See section 2 in our report to find details.

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

---

2. Vectorization

Based on the icc report for vectorization, we changed our code to make it vectorizable. Here are some examples:

2.1 Cannot compute loop iteration count before executing the loop.

For example, the following loop in meshio.h is not vectorized:

```
for (int j = 0; j < sim.ysize(); ++j)
        for (int i = 0; i < sim.xsize(); ++i) {
                float uij = sim(i,j)[0];
                fwrite(&uij, sizeof(float), 1, fp);
        }
```

In order to fix this problem, you can just simply define the loop number outside the loop like the following:

```
const int y_size = sim.ysize();
const int x_size = sim.xsize();
for (int j = 0; j < y_size; ++j) {
     for (int i = 0; i < x_size; ++i) {
          fwrite(&(sim(i,j)[0]), sizeof(float), 1, fp);
     }
}
```

2.2 More than one exit point in the loop.

For example, the following loop in central2d.h has more than 1 exit:

```
for (int j = nghost; j < ny+nghost; ++j)
    for (int i = nghost; i < nx+nghost; ++i) {
         vec& uij = u(i,j);
         real h = uij[0];
         h_sum += h;
         hu_sum += uij[1];
         hv_sum += uij[2];
         hmax = max(h, hmax);
         hmin = min(h, hmin);
         // need to comment out if we want to vectorize this loop
         //assert( h > 0) ;
    }
```

In order to fix this problem, you can just comment out the assert command.

2.3 assumed OUTPUT dependence between ** and ** or assumed FLOW dependence between ** and **.

For this problem, you can solve it using **`#pragma ivdep`** or **`#pragma simd`**. The difference between these two is **`#pragma simd`** (!DIR$ SIMD for Fortran) behaves somewhat like a combination of **`#pragma vector always`** and **`#pragma ivdep`**, but is more powerful. The compiler does not try to assess whether vectorization is likely to lead to performance gain, it does not check for aliasing or dependencies that might cause incorrect results after vectorization, and it does not protect against illegal memory references. **`#pragma ivdep`** overrides potential dependencies, but the compiler still performs a dependency analysis, and will not vectorize if it finds a proven dependency that would affect results. With **`#pragma simd`**, the compiler does no such analysis, and tries to vectorize regardless. It is the programmer's responsibility to ensure that there are no backward dependencies that might impact correctness[5]. One example is:

```
 #pragma simd
for (int j = nghost; j < ny+nghost; ++j){
      for (int i = nghost; i < nx+nghost; ++i){
          u(i,j) = v(i-io,j-io);
      }
  }
```

After adding the #pragma simd, the vectorization report from the icc is the following:

```
LOOP BEGIN at central2d.h(387,5)
remark #15415: vectorization support: gather was generated for the variable _M_elems:
indirect access, 64bit indexed   [ central2d.h(389,22) ]
remark #15415: vectorization support: gather was generated for the variable _M_elems:
indirect access, 64bit indexed   [ central2d.h(389,22) ]
remark #15415: vectorization support: gather was generated for the variable _M_elems:
indirect access, 64bit indexed   [ central2d.h(389,22) ]
remark #15301: SIMD LOOP WAS VECTORIZED
remark #15458: masked indexed (or gather) loads: 3
remark #15459: masked indexed (or scatter) stores: 3
remark #15475: --- begin vector loop cost summary ---
remark #15476: scalar loop cost: 49
remark #15477: vector loop cost: 54.500
remark #15478: estimated potential speedup: 0.890
remark #15479: lightweight vector operations: 50
remark #15488: --- end vector loop cost summary ---
LOOP BEGIN at central2d.h(388,9)
remark #25460: No loop optimizations reported
LOOP END
LOOP END
```

Based on this vectorization report, we can see the potential speed up is 0.890, which is less than 1. This is mainly due to the indirect access to the data, which means the data is not stored contiguously. For our most time consuming loop, the vectorization leads to a better performance:

Function	**Speed Up**
limdiff	0.95
first for loop in compute_step	1.9
second for loop in compute_step	3.1
compute_fg_speeds	2.19

Since limited_devis mainly called limdiff function inside of its nested for loop, we vectorized the limdiff instead of limited_devis to get a better performance. However, since the speedup of limdiff is only 0.95, we didn’t vectorize it at the end.

A detailed reference provided by intel can be found in [6].

3. Parallelization

3.1 OpenMP

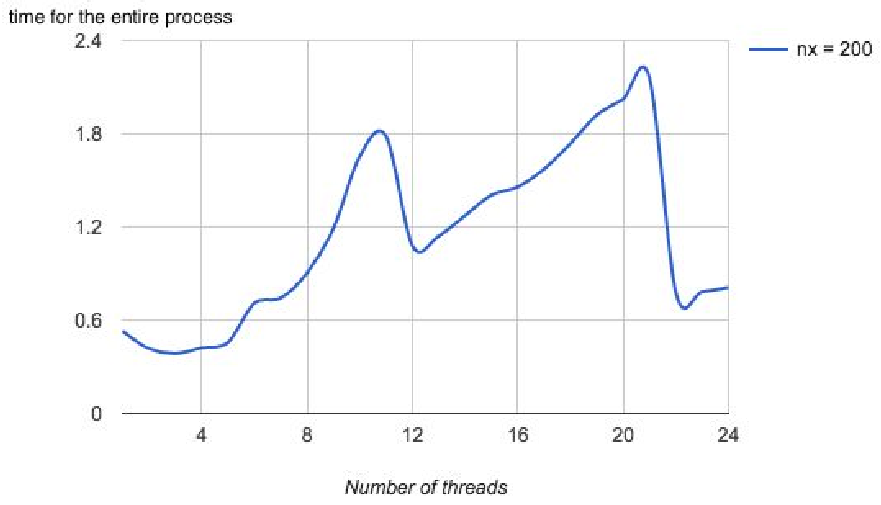
We used the OpenMP annotation **`#pragma omp parallel for`** to parallelize most of the for loops within the central2d.h file, and collapsing nested loops whenever possible using the **`collapse(x)`** clause. For instances where we have multiple for loops consecutively, we initialize the thread pool once (using **`#pragma omp parallel`**) and parallelize the subsequent loops using **`#pragma omp for`** to prevent us from having to instantiate the thread pool multiple times for each loop.

In the main **`run`** function, we start a thread pool with one thread for each subdomain, in order for subsequent calls to **`#pragma omp parallel`** within the subdomain threads to create new threads, (instead of limiting itself to the one thread that's running for that subdomain), we set the environment variable **`OMP_NESTED=true`** to allow for nested parallel regions.

We also experimented with using **`#pragma omp sections`** for portions of the code that could be executed in parallel (eg. when copying independent blocks of ghost cells), but since we could not nest **`#pragma omp parallel`** within a **`#pragma omp section`** block, and our experimental results showed that parallelizing the for loops gave us a much better speedup than executing the copying in parallel, and thus, we did not use sections in the final code.

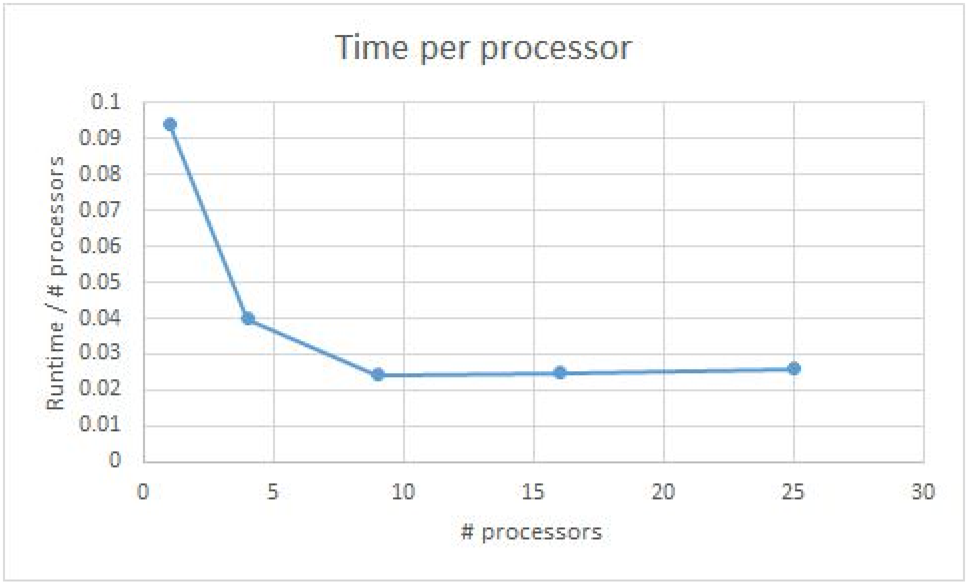
In main method in central2d.h is `run()`. Here we repeat a few steps in computing the next timestep. By blocking the grid into pieces, we can parallelize the computation in these iterations. The blocking details are in the next section, however we used three pragma keywords to help properly parallelize `run()`. In method `compute_fg_speeds()`, we iterate over the entire grid to determine a maximum value. Each thread will run that method to compute a maximum for that block, but then it will hit the **`#pragma omp barrier`**. This collects all threads and when all have arrived, they can continue. The following section has **`#pragma omp flush (maxCx, maxCy)`** which identifies common variables across the threads which will be written at the end of the block. We also include a **`#pragma omp critical`** section here to allow each thread to potentially modify the common variables holding the maximum value (in x and y). These pragma keywords allow the maximum wave speed to be properly identified and provided to each thread.

We also ran a strong scaling test for the number of threads that would be optimal for a problem size of 200 x 200. Since we define the number of threads in the main run loop to be equal to the number of subdomains that we have, the number of threads in this test only affects **`#pragma omp parallel`** sections where we do not explicitly state the number of threads to instantiate in the thread pool. We define the number of threads to be used using the environment variable **`OMP_NUM_THREADS=k`**. We did this for all the nested loops in **`central2d.h`** and added **`omp_get_wtime`** at the beginning and end of **`driver.cc`** file to compute the total amount of time we need in order to run the entire process. The test result on the node is the following:



From the plot, we can see a decrease in performance (ie. an increase in time taken for the entire process) after 3 threads -- this may be due to the fact that for several of the loops that we had attempted to parallelize, the problem size was so small that the gains in parallelizing the computation was outweighed by the cost of starting and synchronizing the thread pool, and thus for those loops we might have been better off leaving them in serial.

In a weak scaling study, we kept the problem size constant per thread, keeping the workload constant on every thread. We used a block size of 50 and thread counts 1, 4, 9, 16, 25 (due to a bug which is described in the final section). This means we use square grids with side length 50, 100, 150, 200, and 250.



From the plot, we see the runtime divided by the number of processors to decrease and seemingly approach a constant. Unfortunately this means runtime does not stay constant as workload increases in direct proportion to number of processors. We believe there is a bottleneck when copying the grid to blocks and vice versa after one full timestep iteration. This could be improved by only communicating ghost cells to other processors and avoiding the expensive copying so often. MPI seems better suited to support this since we would be able to pass data directly from one processor to another.

3.2 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. The starter code uses a half step scheme to compute each successive timestep. 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.

The first thing to realize in central2d.h is that the grid is stored as a class variable. Each method has access to it and makes changes to its values. This must be changed for parallelization. One solution is to lock and unlock the data structure, but that slows down a thread's potential speed. The smart solution is to block the grid into smaller grids and pass them in by reference to the method operating on it. This means we need to re-write many of the helper methods in central2d.h because they currently operate on the common class variable.

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 centered at the non-ghost cells. Now we can compute a time-step for the entire domain more quickly. We can repeat for subsequent time-steps.

This method operates on many variables which need to be passed in by reference. They are: u, fluxes in x and y, ux, uy, fx, and gy.

`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.

This method needs to be re-written so loops and calculations are expressed in terms of arguments (passed in by reference). Variables in play here are: u, fluxes in x and y, ux, uy, fx, and gy.

`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. This is accomplished with a few OpenMP keywords, described in the previous section.

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. Our blocking scheme has a bug where specific block sizes for some domains cause an assertion to fail (h>0). If we had more time, we could pursue this option further.

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 and we have described how we implemented it with keywords in a previous section.

4. 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. Unfortunately there are a couple of bugs related to block size in the code so we did not go through with this tuning.

5. Known Bugs

We have two known bugs in our final code which are both related to blocking.

We can argue the first bug exists by design. It is a non-enforced requirement to have the grid size be a multiple of the block size. For simplicity this base case was implemented first and we decided to spend our remaining time fixing other bugs before this one because this constraint is reasonable.

The second bug is at large. It seems if we use a total number of blocks that is a power of two, the simulation will run fine. However using 9 blocks for example will fail the wave height assertion in `solution_check()`. The computations can continue if we comment out the assertion and we can still observe speedups.

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. https://software.intel.com/en-us/articles/requirements-for-vectorizing-loops-with-pragma-simd
6. https://software.intel.com/sites/default/files/m/4/8/8/2/a/31848-CompilerAutovectorizationGuide.pdf
7. http://stackoverflow.com/questions/11095309/openmp-set-num-threads-is-not-working