CS 5220: Homework 2

Group 19: Robert Carson (rac428), Robert Chiodi (rmc298), Sam Tung (sat83)

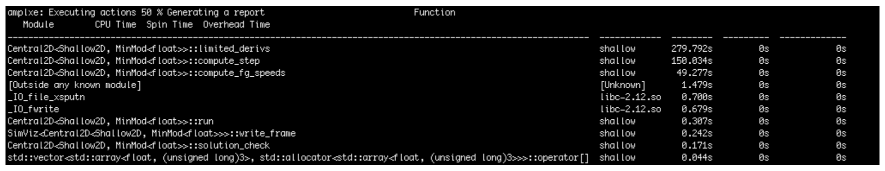
October 29, 2015

1 Initial Profiling

In order to perform initial profiling of the code before any improvements are made, Intel's VTUNE was used via the terminal command line on Totient. In an attempt to get the most accurate results (taken with a large sample size), we decided to gather data while running the large wave simulation, invoked with the command `make big`. Information collected from VTUNE's "advanced-hotspots" option is used in our analysis.

1.1 Whole Program - Advanced Hotspots

First, hotspots in the entire program were examined in order to determine where our efforts should be directed. The time taken in the top 10 most time consuming functions can be seen below in Figure 1. Of these, it is clear that most of our optimization efforts should be directed to the functions `limited_derivs`, `compute_step`, and `compute_fg_speeds`. These functions were then examined individually, once again using VTUNE on our advanced-hotspots collection.

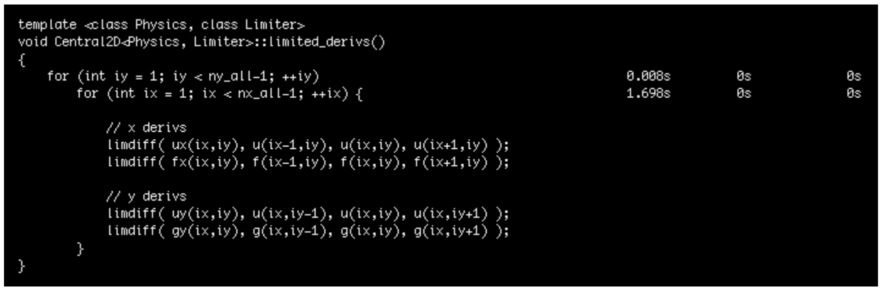


```
amplxe: Executing actions 50 % Generating a report                          Function
   Module        CPU Time  Spin Time  Overhead Time
------------------------------------------------------------------------------------------------------------------  ------------  --------  ---------  ------------
Central2D<Shallow2D, MinMod<float>>::limited_derivs                                                         shallow       279.792s        0s           0s
Central2D<Shallow2D, MinMod<float>>::compute_step                                                           shallow       150.034s        0s           0s
Central2D<Shallow2D, MinMod<float>>::compute_fg_speeds                                                      shallow        49.277s        0s           0s
[Outside any known module]                                                                                  [Unknown]       1.479s        0s           0s
_IO_file_xsputn                                                                                             libc-2.12.so    0.700s        0s           0s
_IO_fwrite                                                                                                  libc-2.12.so    0.679s        0s           0s
Central2D<Shallow2D, MinMod<float>>::run                                                                    shallow         0.307s        0s           0s
SimViz<Central2D<Shallow2D, MinMod<float>>>::write_frame                                                    shallow         0.242s        0s           0s
Central2D<Shallow2D, MinMod<float>>::solution_check                                                         shallow         0.171s        0s           0s
std::vector<std::array<float, (unsigned long)3>, std::allocator<std::array<float, (unsigned long)3>>>::operator[] shallow         0.044s        0s           0s
```

Figure 1: Top 10 most time consuming functions in the wave simulation. Generated using Intel's VTUNE on Totient.

1.2 **`limited_derivs`** - Advanced Hotspots

The function `limited_deriv` is used to calculate the fluxes into and out of each cell in order to advance to the next time step. This involves a three point computational stencil in each direction and loops through the entire domain interior (the whole domain except for those where boundary conditions are applied). Each point requires `du.size()` $\times$ 9 floating point operations as well as `du.size()` $\times 2$ calls to the intrinsic function `min`. Sadly, the hotspot analysis on the `limited_deriv` function, shown in Figure 2, does not give any hints on possible optimizations or bottle necks.



```
template <class Physics, class Limiter>
void Central2D<Physics, Limiter>::limited_derivs()
{
    for (int iy = 1; iy < ny_all-1; ++iy)                                    0.008s      0s        0s
        for (int ix = 1; ix < nx_all-1; ++ix) {                              1.698s      0s        0s

            // x derivs
            limdiff( ux(ix,iy), u(ix-1,iy), u(ix,iy), u(ix+1,iy) );
            limdiff( fx(ix,iy), f(ix-1,iy), f(ix,iy), f(ix+1,iy) );

            // y derivs
            limdiff( uy(ix,iy), u(ix,iy-1), u(ix,iy), u(ix,iy+1) );
            limdiff( gy(ix,iy), g(ix,iy-1), g(ix,iy), g(ix,iy+1) );
        }
}
```

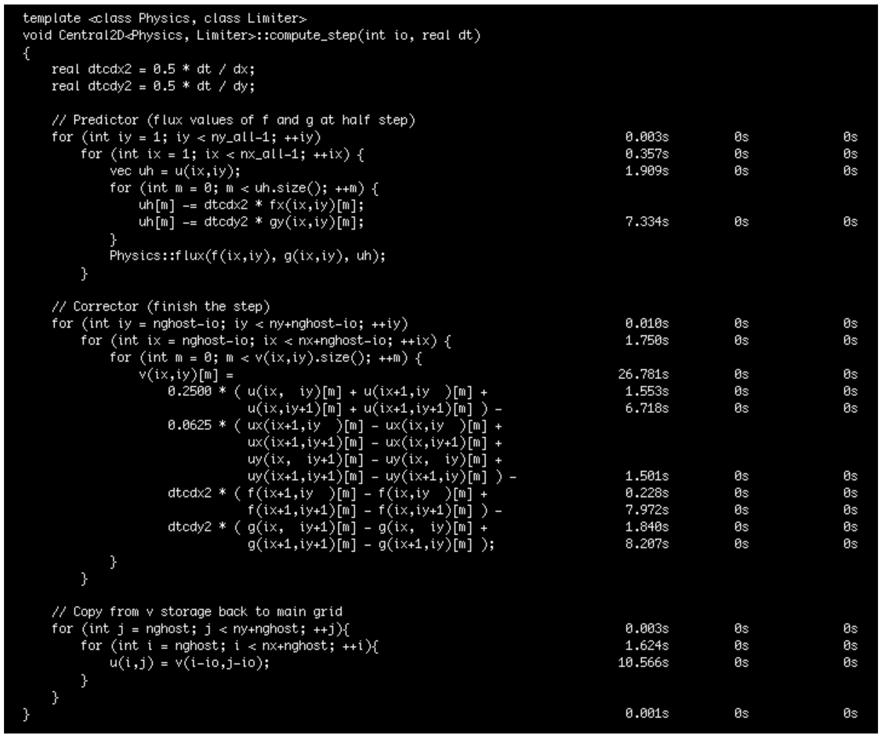
Figure 2: Time taken to perform each loop present in `limited_derivs`, recorded in core-seconds.

1.3 **`compute_step`** - Advanced Hotspots

The purpose of `compute_step` is to update the wave equation to the next time step using a predictor-corrector method. First, the fluxes are calculated in the prediction. Next, the corrector step uses the predicted fluxes, the differences in velocities, and the current velocities, to advance to the next time state. Luckily, VTUNE's report is more helpful than in the previous case, and provides extensive timings for this function, shown in Figure 3. The calculation in the corrector step can be seen as the most expensive cost of the function. It is important to note, however, that the predictor step and copying of the solution to the `u` array sum to half of the function's cost.

1.4 **`compute_fg_speeds`** - Advanced Hotspots

The function of `compute_fg_speeds` has two primary responsibilities: to update the cell centered fluxes, `f` and `g`, and to calculate the maximum speed in the domain, allowing dynamic adjustment of the time step in order to satisfy the CFL condition and ensure numerical stability. The timing data for this function can be seen in Figure 4. While the most time consuming portion of the code is most likely the calculation of the fluxes and wave speed (both of which are in the `Shallow2d` structure), the calls to the intrinsic function `max` also represent a non-trivial amount of time.



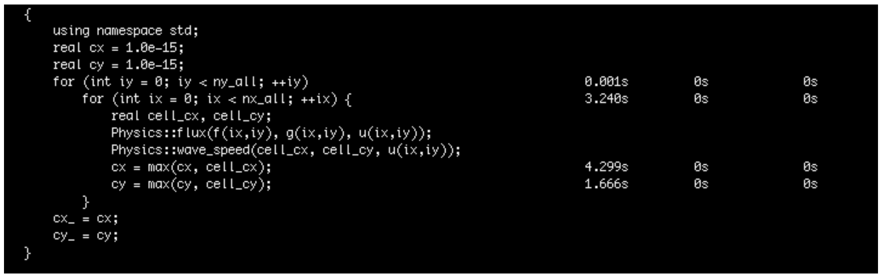
```
template <class Physics, class Limiter>
void Central2D<Physics, Limiter>::compute_step(int io, real dt)
{
    real dtcdx2 = 0.5 * dt / dx;
    real dtcdy2 = 0.5 * dt / dy;

    // Predictor (flux values of f and g at half step)
    for (int iy = 1; iy < ny_all-1; ++iy)                              0.003s     0s      0s
        for (int ix = 1; ix < nx_all-1; ++ix) {                        0.357s     0s      0s
            vec uh = u(ix,iy);                                         1.909s     0s      0s
            for (int m = 0; m < uh.size(); ++m) {
                uh[m] -= dtcdx2 * fx(ix,iy)[m];
                uh[m] -= dtcdy2 * gy(ix,iy)[m];                        7.334s     0s      0s
            }
            Physics::flux(f(ix,iy), g(ix,iy), uh);
        }

    // Corrector (finish the step)
    for (int iy = nghost-io; iy < ny+nghost-io; ++iy)                  0.010s     0s      0s
        for (int ix = nghost-io; ix < nx+nghost-io; ++ix) {            1.750s     0s      0s
            for (int m = 0; m < v(ix,iy).size(); ++m) {
                v(ix,iy)[m] =                                         26.781s     0s      0s
                    0.2500 * ( u(ix,  iy)[m] + u(ix+1,iy  )[m] +       1.553s     0s      0s
                               u(ix,iy+1)[m] + u(ix+1,iy+1)[m] ) -     6.718s     0s      0s
                    0.0625 * ( ux(ix+1,iy  )[m] - ux(ix,iy  )[m] +
                               ux(ix+1,iy+1)[m] - ux(ix,iy+1)[m] +
                               uy(ix,  iy+1)[m] - uy(ix,  iy)[m] +
                               uy(ix+1,iy+1)[m] - uy(ix+1,iy)[m] ) -   1.501s     0s      0s
                    dtcdx2 * ( f(ix+1,iy  )[m] - f(ix,iy  )[m] +       0.228s     0s      0s
                               f(ix+1,iy+1)[m] - f(ix,iy+1)[m] ) -     7.972s     0s      0s
                    dtcdy2 * ( g(ix,  iy+1)[m] - g(ix,  iy)[m] +       1.840s     0s      0s
                               g(ix+1,iy+1)[m] - g(ix+1,iy)[m] );      8.207s     0s      0s
            }
        }

    // Copy from v storage back to main grid
    for (int j = nghost; j < ny+nghost; ++j){                          0.003s     0s      0s
        for (int i = nghost; i < nx+nghost; ++i){                      1.624s     0s      0s
            u(i,j) = v(i-io,j-io);                                    10.566s     0s      0s
        }
    }
}                                                                      0.001s     0s      0s
```

Figure 3: Time taken to perform each loop present in `compute_step`, recorded in core-seconds.



```
{
    using namespace std;
    real cx = 1.0e-15;
    real cy = 1.0e-15;
    for (int iy = 0; iy < ny_all; ++iy)                           0.001s     0s      0s
        for (int ix = 0; ix < nx_all; ++ix) {                     3.240s     0s      0s
            real cell_cx, cell_cy;
            Physics::flux(f(ix,iy), g(ix,iy), u(ix,iy));
            Physics::wave_speed(cell_cx, cell_cy, u(ix,iy));
            cx = max(cx, cell_cx);                                4.299s     0s      0s
            cy = max(cy, cell_cy);                                1.666s     0s      0s
        }
    cx_ = cx;
    cy_ = cy;
}
```

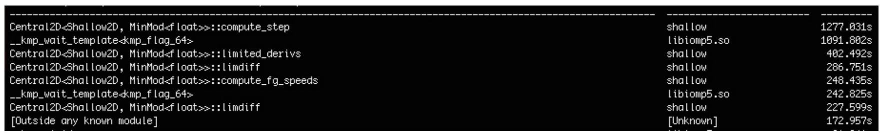
Figure 4: Time taken to perform each loop present in `compute_fg_speeds`, recorded in core-seconds.

2 Parallelization

While we are aware that the optimal programming of this code will not wholly be due to parallelization by OpenMP, it is most certainly necessary. Furthermore, future tuning and optimization of the code may change depending on whether the code is run in serial or parallel. For these reasons, we first decided to parallelize the code and compare its performance against the initial, serial code.

2.1 Naive Implementation of OpenMP

First, a naive implementation of OpenMP using pragmas was used. This involved placing `#pragma omp parallel for` in front of the start of each for loop in the functions `init`, `apply_periodic`, `compute_fg_speeds`, `limited_derivs`, and `compute_step`. This proved to improve performance significantly when run on one node of Totient (24 threads), as seen in Figure 5. It should be noted that the time shown is the total processor time spent and not the per processor time.



```
-----------------------------------------------------------------------------------------------------------  ------------------------  ---------
Central2D<Shallow2D, MinMod<float>>::compute_step                                                            shallow                   1277.031s
__kmp_wait_template<kmp_flag_64>                                                                             libiomp5.so               1091.802s
Central2D<Shallow2D, MinMod<float>>::limited_derivs                                                          shallow                    402.492s
Central2D<Shallow2D, MinMod<float>>::limdiff                                                                 shallow                    286.751s
Central2D<Shallow2D, MinMod<float>>::compute_fg_speeds                                                       shallow                    248.435s
__kmp_wait_template<kmp_flag_64>                                                                             libiomp5.so                242.825s
Central2D<Shallow2D, MinMod<float>>::limdiff                                                                 shallow                    227.599s
[Outside any known module]                                                                                   [Unknown]                  172.957s
```

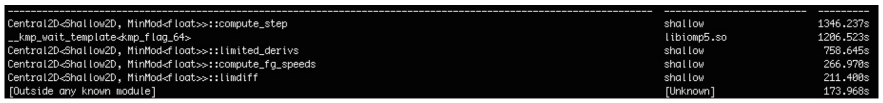
Figure 5: Timing of entire simulation using naive implementation of OpenMP parallelized for-loops. Generated using Intel's VTUNE on Totient and recorded in core-seconds.

2.2 OpenMP for with `collapse`

"OpenMP parallel for" also has the `collapse` option, which essentially informs OpenMP how many nested loops are present. By knowing this, OpenMP is able to section both loops to run on multiple processors, creating blocks for each processor to work on. Results when using `collapse` can be seen in Figure 6. It should be noted that the time shown is the total processor time spent and not the per processor time. It was found that using the `collapse` option actually led to worse performance. We believe this is due to the creation of blocks, which will limit the amount of memory accesses each core has that is of unit stride. When only sectioning the for loops based on the vertical (y) direction, each processor gets a $n \times nx$ block of the array, where n is some number of rows (vertical lines of computational cells) in the domain. This increases memory access locality, which in turn increases cache hits.

2.3 OpenMP thread creation limited:

The constant creation of threads creates a large overhead cost that negatively affects the performance of the program. Therefore, the fewer times that a pool of threads needs to be created the better. In the previous examples, the thread pools were created each time a parallel for loop was called. In order to reduce this costly operation, a pool of threads was only created within the main run function as shown in the below code section.

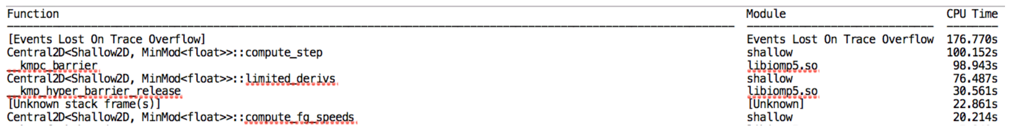


```
-----------------------------------------------------------------------------------------  -------------------------  ---------
Central2D<Shallow2D, MinMod<float>>::compute_step                                          shallow                    1346.237s
__kmp_wait_template<kmp_flag_64>                                                           libiomp5.so                1206.523s
Central2D<Shallow2D, MinMod<float>>::limited_derivs                                        shallow                     758.645s
Central2D<Shallow2D, MinMod<float>>::compute_fg_speeds                                     shallow                     266.970s
Central2D<Shallow2D, MinMod<float>>::limdiff                                               shallow                     211.400s
[Outside any known module]                                                                 [Unknown]                   173.968s
```

Figure 6: Timing of entire simulation using implementation of OpenMP parallelized for-loops with `collapse` option. Generated using Intel's VTUNE on Totient and recorded in core-seconds.

```
for (int io = 0; io < 2; ++io) {
        real cx, cy;
        #pragma omp parallel
        {
        apply_periodic();
        compute_fg_speeds(cx, cy);
        limited_derivs();
        #pragma omp single
            {
        if (io == 0) {
            dt = cfl / std::max(cx/dx, cy/dy);
            if (t + 2*dt >= tfinal) {
                dt = (tfinal-t)/2;
                done = true;
            }
        }
            }
        compute_step(io, dt);
        #pragma omp single
        t += dt;
        }
```

Sections of the code that only required one thread to execute took advantage of the omp single pragma command which tells only one thread to run that section of code while the rest wait for it to finish. Inside each of the computation calls the regular omp for pragma command is used to parallelize the various for loops being run. The performance of this operation for the top five costliest function calls when run on the big simulation can be found in Figure 7.
It should also be noted that these results do include the effects of attempting to force the compiler to vectorize several loops by replacing the min and max calls with an equivalent if statement min and max code. Also from Figure 7, it can be seen that the program still spends a large amount of time in block operations. Based on Figure 5, it appears that the main cost of thread creation is no longer one of the top performance issues with the code. Now, the internal barrier times is of much more concern.

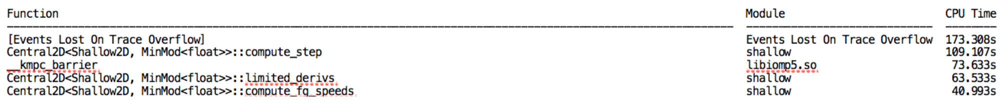


```
Function                                                          Module                      CPU Time
----------------------------------------------------------------  --------------------------  --------
[Events Lost On Trace Overflow]                                   Events Lost On Trace Overflow  176.770s
Central2D<Shallow2D, MinMod<float>>::compute_step                 shallow                     100.152s
__kmpc_barrier                                                    libiomp5.so                  98.943s
Central2D<Shallow2D, MinMod<float>>::limited_derivs               shallow                      76.487s
__kmp_hyper_barrier_release                                       libiomp5.so                  30.561s
[Unknown stack frame(s)]                                          [Unknown]                    22.861s
Central2D<Shallow2D, MinMod<float>>::compute_fg_speeds            shallow                      20.214s
```

Figure 7: Timing of five costliest functions using fewer thread creation events. Generated using Intel's VTUNE on Totient.

2.4 OpenMP nowait implementation:

When using the `omp for` pragma command, an implicit barrier is placed at the end of the loops. Due to the nature of this program, it is possible in several places to be able to assume that a barrier is not needed, and therefore the `nowait` pragma command can be used. The nowait command could be safely used in the `compute_fg_speeds` and `limited_derivs` functions. Then it could be partially used in the `compute_step` function once the predictor step was calculated. If the nowait command was used for the predictor loop, synchronization issues are possible when a thread is able to go onto the corrector step while other threads are still in the predictor step. The performance of this implementation for the top five costliest function calls when run on the "big" simulation can be found in Figure 8



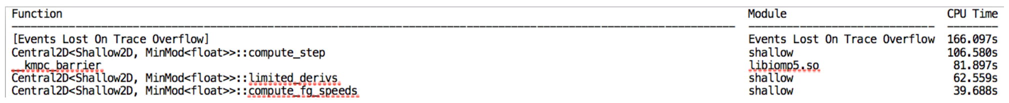
```
Function                                                          Module                      CPU Time
----------------------------------------------------------------  --------------------------  --------
[Events Lost On Trace Overflow]                                   Events Lost On Trace Overflow  173.308s
Central2D<Shallow2D, MinMod<float>>::compute_step                 shallow                     109.107s
__kmpc_barrier                                                    libiomp5.so                  73.633s
Central2D<Shallow2D, MinMod<float>>::limited_derivs               shallow                      63.533s
Central2D<Shallow2D, MinMod<float>>::compute_fg_speeds            shallow                      40.993s
```

Figure 8: Timing of five costliest functions using `nowait` pragma command. Generated using Intel's VTUNE on Totient.

It can be seen by comparing Figure 7 and Figure 8 that the overall time spent in the barrier function call has gone down drastically by including the nowait call.

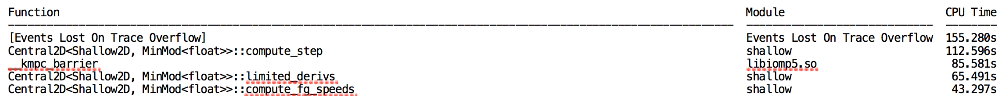
2.5 Thread Affinity:

OpenMP 4.0 offers the ability to describe the ordering of the threads among the CPUs/MICs. Threads can be placed near each other on the same CPU using the OpenMP pragma command `proc_bind(close)`. The benefit of doing this is that amount of time for synchronization between threads should decrease. The downside of this is that the available cache and bandwidth per thread will decrease. Another option available distributes the threads more evenly throughout the available CPUs/MIC, and it can be turned on using the OpenMP pragma command `proc_bind(spread)`. It should lead to the opposite effects of the close command. The thread affinity was implemented on the code base that used the `nowait` pragma command. The performance of this implementation for the top five costliest function calls when run on the big simulation can be found in Figure 9 and Figure 10 for the `proc_bind(close)` pragma command. Due to system traffic, timing can vary across multiple runs. For this reason, two figures were shown so some measure of variance could be judged. The results for the `proc_bind(spread)` command can be found in Figure 11.



```
Function                                                                  Module                       CPU Time
------------------------------------------------------------------------  ---------------------------  --------
[Events Lost On Trace Overflow]                                           Events Lost On Trace Overflow  166.097s
Central2D<Shallow2D, MinMod<float>>::compute_step                         shallow                      106.580s
__kmpc_barrier                                                            libiomp5.so                   81.897s
Central2D<Shallow2D, MinMod<float>>::limited_derivs                       shallow                       62.559s
Central2D<Shallow2D, MinMod<float>>::compute_fg_speeds                    shallow                       39.688s
```

Figure 9: Timing of five costliest functions using `proc_bind(close)` pragma command for run 1. Generated using Intel's VTUNE on Totient.



```
Function                                                                  Module                       CPU Time
------------------------------------------------------------------------  ---------------------------  --------
[Events Lost On Trace Overflow]                                           Events Lost On Trace Overflow  155.280s
Central2D<Shallow2D, MinMod<float>>::compute_step                         shallow                      112.596s
__kmpc_barrier                                                            libiomp5.so                   85.581s
Central2D<Shallow2D, MinMod<float>>::limited_derivs                       shallow                       65.491s
Central2D<Shallow2D, MinMod<float>>::compute_fg_speeds                    shallow                       43.297s
```

Figure 10: Timing of five costliest functions using `proc_bind(close)` pragma command for run 2. Generated using Intel's VTUNE on Totient.

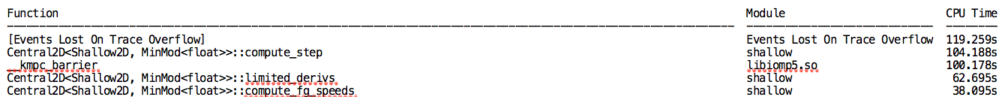
Overall, both thread affinity options have shown to lead to decreased performance. Therefore, they will not be used in the production code.

3 Analysis

The current implementation, while offering a sizable speed up over the strictly serial version, still leads much to be desired. Since it currently depends on the `parallel for` command to run with OpenMP automating certain commands, barriers are automatically placed into the code that limits how each processor accesses the memory of the variables. Further improvements can be made to the code by eliminating these blocking structures where allowed and only syncing the data when the need arises in order to reduce the amount of overhead costs that occur from this type of synchronization. One such place where this could be beneficial is in the `compute_step` function. Specific structures could be used in order to ensure the predictor step is done for edge cases before the corrector loop begins without requiring a barrier. To do this, a more rigorous implementation of OpenMP will need to be done, which mimics more of a distributed memory parallel code. Blocks will be made of the domain, where each subdomain has ghost cells and is unaware of what happens outside of itself (except through the ghost cells). This gives more control and limits necessary communication.

3.1 Strong and Weak Scaling

The current version of the code can be analyzed for its strong and weak scaling capabilities. The strong scaling will be tested using the "big" wave test case with 1000 points in each direction and 100 time steps between frames. The results can be seen in Table 1. This is tested on the main Xeon chips. With the highest number of threads tested being 16, inter-node communication costs were avoided. VTUNE analysis was also not collected in order to remove any additional time the collection process takes. In Table 1, strong scaling is defined by Eq. 1 and strong scaling efficiency is defined by Eq. 2.



```
Function                                                                                  Module                       CPU Time
----------------------------------------------------------------------------------------  ---------------------------  --------
[Events Lost On Trace Overflow]                                                           Events Lost On Trace Overflow  119.259s
Central2D<Shallow2D, MinMod<float>>::compute_step                                         shallow                      104.188s
__kmpc_barrier                                                                            libiomp5.so                  100.178s
Central2D<Shallow2D, MinMod<float>>::limited_derivs                                       shallow                       62.695s
Central2D<Shallow2D, MinMod<float>>::compute_fg_speeds                                    shallow                       38.095s
```

Figure 11: Timing of five costliest functions using `proc_bind(spread)` pragma command. Generated using Intel's VTUNE on Totient.

$$\text{Strong Scaling} = \frac{t_{\text{serial}}}{t_{\text{parallel}}} \tag{1}$$

$$\text{Strong Scaling Efficiency} = \frac{t_{\text{serial}}}{t_{\text{parallel}}}\frac{1}{p} \tag{2}$$

where t_{serial} and t_{parallel} are the times taken when running in serial or parallel, respectively, and p is the number of processors. The strong scaling efficiency is essentially what percentage of perfect linear scaling is actually achieved. This is plotted in Figure 12.

Threads	Total Time (seconds)	Strong Scaling	Strong Scaling Efficiency
1	539	1.0	100%
2	288	1.87	93.5%
4	218	2.47	61.7%
8	198	2.72	34.0%
16	188	2.87	17.9%

Table 1: Strong scaling for wave simulation with 1000 mesh points in each direction and 100 timesteps between frames.

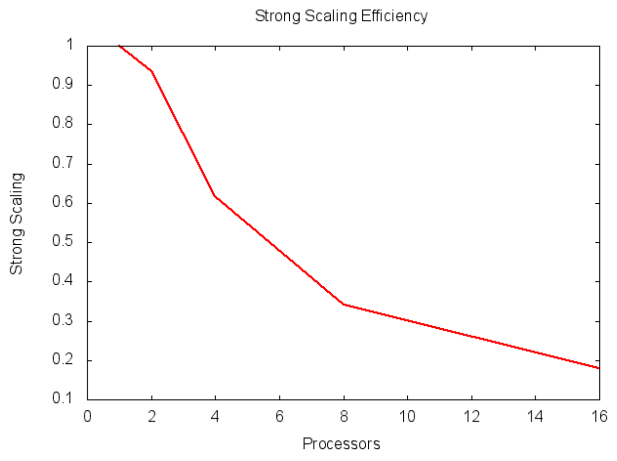


Figure 12: Plot of strong scaling efficiency, representing the data shown in Table 1.

For weak scaling, three simulations were performed, scaling the number of processors equally with the number of cells. The results can be seen in Table 2. Here, weak scaling is defined by Eq. 3. It was decided to double the side length for each simulation in order to perform the weak scaling test, meaning the number of processors needed to be quadrupled for each run. A plot of the weak scaling can also be seen in Figure 13. A more thorough weak scaling test is planned for the future.

$$\text{Weak Scaling} = \frac{t_{\text{serial}}(n(p))}{t_{\text{parallel}}(n(p), p)} \tag{3}$$

Threads	Cells Per Side	Total Time (seconds)	Weak Scaling
1	200	4.0	1.00
4	400	8.3	0.485
16	800	73.5	0.0544

Table 2: Weak scaling for wave simulation with 1000 mesh points in each direction and 100 timesteps between frames.

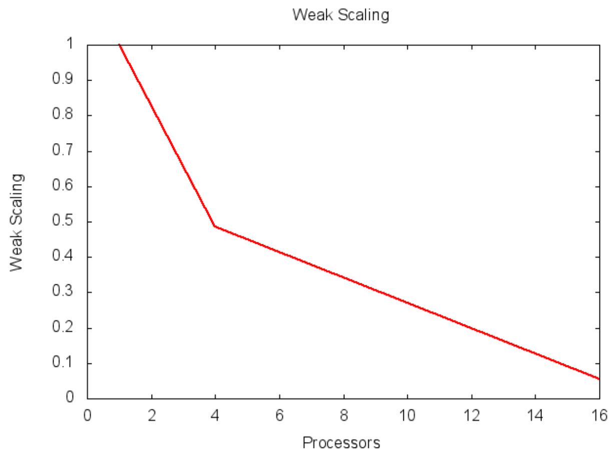


Figure 13: Plot of weak scaling, representing the data shown in Table 2.

We believe the weak scaling performance is very poor due to our implementation of OpenMP. Weak scaling usually tests the importance of communication on distributed memory systems, since each processor should be performing the same amount of algorithmic work as it is in the serial case. The main difference is the size of data being communicated is largely increased with the scaling in problem size. Since this is a shared memory code, we believe this indicates poor memory handling, leading to many cache misses. This is especially evident in the largest case, where we believe the problem size exceeded the cache size, causes very poor memory use. This could be fixed by a blocking scheme, where the domain is decomposed into subdomains, in order to increase memory locality, reducing the number of cache misses substantially. Another possible reason for the poor performance of the 16 thread case is an increase in time for each communication, due to the threads existing on two separate chips.

The scaling can be represented theoretically by creating an equation for the parallel time taken in the code. This would be primarily made up of two parts: the time to perform the floating point operations, and the portion spent communicating and synchronizing. In this code, since it uses shared memory, the only communications required involves OpenMP barriers and informing each processor of their section of the "for loops". These communications will be incorporated into the relation using the variable t_c. Predicting the value of t_c consistently is difficult due to uncertainty in how long processors will be required to wait at a barrier. Through running simulations with various parameters, it may be possible to get a rough estimate of the communication time per processor, which could then be used in this model to approximate the scaling for a specific simulation.

The expression for time taken to execute the simulation in parallel can be represented by Eq. 4.

$$t_{\text{parallel}} = \frac{t_{\text{serial}}}{p} + pt_c \tag{4}$$

This equation assumes that the serial work will be evenly distributed among all processors, leading to the serial work being completed in t_{serial}/p time. The cost of parallelizing the code per processor, in the form of communication, barriers, and synchronization, is represented by t_c. With an expression for the parallel time, it is now possible to create a rough approximation for scaling, shown in Eq. 5. The difference in strong and weak scaling appears in the values used for t_{serial} and t_c.

$$\text{Scaling} \approx \frac{t_{\text{serial}}}{t_{\text{parallel}}} = \frac{t_{\text{serial}}}{\frac{t_{\text{serial}}}{p} + pt_c} = \left(\frac{1}{p} + p\frac{t_c}{t_{\text{serial}}}\right)^{-1} \tag{5}$$

4 Post Initial Report Work:

4.1 Swapping Programming Languages:

Due to the problems that the layer of abstraction within C++ code base gave the compiler, it was decided it would be best to transition over to Professor Bindel's C version of the code. The initial results from that swap resulted in a set of code that was performing at the same level of performance as the parallelized C++ code when using 16+ threads. Therefore, this code base will provide an appropriate environment to see what effects various forms of parallelization can have on the performance on the code.

4.2 Vectorization and other optimizations of the code base:

While the C code is rather vectorized, it still had a couple of areas of improvement that could be made to allow the compiler to better vectorize the various loops in the code. First off, the various techniques used in the matmul assignment can be introduced here including aligning the memory to a 32 byte boundary and the use of the `# pragma vector aligned` command. The `_mm_malloc()` and `_mm_free()` commands were used in place of the `malloc` and `free` commands for the creation of the `sim` and `sim->u` variables. This allowed the compiler to know that these variables would be aligned to a 32 byte boundary. Also since several other variables are based upon the `sim->u`, the compiler recognizes that these also should be aligned to a 32 byte boundary as well. Next, the `# pragma vector aligned` is used at each for loop used in both the stepper.c and shallow2d.c files. Since all the vector variables with initial data have been aligned, the various calculations done inside the loop will not lead to a segmentation fault. If this was not the case, the `__attribute__((aligned( 32 )))` would want to be used during the declaration of those variables in order to ensure those variables were aligned. It should also be noted that if the Xeon Phi chips were being used than the vectors could be aligned to a 64 byte boundary instead, since the chip set supports the AVX512 instruction set. Finally, the following Intel compiler optimization flags were used to ensure we are taking full advantage of the Xeon architecture during our vectorization attempts:`-O3 -axcore-avx2 -march=core-avx2 -unroll-aggressive -m64 -ipo -no-prec-div -ansi-alias`.

Next, several times a variable is created inside a for loop. This constant creation of these variables could lead to a slow down of the code. Therefore, if the variable being created did not call for the use of the const or restrict keywords it was initially created outside of the loop(s). This move was done for both the stepper.c and shallow2d.c file. The results when comparing the baseline C code, and the further optimized code can be seen in Table 3 for a 1000x1000 size mesh. It can be seen while the speed up is minor it is a nice improvement on an already well optimized code base.

Simulation Name	Time(s)	Speed up(%)
Baseline Code	59.8	100
Aligned Code	55.9	107

Table 3: Comparison of baseline and aligned C code bases for a simulation with a 1000 mesh points and 50 frames

4.3 Naive OpenMP Implementation:

A naive OpenMP implementation was used initially in this code base where no subdomain blocking has yet to be implemented. The purpose of this is to see what effects parallelization techniques will have on an already fairly well optimized code base with good performance. The OpenMP implementation will be based upon the best implementation used in the C++ code base. Although, one of the major differences will be that the user can now directly specify the number of threads to use when the program runs. The main shallow2d run function now looks like the following:

```
omp_set_num_threads(p); \\where p is the number of threads being used
        #pragma omp parallel
        {
```

```
        #pragma omp single
        {
        central2d_periodic(u, nx, ny, ng, nfield);
        speed(cxy, u, nx_all * ny_all, nx_all * ny_all);

        dt = cfl / fmaxf(cxy[0]/dx, cxy[1]/dy);
        if (t + 2*dt >= tfinal) {
            dt = (tfinal-t)/2;
            done = true;
        }
        }
        central2d_step(u, v, scratch, f, g,
                       0, nx+4, ny+4, ng-2,
                       nfield, flux, speed,
                       dt, dx, dy);
        central2d_step(v, u, scratch, f, g,
                       1, nx, ny, ng,
                       nfield, flux, speed,
                       dt, dx, dy);
        #pragma omp single
        {
        t += 2*dt;
        nstep += 2;
        }
        }
```

It was then found that the following functions could contain a parallelized for loop: central2d_step, central2d_correct_sd, limited_deriv1, and limited_derivk. The various other functions contained in here if parallelized would lead to negative height being seen. Also, it should be noted that loop counter variable needed to be declared outside the loop and then declared private for the C code. The actual results of implementing this will be seen in the weak and strong scaling subsection, but it should be noted that this implementation resulted in a vastly slower performance. It needs to also be noted that when the mesh becomes more refined an increase in threads will lead to a height less than 0 assertion to happen. This was discovered when trying to run the weak scale study. One explanation could be that a nonintuitive race condition is occurring that is allowing the one thread change the value of variable that another thread has yet gotten to use yet. However, the OpenMP implementation of a barrier at the end of each for loop should prevent this from occurring.

4.4 Weak and Strong Scaling for C code:

The strong scaling for the naive OpenMP implementation can be seen in Table 4 and Figure 14. It can be seen that the parallelization has led to a degradation in performance and this can be mainly contributed to the communication cost between threads. Also, it is not as large of a factor anymore the constant creation of threads is also a problem. One thing that should also be noted is that since a less than 0 assertion can be reached for larger mesh sizes the strong scaling done for the C code base is not 1:1 to the one done for the C++ code base.

Threads	Total Time (seconds)	Strong Scaling	Strong Scaling Efficiency(%)
1	2.39	1.00	100.0
2	11.40	0.21	10.0
4	20.71	0.12	3.0
8	25.03	0.10	1.0
16	31.13	0.08	0.5

Table 4: Strong scaling for wave simulation with 400 mesh points in each direction and 50 timesteps between frames.

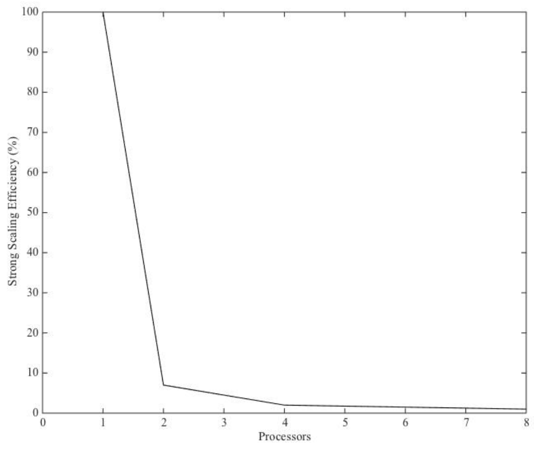


Figure 14: Plot of strong scaling efficiancy, representing the data shown in Table 4.

Next, the weak scaling can be seen in Table 5 and Figure 15. A couple more points were obtained for this study by taking a $\sqrt{2}$ increase in cell size in order to be able to just double the number of threads. It can also be seen from this table that the naive implementation of OpenMP had detrimental effects on the performance of the code.

It can be seen from these scalings that in order to even attempt at getting better performance out of the code a subdomain implementation of OpenMP will need to be used.

4.5 Parallelization

The C code was explicitly decomposed into subdomains using OpenMP with a distributed memory type paradigm. In `ldriver.c`, a thread team is launched that decomposes the domain by `npx` processors in the x-direction and `npy` processors in the y-direction, where

Threads	Cells Per Side	Total Time (seconds)	Weak Scaling(%)
1	200	0.35	100
2	283	5.11	7
4	400	17.70	2
8	566	55.61	1

Table 5: Weak scaling for wave simulation with 50 frames and N varying mesh size.

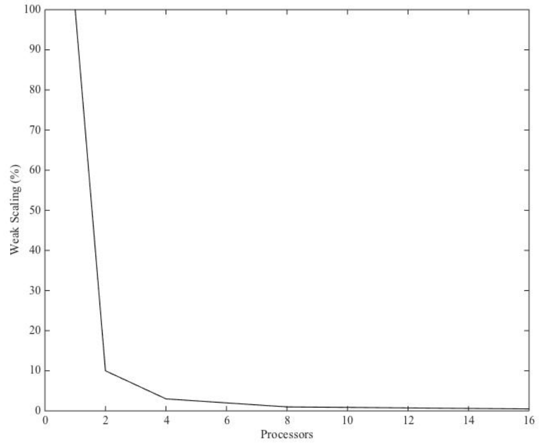


Figure 15: Plot of weak scaling, representing the data shown in Table 5.

`npx` and `npy` can be set by the lua script. It is required in the code, however, that the size of the domain be evenly divisible by the number of processors. Once the thread team is launched, the location of the subdomain on the global domain is noted for each processor for easy copying to the global domain. After this point, each subdomain is not aware of the others and the simulation is effectively distributed memory. For exchanging information between threads and informing the ghost cells, the global domain is used. Communication is handled as followed. While in the flow solver, boundary cell values are pushed to their correct location on the global domain. Next, one processor runs the original periodic boundary condition function. Then, each subdomain fills its ghost cells using the global domain. This is not the most efficient method due to the barriers required while one processor performs the periodic boundary condition update on the global domain, however provided easy implementation. The one other quantity communicated between the processors is the time step, used to restrict the time step each processor used to the most stringent time step in the entire global domain. This required large modifications to the `central2d_t` structure and other portions of the code. The modified code can be seen in the `dist_par` subdirectory of the git repository. As shown below, significant improvements in strong and weak scaling were gained when compared to a naive OpenMP parallel-for implementation.

4.5.1 Strong and Weak Scaling

The strong and weak scaling for this parallelization can also be analyzed as before. Although it will not be a direct comparison to the study done using the naive OpenMP implementation and more tuned C code, the same running parameters will be used, allowing some small comparison. The strong scaling study was run for a simulation with 400 mesh points per side and 50 frames, the results of which can be seen in Table 6 and Figure 16. Strong scaling was once again calculated using Eq. 1, with the normalized strong scaling calculated with Eq. 2.

Threads	Total Time (seconds)	Strong Scaling	Strong Scaling Efficiency(%)
1	5.974	1.00	100.0
2	3.272	1.82	91.0
4	1.695	3.52	88.0
8	0.9780	6.11	76.4
16	0.8821	6.77	42.3

Table 6: Strong scaling for wave simulation with 400 mesh points in each direction and 50 timesteps between frames using distributed memory paradigm.

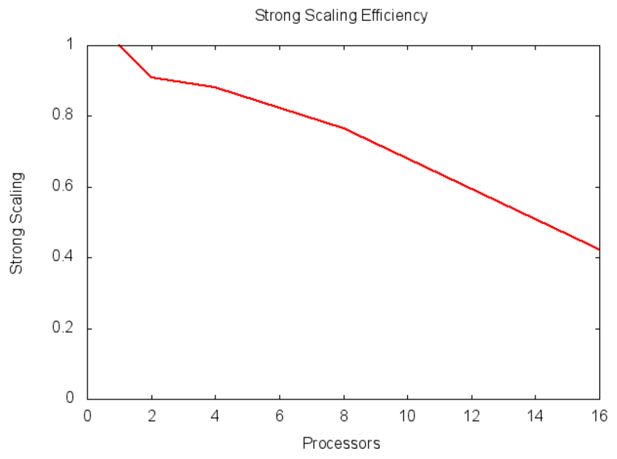


Figure 16: Plot of strong scaling efficiency, representing the data shown in Table 6.

There are three important facts to note from the strong scaling study. First, running this parallelized version with one processor (in serial) is a little less than three times slower than the tuned version of C we also presented. This is most likely due to the more complicated boundary condition handling and the existence of two grids, even when running the parallel code in serial. Second, the strong scaling shown here for this parallelized C version performs much better than the naively OpenMP parallelized version. This is due to the separation of the thread team during the driver stage and limited communication afterwards. Third, There is a stark degradation in strong scaling between 8 processors and 16 processors. This is most likely due to the fact that the processors used now exist on two separate chips on the node. This will significantly increase communication costs which will impact performance, as seen in Table 6.

Similarly, a weak scaling study can also be performed. This is shown in Table 7 and Figure 17. Weak scaling was once again calculated using Eq. 3.

Threads	Cells Per Side	Total Time (seconds)	Weak Scaling(%)
1	200	0.8876	100.0
2	284	1.308	67.9
4	400	1.708	52.0
8	566	2.655	33.4

Table 7: Weak scaling for wave simulation with 50 frames and N varying mesh size using distributed memory paradigm.

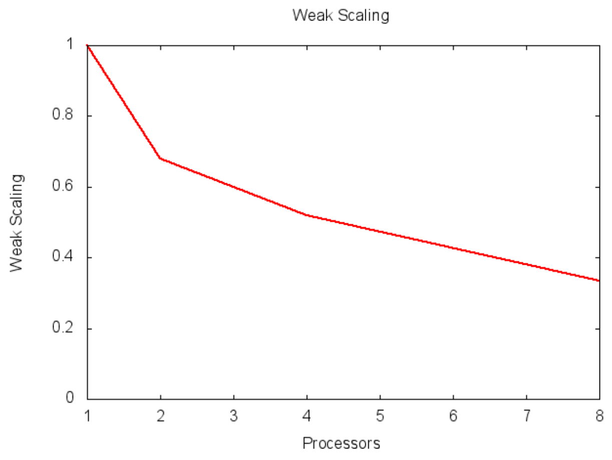


Figure 17: Plot of weak scaling, representing the data shown in Table 7.

Once again, the weak scaling with this parallelization method proves to be much more effective than the naively implemented OpenMP parallelized code used on the tuned C code. This is due to the much lower communication and synchronization in this parallelization method as opposed to the naive OpenMP implementation. Please note, the number of cells per side were slightly modified to make them evenly divisible by the number of processors for that side, a requirement of this parallelization method.

5 Conclusion

The C++ and C code implementation showed the different effects that parallelization can have on different effects forms of code. In the C++ code base where the level of abstraction and structure of arrays setup made it difficult to vectorize the code, it was seen that a basic

OpenMP implementation could lead to a nice speed up of the code. On the other hand, the C code was fairly well optimized and vectorized already, so the cost of communication between threads can not hide between the calculations time. Therefore, the naive OpenMP implementation leads to a performance degradation. When the subdomain decomposition was added, it could be seen that a performance boost could be had by adding a parallelization portion of the code.