Group 18 Project 2

Eric Gao – emg222
Yu Su – ys576
Vikram Thapar – vt87

0 README

We have several versions of `central2d.h`.

- `central2dBasic.h`: This is the original code.
- `central2d_handvectorized_serial.h`: This is the hand vectorized version of the corrector part of the `compute_step` function. This code is run serially.
- `central2dParallel.h`: This is the parallelized version with domain decomposition with no vectorization.
- `central2dParallelAligned.h`: This is the parallelized version with `ivdep`.
- `central2d.h`: This is our fastest version. It is the same as `central2dParallelAligned.h`.

You can replace `central2d.h` with any version you want.

In the end, we got a speedup of about 5.5 times for a 300 x 300 board with 50 frames when comparing our fastest code with the naive code. In both the naive and our code, we have replaced the `copysign` function with `if` statements.

1 Profiling

We began the project with some basic profiling with basic hotspot analysis. We first looked at the vectorization report to see that nothing was vectorized and that could be something we could start with. We then used VTune Amplifier to see where the bottlenecks in the code are.

The initial report created by VTune Amplifier was:

```
Function              Module        CPU Time   Spin Time   Overhead Time
--------------------  ------------  --------   ---------   -------------
limited_derivs        shallow         1.174s          0s              0s
compute_step          shallow         0.562s          0s              0s
compute_fg_speeds     shallow         0.199s          0s              0s
write_frame           shallow         0.005s          0s              0s
run                   shallow         0.003s          0s              0s
solution_check        shallow         0.001s          0s              0s
```

We see that the slowest part of the code are the two functions: `limited_derivs` and `compute_step`. Looking at the vectorization report, we see that there is a lot of unvectorized code that could be optimized. Furthermore, there are several parts that are inside of loops that could be parallelized `omp parallel for` or some other scheme of parallelization.

We then could use VTune Amplifier to look at specific lines of the code for the given functions and see how long they take. It seems pretty obvious what we need to do for the `limited_derivs` function: we need to implement some blocking scheme so that we can do some parallelization using `OpenMP`. We then saw that the slowest part of the `compute_step` function was the corrector step of the function, which took about 3 times as long as the predictor step. The equation that was being implemented involves operations on vectors that could first be vectorized and then parallelized. To do this, we thought we would have to modify how the data was laid out in memory so that it was contiguous in memory.

2 Parallelization

2.1 Naive omp parallel for

We have used "pragma loop for" in two functions which are the bottleneck in the calculations: `limited_derivs` and `compute_step`. The report created by VTune amplifier after modifying the code was:

Function	Module	CPU Time	Spin Time	Overhead Time
limited_derivs	shallow	1.173s	0s	0s
compute_step	shallow	1.422s	0s	0s
compute_fg_speeds	shallow	0.368s	0s	0s
write_frame	shallow	0.004s	0s	0s
run	shallow	0.015s	0s	0s
solution_check	shallow	0.009s	0s	0s

The above time report shows that the parallelized version of the code for the two bottleneck functions is slower than the basic code. Contradictorily, we found that if we compare the times reported in .o files, the parallel code takes a total time which is approximately 2 times faster than the basic code. We are not able to understand this discrepancy as one would expect similar trends in both the cases.

2.2 Domain decomposition

The serial computation of time step is done on a thread number 0 by using if loop. The alternate/more efficient way is to use pragma omp critical/atomic to perform the serial work. But, we faced some complications while using it as we were not able to synchronize all the threads leading to incorrect results. The number of domains used are N*N where N is an integer.

To perform the initial test, we use N = 4, NUM = N*N = 16 on 200by200 system for 50 frames. *Note that for both basic, parallel and later versions, we removed the copysign in minmod and inserted if loops.* The total time taken by our parallelized code is approximately 0.6 seconds as compared to 1.6 seconds taken by the basic code. So, we were able to obtain the gain of about 2.5 times. Also,

Algorithm 1 DomainDecomposition()

```
Use pragma omp parallel by setting u_ as the shared variable
Use omp_set_num_threads(NUM) where NUM is the number of domains. There were 3 ghost cells per domain.
Initialize vectors in each thread's domain.
Assign domain index for each thread using omp_get_thread_num().
Copy the center of domain vectors from u_ for each thread.
while time < endTime do
  barrier
  Serial computation of time step
  barrier
  Copy the domain boundary vectors from u_ for each thread.
  for i=0; i < 2*Nsteps; i++ do
    Here Nsteps is the number of steps by which domain evolves.
    Flux computation
    x and y derivatives
    Predictor and Corrector step
    Copy back to main grid of the domain
  end for
  Copy the center domain vectors to u_
  barrier
end while
```

when comparing shallow.oParallel and shallow.oBasic, the momentum, volume and range values are approximately equal to each other but not exactly. We believe that this could be due to the machine precision errors as our generated out.mp4 files looks qualitatively the same. Note, that in this code we do not perform any vectorization. So, in the next section, we will go through the vectorization part and combine it with our parallel code to perform strong and weak scaling.

3 Tuning

3.1 Initial attempts

We tried writing a kernel for the corrector step of the `compute_step` function. However, it looked like the most effective thing that we could do was also one of the simplest. We saw that we were always using the `vec` of length 3 for the innermost loop, so we just unrolled that loop. The compiler then took care of the rest and we went from taking approximately **4.5e-2** time to about **3.6e-2**, which is about a 25% speed up due to this simple optimization.

We tried another thing which was to try to cache all the `vec`s that were being used to attempt to increase speed, but it resulted in times similar to the basic approach. We also attempted to implement our own kernel, but it looked like the compiler beat us, since were were getting times comparable to the basic approach. Our attempted kernel is commented out in our `central2d.h` in the `compute_step` function.

3.2 Hand Vectorized Code

So afterwards, we went back and we wrote a kernel that works for an 8 x 8 block. This works by looping over all of the relevant `vec`'s in in the block and loading them into contiguous memory that was `_mm_malloc`'d in the `Central2D` class. We then wrote about 2000 lines of hand vectorized code to perform the corrector step of the `compute_step` function. The relevant code can be found in `central2d_handvectorized_serial.txt`.

We then added timers surrounding the corrector step of the `compute_step` function. We then ran on the initial 200 x 200 simulation to see the speedup.

Naive times	Hand vectorized times
9.019375e-04	7.851124e-04
8.959770e-04	7.750988e-04
9.019375e-04	7.801056e-04
8.959770e-04	7.729530e-04
9.009838e-04	7.808208e-04
8.931160e-04	7.739067e-04
9.021759e-04	7.810593e-04
8.950233e-04	7.739067e-04
9.019375e-04	7.879734e-04
8.950233e-04	7.741451e-04

From this we can see that we get about a 15% speed up, which is not that great considering how much code we wrote and how difficult it was to debug.

One particular reason why it was not that great of a speedup was because we have to copy data from memory over from unaligned memory into aligned memory and then copy the results back into the appropriate result vectors. One way to overcome this would be to refactor the C++ code to use C style arrays. If we do that, we could call `_mm_malloc` on the arrays initially created. We can then avoid having to reload arrays every time we call our kernel. However, this required extensive refactoring of the C++ code, so we did not do this.

3.3 Vectorization using ivdep

To perform vectorization on this code, we have used pragma ivdep. As mentioned in intel website, ivdep provides the instruction to the compiler to ignore vector dependecies and performs the auto vectorization. In the profiling section, we show that the limited_derivs is the computationally expensive step. So we decided to put #pragma omp ivdep in limdiff function in central2d.h. The modified function looks like

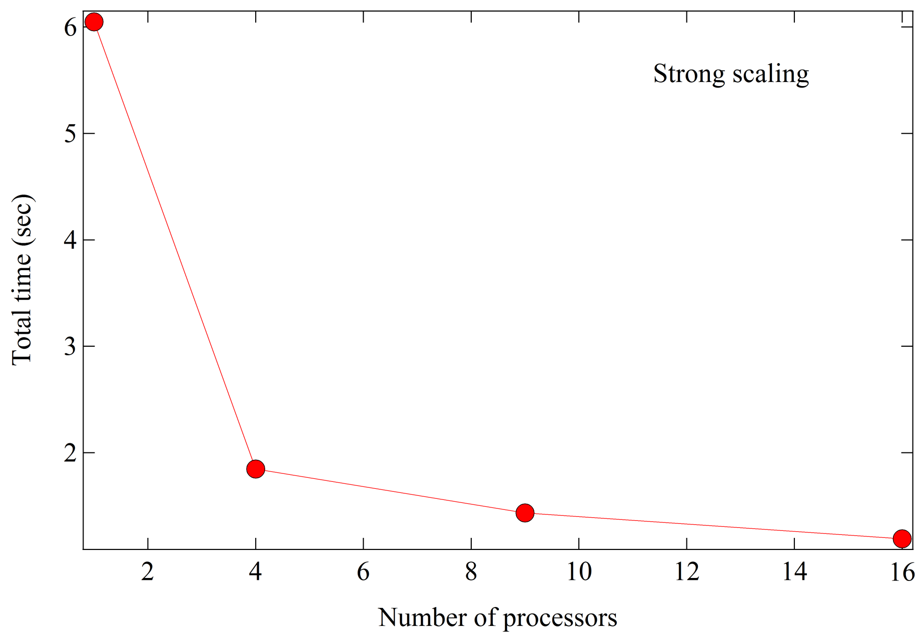
```
static void limdiff(vec& du, const vec& um, const vec& u0, const vec& up) {
    #pragma ivdep
    for (int m = 0; m < du.size(); ++m)
        du[m] = Limiter::limdiff(um[m], u0[m], up[m]);
}
```

On adding ivdep, we were able to get a significant gain. The parallelized code run for section 2.2 now takes a total time of 0.43 seconds.

4 Scaling

We split the scaling section into two subsections. For the scaling, we use our fastest code which is the parallelized code combined with the vectorization of limdiff using ivdep.

4.1 Strong Scaling



When performing these studies, we kept the number of frames constant at 50, and the size of the board constant at 300 $\times$ 300.

When we look at the results, the total time taken decreases significantly decreases initially, but as we throw more and more processors at the problem, the speedup plateaus. This is most likely because with a fewer processors, there is little communication overhead. However, when we throw more processor at it, the communication overhead increases and as a result, the speedup is not as significant.

One of the ways we could avoid this issue is to use `omp critical` or `omp atomic`. Another thing we could have done is to use two copies of the board. On even time steps we could then write to the first copy of the board and on odd timesteps, we could write the second copy of the board. As a result, we would only have one barrier instead of two, which would reduce the amount of time threads are idle and potentially increase speedup.

According to Amdahl's law, the speedup we should see according to this simple model is:

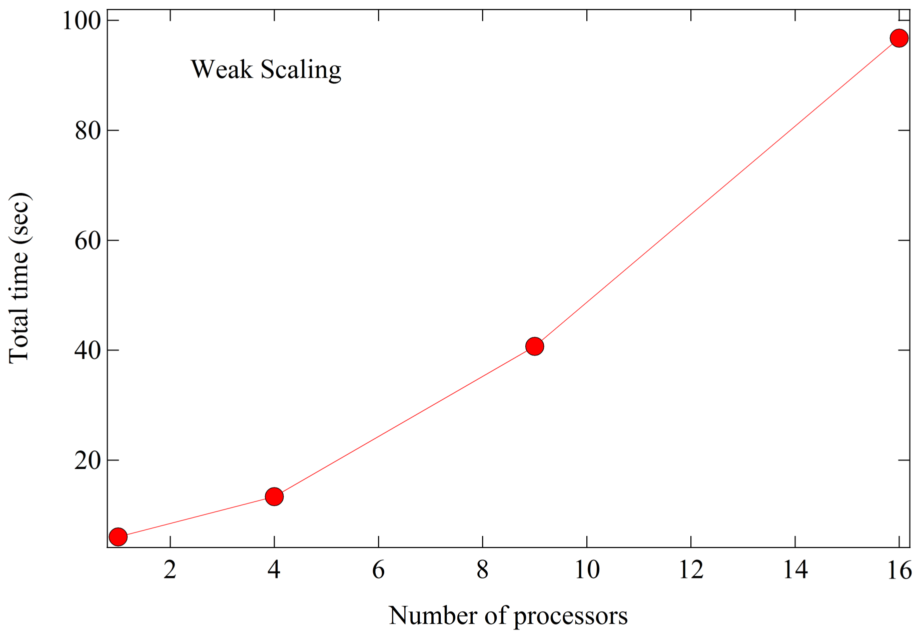
$$S(p) \leq \frac{1}{\alpha + (1-\alpha)/p} \leq \frac{1}{\alpha}$$

where α is the fraction of serial work that cannot be parallelized. Therefore, because there is some fraction of serial work that cannot be parallelized. Therefore, the amount of time that it takes for the simulation to run should approach the amount of serial work there is. If there were no serial work, then the speedup should be directly proportional to the number of processors.

If we compare the parallelized code with the serial code, we see a total speedup of:

$$Speedup = \frac{6.0 \text{ seconds}}{1.2 \text{ seconds}} = 5$$

4.2 Weak Scaling



When performing these studies, we kept the number of frames constant, but we increased the size of the board proportional to the number of processors. The exact board sizes and number of threads we used were:

Number of threads	Board size
1	300 x 300
4	600 x 600
9	900 x 900
16	1200 x 1200

Ideally, if there were no serial overhead and no communication overhead, then the amount of time should be constant as we increase the size of the board proportional to the number of processors. However, this is not the case. The deviation from a constant time for increasing number of processors indicates that there is increasing communication overhead as we increase the number of processors and we are increasing the amount of serial work when we increase the number of processors.