CS 5220 Project 2 Final Report

Dylan Foster (dj244)
Weici Hu (wh343)
Yuan Huang (yh638)

October 30, 2015

1 Introduction

We sought to optimize the performance of a simple hyperbolic PDE simulator. In particular we worked on optimizing the simulator's performance using parallelization on the Xeon Phi cluster. We initially profiled a fast serial implementation of the simulator, then worked on parallelizing the simulator's critical loop using OpenMP. For the second half of the project, we first implemented a domain decomposition to parallelize our main computation loop, and then sped this implementation up by offloading the computation to the Xeon Phi coprocessor.

2 Serial Performance

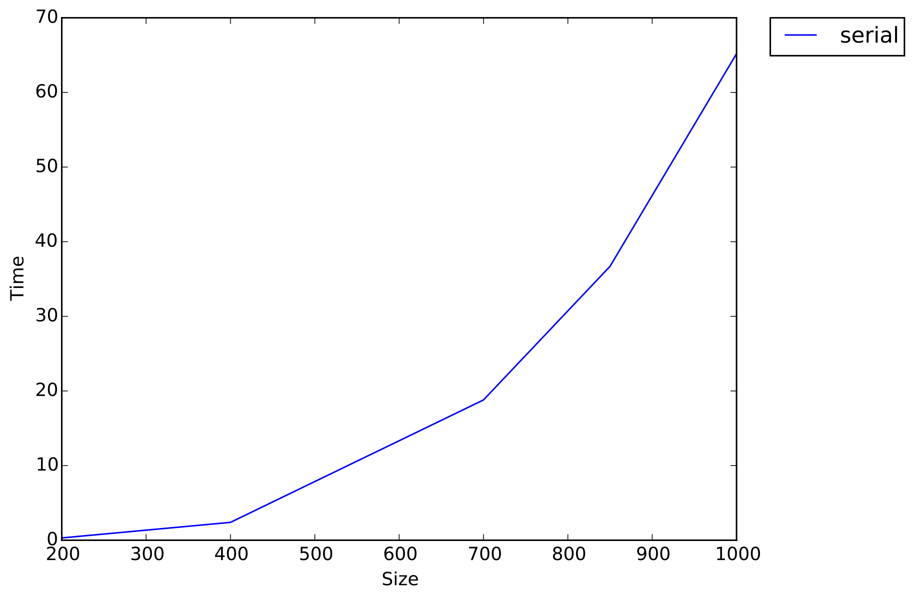
The serial implementation of the shallow wave simulator we used is available at `https://github.com/dbindel/water`. We compiled this implementation with icc. This implementation is written and C and features optimizations including:

- Improved aliasing through use of the restrict keyword.
- Branch-free computation of the MinMod limiter.
- Vectorization via the `-axCORE-AVX2` compiler flag.
- Data vectors such as u are stored in a "struct of vectors" style instead of the "vector of structs" style from the original C++ implementation.

We ran the serial code with a range of grid dimensions. The results of this test are shown in Figure 1. This running time shown in the plot appears to quadratic in the board dimension, which is to be expected. Running the Intel's VTune Amplifier profiler on the executable indicates that most of the computation time is spent in the `central2d_predict`, `central2d_correct`, and `shallow2d_flux` functions.

Table 1 below gives a list of functions that takes up the most time in the serial implementation with an input dimension of 400. The total run time is

Figure 1: Performance for baseline serial implementation.



2.4s, which is all calculation time.
In view of these results, it made most sense to try to paralellize the `central2d_step` function, which is comprised almost entirely of the functions described above.

Table 1:

function	CPU time(s)	spin time (s)
central2d_step	0.636	0
shallow2d_flux	0.473	0
_intel_ssse3_rep_memcpy	0.326	0
central2d_correct	0.316	0
xmin2s	0.173	0
limited_deriv1	0.153	0
limited_derivk	0.072	0
central2d_correct_sd	0.071	0

3 Basic Parallelization with OpenMP

To familiarize ourselves with the framework and OpenMP, we tried some very primitive approaches to parallelization which, unsurprisingly, were not successful

Table 2:

function	CPU time(s)	spin time (s)
__kmp_wait_template<kmp_flag_64> libiomp5.so	68.98	68.98
shallow2d_flux	1.556	0
__kmp_yield libiomp5.so	1.539	1.539

on their own. Since shallow2d_flux uses a large amount of the runtime, and is embarrassingly paralell, we naively parallel the for loops in this function, and get the profiling results shown in Table 2. The total run time increases from 2.4s to 4.3, which shows that naive parallel introduces a large amount of overhead time, and also time spent on communication, which can't be compensated for by having multiple jobs run concurrently. These results suggest that domain decomposition is the correct path to pursue if we want to improve our parallel performance.

4 Domain Decomposition

One way to realize paralellization of the code is to use domain decomposition. The idea of domain decomposition is to divide the initial nx×ny grids into sub-domains and to use ghost cells to enable independent computation of each sub-domains in some period of time before communicating the results.

4.1 Division of sub-domains and addition of ghost cells

We perform decomposition by horizontally dividing the domain into NUMPARA sub-domains. We chose this approach to domain decomposition because it enjoys spatial locality, and because we don't benefit from breaking the domain into smaller chunks with the relatively modest number of processors on the main Xeon board. horizontal division is used since this gives a better spatial locality. The total number of cells in each a sub-domain is ny/NUMPARA * nx. We also add N layers of ghost cells around the sub-domains, so we do N steps of computations on each sub-domain before communicating the results among different threads.

Before performing the computation on each sub-domain, we first copy the data from the main block for the corresponding sub-domain and the corresponding ghost cells to a separate location. For example,

```
memcpy((ublock[curthread]+k*nx_all*(2*NBATCH+blocksize)),
u+k*nx_all*ny_all+(nx_all*(blocksize*curthread)),
(nx_all*(2*NBATCH+blocksize))* sizeof(float));
```

Within each round of computation, for the i^{th} step, $1 \leq i \leq N$, we update the value for the canonical cells as well as the inner $N - i$ layers of ghost cells.

Here's how we do the update:

```
for(int j = 0; j<NBATCH/2;++j){
    central2d_step_i(ublock[curthread], vblock[curthread],
    sblock[curthread], fblock[curthread], gblock[curthread],
    0, nx+2*(NBATCH-1-j*2), blocksize+2*(NBATCH-1-j*2),
    1+j*2, nfield, flux, speed, dt, dx, dy, curthread);

    central2d_step_i(ublock[curthread], vblock[curthread],
    sblock[curthread], fblock[curthread], gblock[curthread],
    1, nx+2*(NBATCH-2-j*2), blocksize+2*(NBATCH-2-j*2), 2+j*2,
    nfield, flux, speed, dt, dx, dy, curthread);}
```

After we complete the updates for each sub-domain, we copy the updated data back to the main block in the same manner as we copy it from the main block.

4.2 Use of OpenMP

We use NUMPARA number of threads. We wait for all the threads to complete copying before performing the update by adding a barrier at the end of copying. Also we put a barrier at the end of copy back to make sure every thread gets the correct data before starting the copy in next round. The basic structure is:

```
#pragma omp parallel num_threads(NUMPARA)
{
    // copy things to each sub-domain
    for(int k =0;k<nfield;++k){
        ...// copy into private space
    }
    #pragma omp barrier
    // simulate NBATCH steps, for step j we update the (NBATCH-1-j*2) most insid
    for(int j = 0; j<NBATCH/2;++j){
        ...// do simulation
    }
    // copy back from the sub-domains
    for(int k = 0;k<nfield;++k){
        ...//copy from the private space
    }
    #pragma omp barrier
}
```

4.3 Performance Analysis

Figure 2 is a plot of strong scaling using the parallelization scheme discussed above on a problem of size of 1000×1000. The number of threads are set to be 1,4,8,12. Green line shows the plot when we set N to be 12 and the blue line shows the plot when N is set to be 6. Intuitively we expect the run-time to be shorter for N larger since there is less time spent on communication. However it is not the case. For number of thread being 1, this is because while the total communication time remain the same for $N = 6$ and $N = 12$ (since there is only one thread), there are computation to be done on updating ghost cells. For number of threads greater than 1, we are not entirely sure why the reduction in communication time insufficient to bring down the run-time.

Figure 2: Strong scaling performance of domain decomposition.

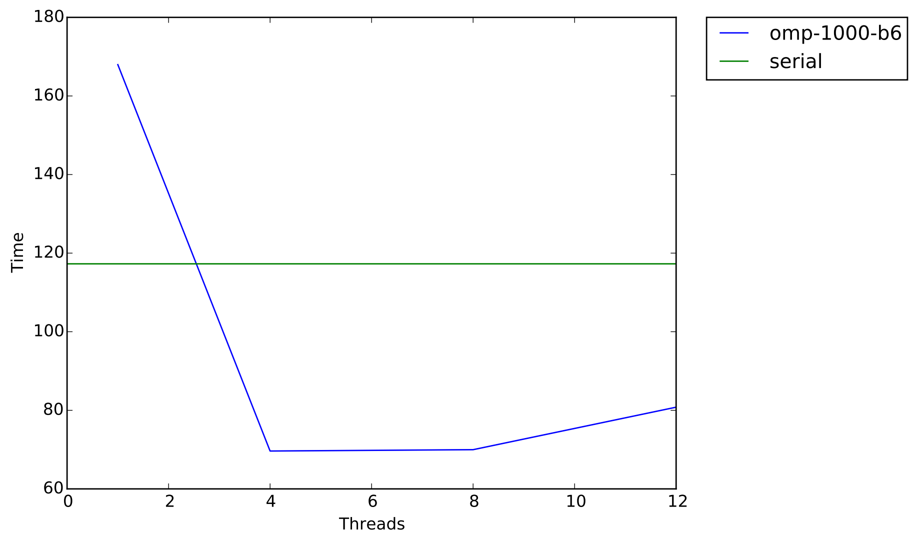


Figure 3 is a plot of weak scaling performance of our domain decomposition implementation. We begin with a board of edge size 400, which has a runtime of 8.5 seconds. We then update the board size to keep the work per thread constant. Note that if we increase the edge N, the total work increases as N^3, because the number of cells grows quadratically with N and the number of iterations grows linearly with N (because the wavespeed decreases). Our edge size as a function of the number of processors is thus

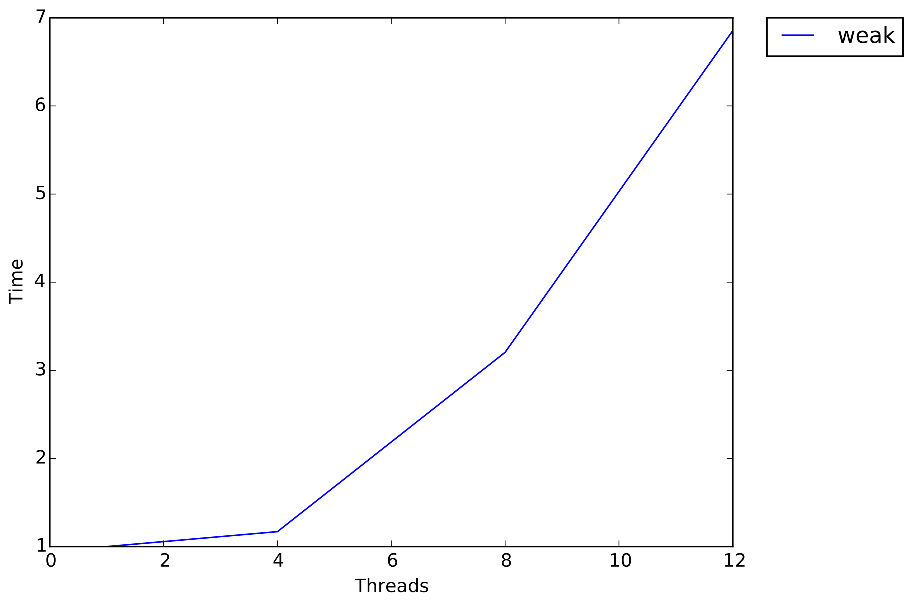
$$N(p) = N(1)p^{1/3},$$

where in this case $N(1) = 400$. We would ideally like the weak scaling plot to be flat, but we can see that it in fact increases significantly with the number of threads p. We suspect that this is due to poor performance due to communication overhead.

5 Offloading

We tried offloading the parallel computation to the Xeon Phi to make use of the accelerator board. We used command "#pragma offload target(mic) " in

Figure 3: Weak scaling performance of domain decomposition.



function "central2d_xrun".

```
#pragma offload target(mic)
inout(u,v,f,g:length(nx_all*ny_all*nfield))
in(scratch:length(6*nx_all)) inout(nstep,t,done)
{
...
}
```

This declaration is basically wrapped around the body of our parallelized `central2d_xrun` function, with some modifications to make the computation more efficient and to ensure that it is correct. Initially there were some problems in using the command. The first problem is the pointer of a function cannot be passed during offloading. So we directly call the "shallow2d_speed" and the "shallow2d_flux" in shallow2d.h. Despite being a relatively primitive attempt at offloading our computation, this implementation had improved performance over the non-offloaded parallel implementation.

5.1 Performance Analysis

Figure 4 shows the strong scaling performance of our offloading implementation with an edge size of 1000 and a range of thread numbers. Also plotted is the performance of the domain decomposition without offloading and our best

purely serial implementation. We see that when there are few threads, the non-offloading implementation works best, but with enough threads the offloading implementation works best of all three implementation. At roughly 90 threads there is an inflection point and increasing the number of threads begins to increase the runtime.
For small thread sizes, say, in the range 1-15, the cost of transferring memory to the Xeon Phi coprocessor relative to the speed at which we finish computing on the coprocessor is such that offloading introduces great inefficiency. As we approach the > 100 thread range, each thread is responsible for < 10 rows of the simulation, so there are almost as many ghost rows as there are primary rows on each processor. This yields the diminishing returns we see for increasing the number of threads.

Figure 4: Strong scaling performance of domain decomp. with offloading.

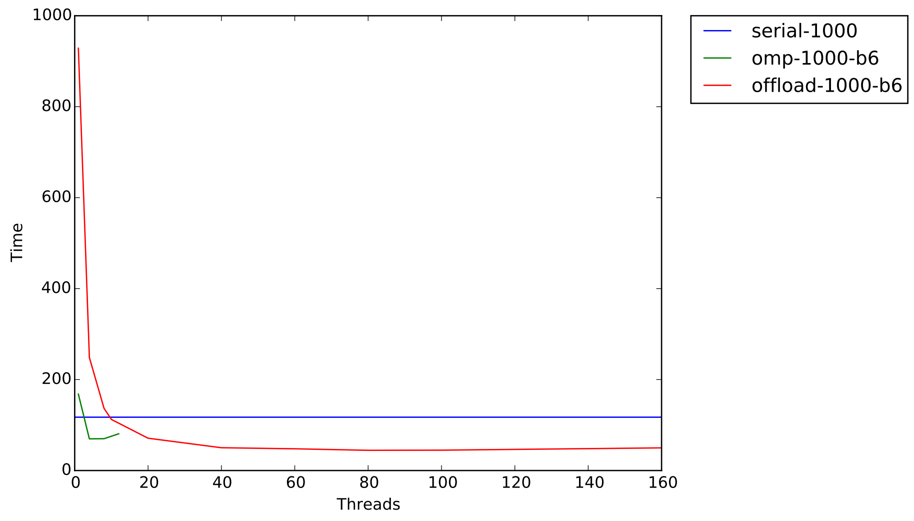


Figure 5 shows the weak scaling performance of our offloading implementation. Pleasantly, weak scaling performance of this implementation is improved compared to the non-offloaded implementation (Figure 3). We notice that in the thread count regime where we tested the non-offloaded implementation, our weak scaling is at most 2, where it was as large as 8 in the non-offloaded implementation. Furthermore, the weak scaling of the offloaded implementation never grows larger than 4, even when there are more than 100 threads.

We also did experiment about the best batch size. This is shown in Figure 6. From the figure, we can see that batch size of 4 is the best batch size in the 400*400 grids with 10 threads. The figure shows that increasing batch size will first decrease then increase the running time. That's because we should have number of ghost cells =batch when we copy the sub-domain to the private space. At the beginning, increasing the batch size can decrease the time needed

Figure 5: Weak scaling performance of domain decomp. with offloading.

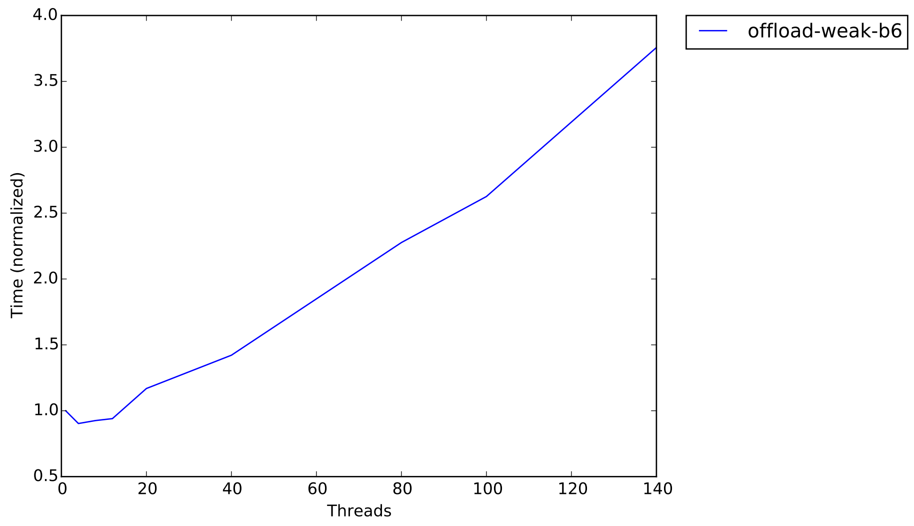
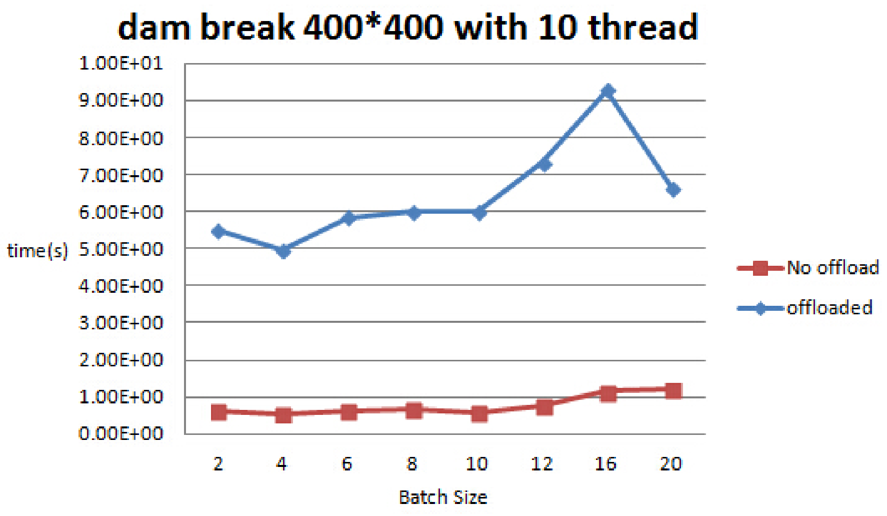


Figure 6: The running time against batch size



for communication; but when the time of calculation dominates the time of communication, the cost for the extra ghost cell is greater than the save of communication time. So we see the running time first decreasing then increasing in the graph.

6 Conclusion

The most obvious improvement to this work would be to tune our offloading code. Our current implementation is relatively naive in that we basically added the necessary offloading preprocessor directives to our the block of code the we had previously parallelized using OpenMP. As it stands, we spend a lot of time on unnecessary data transfers. We suspect that we could also make use of the Phi's more sophisticated features to cache the arrays we compute on across offloading calls, so that only the ghost cells need to be updated during the phases where we copy data back from the Phi. It should also be noted that we currently synchronize our threads at every step, even though we could get away with only synchronizing them every time we run out of ghost cells. This is a simple improvement that could yield noticeable speedup. Our weak scaling results for the offloaded implementation in particular suggest that this could be quite effective, since we get almost unit weak scaling in the regime where each thread processes a significant chunk of the board due to the low overhead in processing ghost cells.

Another future direction to consider would be to make our domain decomposition decompose across both X and Y coordinates depending on the input grid size. For the relatively small domain sizes we have been considering up until this point, the overhead for such a decomposition would not make it worthwhile, but for larger domain sizes, say, 10000x10000, this would yield better cache performance and allow us to make use of more of the Phi coprocessor threads.

With more time, we would have liked to have experimented with offloading to multiple Phi coprocessors or multiple nodes. The extra threads afforded by this setup could yield speedups in the regime where the domain size is very large.

One final direction that seems promising would be to experiment more with vectorization on the Phi coprocessor. We have spent little time ensuring that the code that runs on the coprocessor is vectorized optimally and in particular have not gone to great lengths to ensure that the 512-bit SIMD registers on the Phi are being utilized, so this could be a modification that would greatly improve performance.