# CS 5220 Homework 2 Initial Report

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

October 20, 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. In the future we hope to take advantage of the full power of the cluster by offloading our parallelized code to the Phi accelerator board.

## 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 2.4s, which is all calculation time.

Figure 1: Performance for baseline serial implementation.

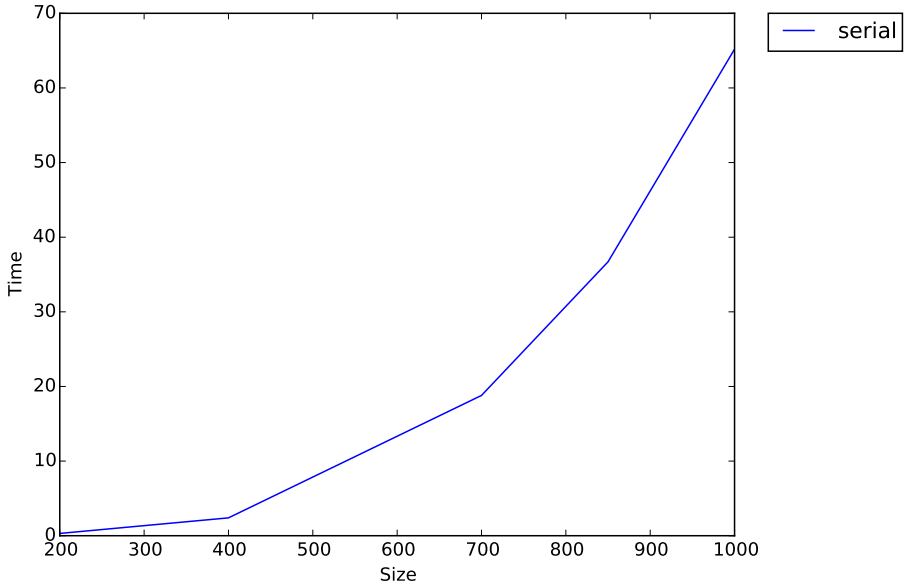


Table 1:

| 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

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.

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 |

# 4 Domain Decomposition

To achieve a better spatial locality in our parallelization, we decompose the domain to NUMPARA sub-domains by horizontally dividing the domain. Hence the dimension of a sub-domain is ny/NUMPARA * nx. We 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. 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.

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 3 is a plot of weak scaling. When keeping the amount of task constant for each of the thread, the total run-time increases with number of threads because there are more communication overhead.

# 5 Future Work

The optimization we hope to pursue is to offload the parallel computation to the Xeon Phi, as we are not currently using the accelerator board. To make full use of the Phi's 60 cores, we will also work on decomposing the domain in both axes. We will perform more profiling and scaling studies to gauge the effectiveness of these approaches. We will also work on optimizing our parallel code further and improving our memory alignment and cache performance.

Figure 2: Strong scaling performance of domain decomposition.

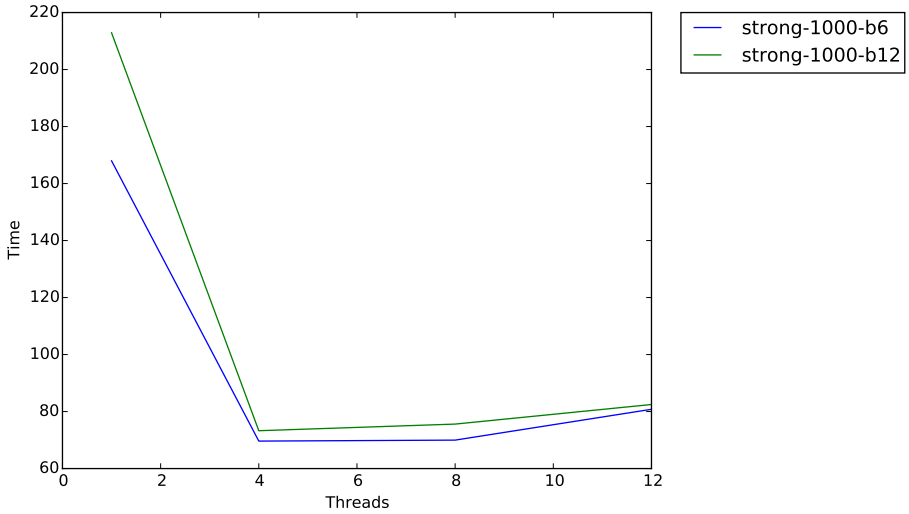


Figure 3: Weak scaling performance of domain decomposition.