Homework 2 - Final
CS 5220

Lara Backer, Xiang Long, Saul Toscano

October 30, 2015

1 Introduction

The purpose of this homework is to analyze and tune a shallow water equation solver to run efficiently in parallel. A basic serial C code is used as a starting point (forked from the bindel repository), which includes the shallow water PDEs.

The sections of this report are organized as follows: first, the basic code is analyzed using profiling tools. Next, serial performance is addressed through vectorization. Code parallelization is then incorporated through the use of openMP pragmas. Finally, the domain is decomposed and offloaded onto the various processors. Final results and scaling runs from the tuned, parallelized code and from testing processor offloading are in the Results section at the end of the report.

All results were run using the 'dam break' problem, with a visualized snapshot of the simulation water height shown in Figure 1.

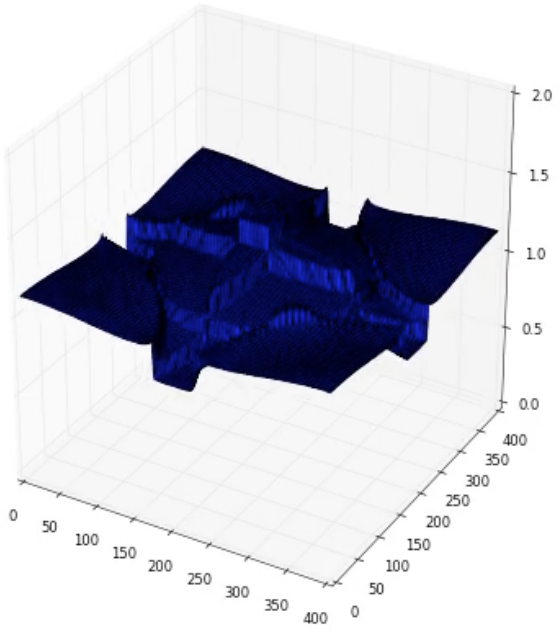

Figure 1: Dam Break Simulation Visualization

2 Initial C Code Profiling

In order to determine the areas of the code to target in tuning for efficiency, we used the Intel profiling tool VTune. For a small dam break run, the slowest 18 sections of the code in terms of time are shown in Figure 2.

central2d_step	lshallow	7.239s
shallow2d_flux	lshallow	6.970s
__intel_ssse3_rep_memcpy	lshallow	3.278s
central2d_correct	lshallow	2.655s
xmin2s	lshallow	1.992s
limited_deriv1	lshallow	1.286s
limited_derivk	lshallow	1.142s
central2d_correct_sd	lshallow	0.774s
xmin2s	lshallow	0.565s
xmin2s	lshallow	0.452s
limdiff	lshallow	0.405s
shallow2d_speed	lshallow	0.401s
[Outside any known module]	[Unknown]	0.192s
central2d_correct_sd	lshallow	0.111s
limdiff	lshallow	0.091s
luaV_execute	lshallow	0.064s
run_sim	lshallow	0.057s
central2d_run	lshallow	0.048s

Figure 2: Vtune Output - Original Code

Further profiling of the slowest functions showed that the portions that contained domain computation for loops took the most time to complete.

3 Vectorization

One method of speeding up the code is to use vectorization, applying operations to arrays instead of array sub elements, reducing the number of required computations.

To vectorize our code, we drew ideas from the vectorized C code produced by Prof. Bindel. We additionally targeted sections of our code that had to be vectorized by use of the ipo out.optreport file output after compilation, which shows the sections that have not been vectorized fully yet. Our vectorization method uses pointers to reduce overheads in manipulating the data structure.

4 OpenMP Parallelization

We parallelized the code using openMP pragmas (#pragma omp parallel) in the time advancement loop, as well as the -fopenmp flag in the compilation. Some sections, such as the new timestep computation, only needed to be performed by a single processor and so #pragma omp single was used inside of the parallel call. At the end of each timestep, a barrier was used to allow synchronization of the processors (particularly necessary for the later domain decomposition). As all for loops were encapsulated within the timestep loop (except for the domain initialization), no additional parallel calls were needed. This parallelization was found to immensely improve the time to run the code.

The periodic boundary conditions were maintained from the original code, but the number of ghost cells around the domain were varied based on the number of iterations performed within a timestep.

5 Domain Decomposition

5.1 Overview

We decomposed the domain by creating rectangles of subdomains within the domain on which each processor solves the shallow wave equations. This was accomplished by obtaining the total number of specified threads, and breaking up the domain into that many rectangles, assuming that the domain is a square with nx=ny cells in each direction. Each thread has a section of the velocity field plus additional ghost cells copied into its own subdomain,

and proceeds to solve the equations on that subdomain. After all processors have computed the new velocities for their subdomains (enforced by a barrier), the subdomain velocity fields are copied back into the overall velocity field and the timestep and periodic boundary conditions are updated for the whole domain. As before, the number of ghost cells must be varied for the number of iterations performed in a timestep in order to have enough cells to fully compute the new velocity field on the given subdomain. The number of ghost cells is set as 4*(number of subiterations), considering that the Jiang-Tadmore central difference scheme already contains an additional iteration for the predictor step.

6 Processor Offloading

Each compute node on the Totient cluster contains 24 cores. Thus, offloading to the Xeon Phi coprocessors which have 236 threads per node is desirable, particularly for large domains and numbers of desired cores. To offload to the Phis, we simply used the command: #pragma offload target(mic:thread_number) immediately inside the parallel section of the code after domain initialization, but prior to the time stepping section. Results comparing the C code containing the domain decomposition, vectorization, and openMP parallelization to the same code offloaded to the Xeon Phi coprocessors, and the oroginal C code, are shown in the following Results section.

Strong scaling is viewable in the strong scaling efficiency graph - better scaling than both original and new c codes on the compute nodes. However, timings for equivalent runs take longer.

7 Results

7.1 Scaling

The code complete with domain decomposition, openMP parallelization, and vectorization was used for strong and weak scaling. Similar scaling cases were done for the same code on the Xeon Phi ncoprocessors, as well as the original C code.

Strong scaling is defined as:

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

and strong scaling efficiency as:

$$\text{Strong scaling efficiency} = \frac{t_{\text{serial}}}{t_{\text{parallel}}} \frac{1}{p} \tag{2}$$

The results, for a 800x800 cell domain with varied thread numbers are shown in Figures 1 - 3. Clearly, the new code performs much better than the old code, by an order of magnitude. The Xeon Phi coprocessors should start to perform better at higher thread numbers, and it is assumed that they are slower for the lower number of processors due to the offloading step.

Threads	Time	Strong Scaling	Scaling Efficiency
1	0.534	1	1
2	0.54	0.988888889	0.494444444
4	0.558	0.956989247	0.239247312
8	0.571	0.935201401	0.116900175

Table 1: Strong Scaling - New Code

Threads	Time	Strong Scaling	Scaling Efficiency
1	1.017	1	1
2	0.953	1.067156348	0.533578174
4	0.913	1.113910186	0.278477547
8	1.014	1.00295858	0.125369822

Table 2: Strong Scaling - New Code, Xeon Phis

Threads	Time	Strong Scaling	Scaling Efficiency
1	29.6	1	1
2	38.3	0.772845953	0.386422977
4	45.1	0.65631929	0.164079823
8	34.2	0.865497076	0.108187135

Table 3: Strong Scaling - Original Code

A graph comparing the scaling efficiencies of all three codes is shown in Figure 3.

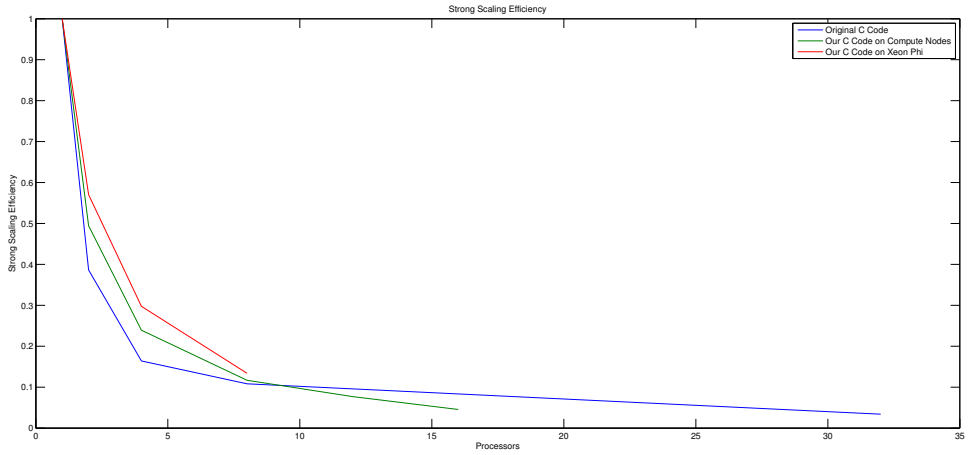

Figure 3: Scaling Efficiency Comparison

Weak scaling is similarly defined as

$$\text{Weak scaling} = \frac{t_{\text{serial}}}{t_{\text{parallel}}} \tag{3}$$

Except that for weak scaling, all processors are designated to have the same workload (number of cells); as the number of processors increases, so will the total number of cells in the domain. Tables 4 - 5 compare the weak scaling for the new code run on the compute nodes, and the original code.

Threads	nx	Total Time	Weak Scaling
1	200	0.366	1
4	400	2.71	0.135055351
16	800	29.7	0.012323232

Table 4: Weak Scaling - Old Code

Threads	nx	Total Time	Weak Scaling
1	200	0.0203	1
4	400	0.0907	0.223814774
16	800	0.734	0.027656676

Table 5: Weak Scaling - New Code

7.2 Number of Iterations

A final graph showing the timings based on the number of iterations run per step is shown in Figure 4. All of these timings were performed using the new code on the compute nodes, on a 800x800 domain, with 16 processors. There is clearly a decrease in simulation time for additional iterations beyond 1, however for higher number of iterations, the simulation run time increases. This could be due to several factors, such as increased communications for each subdomain and larger subdomain array sizes to copy. Furthermore, larger subdomains for computations may extend beyond given cache sizes.

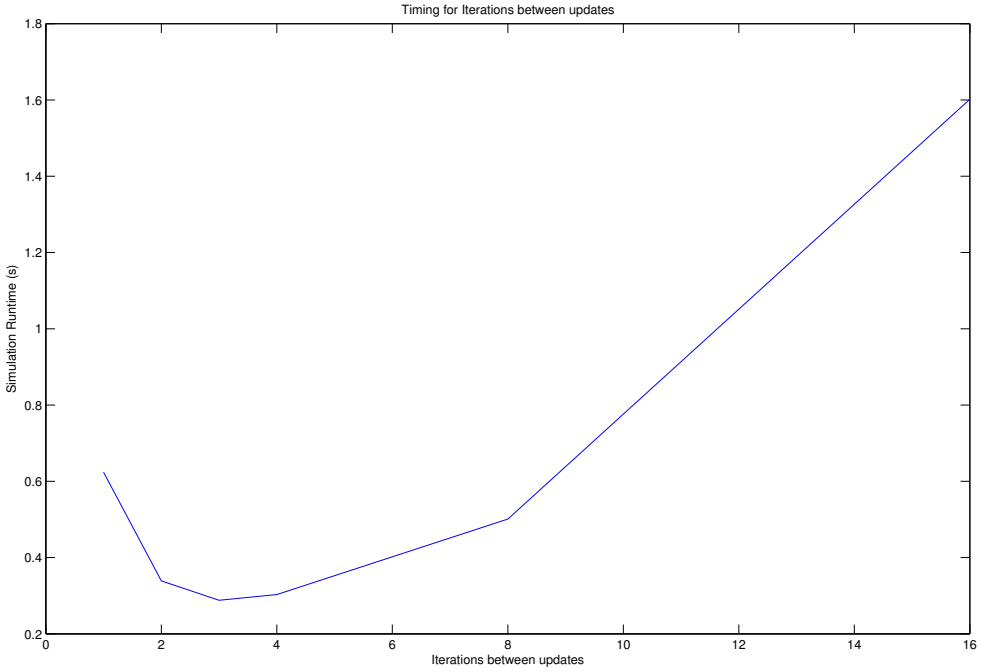

Figure 4: Timings Based on Iterations per Step

References

[1] Kevin Davis. Accessed 25 Oct, 2015. https://software.intel.com/en-us/articles/effective-use-of-the-intel-compilers-offload-features