Homework 3
CS 5220

Lara Backer, Greg Granito, Sam Tung

November 19, 2015

1 Introduction

This goal of this project is to optimize the parallelization of the Floyd-Warshall algorithm for finding the shortest path between two nodes in a graph.

A base parallelized C code using openMP was provided. From this, the objectives are to profile and tune the existing code, and to implement the parallelization of the algorithm using MPI.

2 OpenMP Code

2.1 Baseline Profiling

A look into performance of the code was conducted by using Intel's VTUNE on Totient. Due to some technical issues with the cluster, we could not run the "advanced-hotspots" for profiling.

Function	Module	CPU Time	CPU Time:Idle	CPU Time:Poor	CPU Time:Ok	CPU Time:Ideal	CPU Time:Over	Wait Time
square	path.x	41.551s	0s	8.210s	3.414s	29.927s	0s	
__kmp_barrier	libiomp5.so	12.771s	0.120s	12.321s	0.320s	0.010s	0s	5.464s
__kmpc_reduce_nowait	libiomp5.so	5.685s	0.020s	5.375s	0.210s	0.080s	0s	2.097s
__kmp_fork_barrier	libiomp5.so	3.015s	1.980s	0.863s	0.161s	0.010s	0s	0.318s
__intel_ssse3_rep_memcpy	path.x	0.040s	0.010s	0.030s	0s	0s	0s	
fletcher16	path.x	0.030s	0s	0.030s	0s	0s	0s	
__kmp_launch_thread	libiomp5.so	0.021s	0.010s	0.011s	0s	0s	0s	
gen_graph	path.x	0.010s	0.010s	0s	0s	0s	0s	
__kmp_get_global_thread_id_reg	libiomp5.so	0.010s	0s	0.010s	0s	0s	0s	0.000s
genrand	path.x	0.010s	0s	0.010s	0s	0s	0s	

Figure 1: Most time consuming functions in the base code.

From looking at VTUNE, it appears that the most time spent in the program was done on the square function which finds the minimum path between the nodes. This is not surprising, as there are plenty of nested for loops in the code, which undoubtedly takes a while to run through, as can be seen in Figure 3.

2.2 Vectorized Profiling

One way to further increase the performance of the square function is to implement vectorization, which aligns data to promote efficiency. From the results, it appears that this does indeed increase performance slightly, as time in square is slightly less than it was previously.

Function	Module	CPU Time	CPU Time:Idle	CPU Time:Poor	CPU Time:Ok	CPU Time:Ideal	CPU Time:Over	Wait Time	Spin Time
square	path.x	38.920s	0s	3.823s	2.981s	32.124s	0s		0
__kmp_barrier	libiomp5.so	5.387s	0s	4.797s	0.480s	0.110s	0s	1.173s	5.38661
__kmp_fork_barrier	libiomp5.so	2.689s	1.707s	0.771s	0.162s	0.049s	0s	0.546s	2.68937
__kmpc_reduce_nowait	libiomp5.so	2.118s	0.010s	1.538s	0.570s	0s	0s	0.412s	2.10797
__intel_ssse3_rep_memcpy	path.x	0.020s	0s	0.020s	0s	0s	0s		0
fletcher16	path.x	0.020s	0s	0.020s	0s	0s	0s		0
genrand	path.x	0.010s	0s	0.010s	0s	0s	0s		0
__kmp_get_global_thread_id_reg	libiomp5.so	0.010s	0.010s	0s	0s	0s	0s	0.000s	0
gen_graph	path.x	0.010s	0s	0.010s	0s	0s	0s		0

Figure 2: OpenMP Code with vectorization.

```
int square(int n,               // Number of nodes
           int* restrict l,     // Partial distance at step s
           int* restrict lnew)  // Partial distance at step s+1
{
    int done = 1;
    #pragma omp parallel for shared(l, lnew) reduction(&& : done)
    for (int j = 0; j < n; ++j) {
        for (int i = 0; i < n; ++i) {
            int lij = lnew[j*n+i];
            for (int k = 0; k < n; ++k) {
                int lik = l[k*n+i];
                int lkj = l[j*n+k];
                if (lik + lkj < lij) {
                    lij = lik+lkj;
                    done = 0;
                }
            }
            lnew[j*n+i] = lij;
        }
    }
    return done;
}
```

Figure 3: Square function

2.3 Scaling

Strong scaling is defined as below, as the ratio of the time for a case to run in serial compared to a case with a given number of processors in parallel:

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

Weak scaling is similarly defined as below as the ratio between serial and parallel case run times, although for weak scaling, the size of the computation on each processor is held constant.

$$\text{Weak scaling} = \frac{t_{\text{serial}}(n)}{t_{\text{parallel}}(n)} \tag{2}$$

Both strong and weak scaling studies were performed for the openMP code.

Threads	Time (s)	Strong Scaling	Scaling Efficiency
1	0.0116529	0.902942615	90.29426151
2	0.00655317	1.605619876	80.28099378
4	0.00718093	1.465255893	36.63139733
8	0.00600195	1.753080249	21.91350311
16	0.00977397	1.076522641	6.728266508
24	0.0777621	0.135308846	0.563786858

Figure 4: Strong Scaling for 200x200 Element Graph, openMP

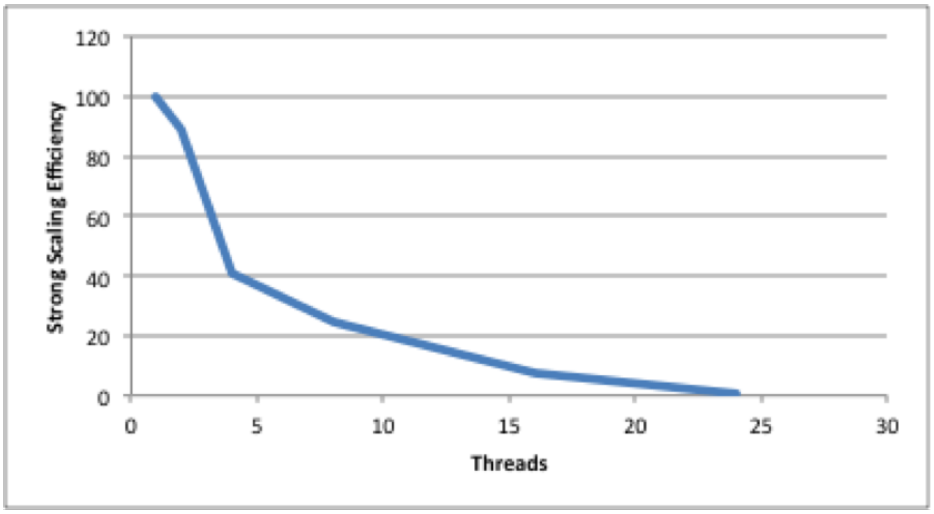

Figure 5: Strong Scaling Efficiency, openMP

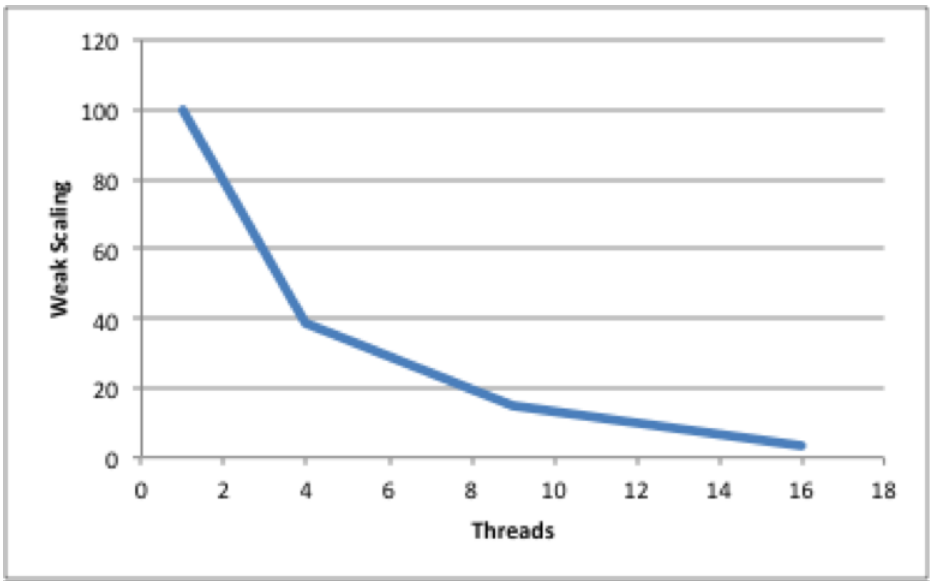

Figure 6: Weak Scaling (%), openMP

3 OpenMPI

OpenMPI was used in place of OpenMP in the square routine to compare the efficiency of both methods. To use openMPI, the graph was split into equally sized sections, with each processor assigned a section. The minimum distance between locations within each section was computed, and then gathered and saved for the overall graph by using the command MPI_ALLREDUCE and operation MPI_MIN during the computation.

3.1 Profiling

After attempting to implement OpenMPI, we once again look into performance of the code through profiling.

Function	Module	CPU Time	CPU Time:Idle	CPU Time:Poor	CPU Time:Ok	CPU Time:Ideal	CPU Time:Over	Wait Time
gen_graph	path-mpi.x	0.010s	0s	0.010s	0s	0s	0s	
genrand	path-mpi.x	0.010s	0s	0.010s	0s	0s	0s	
PMPI_Init	libmpi.so.12.0							0.000s

Figure 7: Most time consuming functions in the OpenMPI implementation.

3.2 Scaling

Similarly, strong and weak scaling studies were performed with the MPI parallelization.

Threads	Time (s)	Strong Scaling	Scaling Efficiency
1	0.0105219	1	100
2	0.00607085	1.733183986	86.65919929
4	0.00353408	2.977267068	74.4316767
16	0.00289488	3.634658431	22.7166152
24	0.00311399	3.378912585	14.07880244

Figure 8: Strong Scaling for 200x200 Element Graph, openMPI

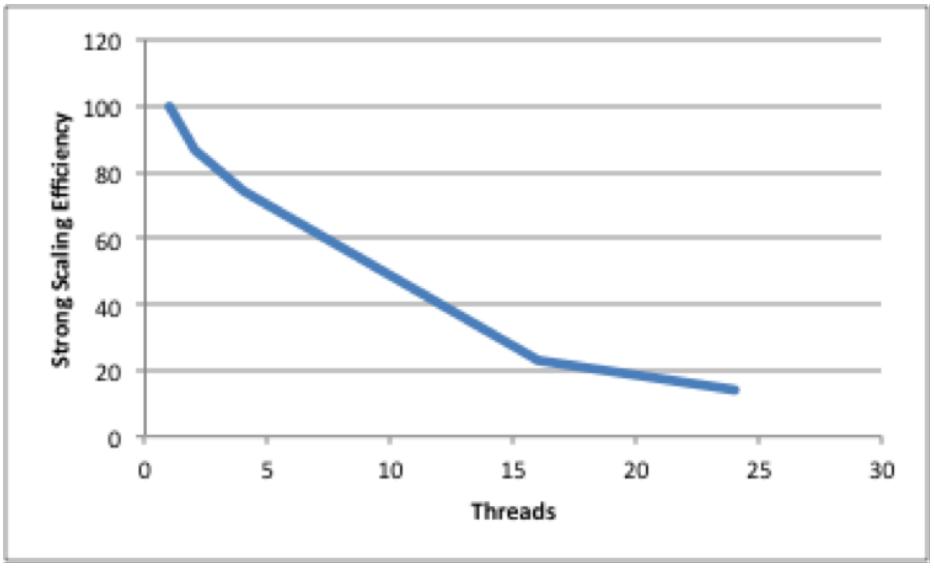

Figure 9: Strong Scaling Efficiency, openMPI

Overall, the current MPI parallelization method has better strong scaling than the openMP parallelization, although a similar domain decomposition for the openMP method would likely improve its' strong scaling characteristics. Weak scaling (which can be used to investigate the cost of communications) in both cases is similar,

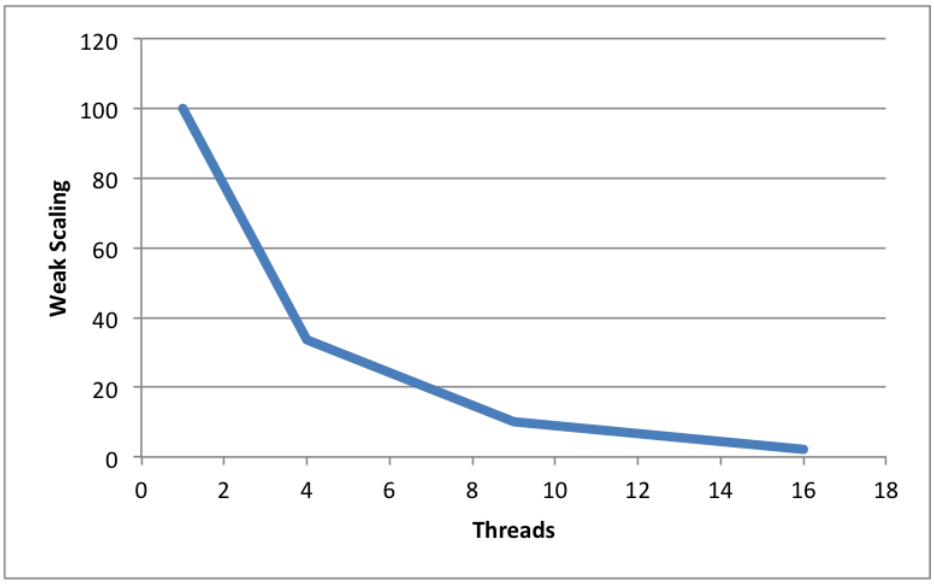

Figure 10: Weak Scaling (%), openMPI

with openMPI performing only slightly more poorly. Additional testing should be done to investigate scaling on multiple nodes, in which case openMPI is expected to outperform openMP significantly due to the cross-node communications.

4 Additional Changes

For the future, there are many different things we can try. As evidenced in the past, proper usage of compiler flags should be able to provide some speedups for the process. Additionally, another option we plan to try is blocking. On a conceptual level, for each block, the fastest paths would be calculated given each potential entry and exit point, which would be the ghost cells around each block. From there, one aggregated faster path through would be calculated from the combination of smaller paths. Finally, we hope to test the use of running the code on the Xeon Phis, for both the openMP and openMPI cases.