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 Baseline OpenMP Code

2.1 Profiling

A look into performance of the baseline 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 2.

Furthermore, openMP is reinitialized and the threads are split within each call to the function square, which is clearly not the most efficient method of paralellization.

2.2 Scaling

Strong scaling is used to investigate the possible speedup due to parallelization and 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:

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

and strong scaling efficiency, which is defined as:

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

```
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 2: Square function

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. This enables us to investigate the role of processor communications in the simulation time.

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

Both strong and weak scaling studies were performed for the openMP code. Strong scaling efficiency is investigated in place of strong scaling, which is normalized by the linear speedup. All plots shown use percentages.

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 3: Strong Scaling for 200x200 Element Graph, baseline openMP

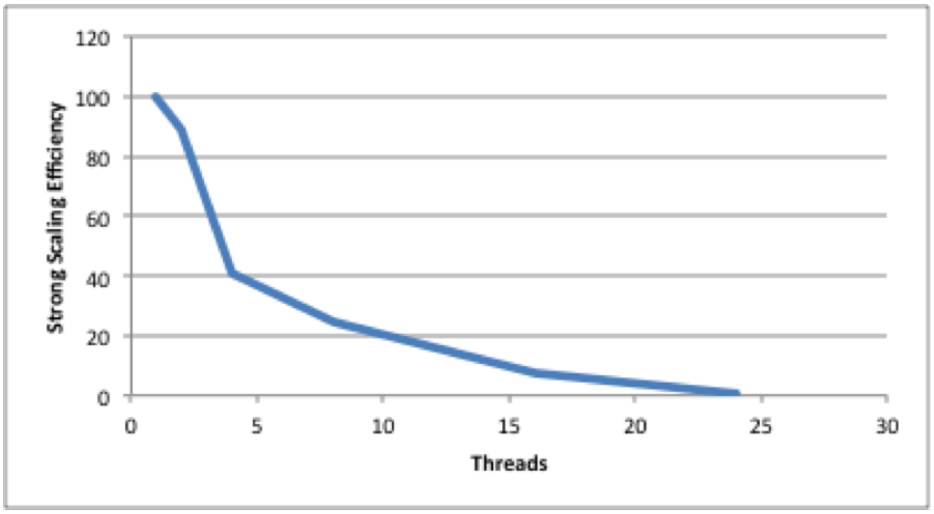

Figure 4: Strong Scaling Efficiency (%), baseline openMP

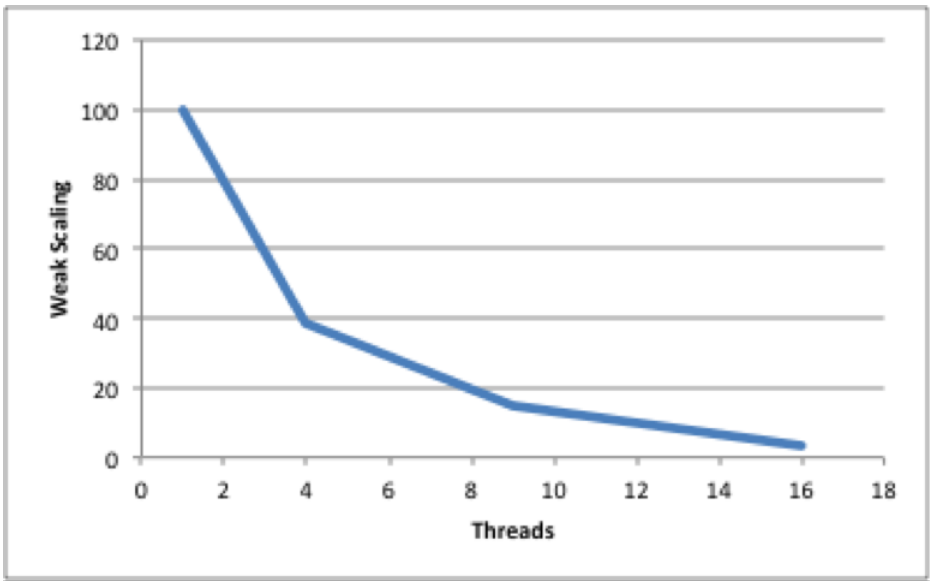

Figure 5: Weak Scaling (%), baseline openMP

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 (a basic domain decomposition). 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. This was all done in the main routine, so openMPI was only initialized the one time, as shown in the snippet of code from the main routine in Figure 6 and in the shortest paths routine in Figure 7.

```
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&irank);

imin =  (irank % npx)*(n/npx + 1);
imax = min(imin + (n/npx), n - 1);
jmin = floor((irank)/npx)*(n/npy + 1);
jmax = min(jmin + (n/npy), n - 1);

// Time the shortest paths code
if(irank == 0) t0 = MPI_Wtime();
//ok, now probably just each processor computes some shortest paths and then broadcasts
shortest_paths(n, l, irank, imin, imax, jmin, jmax);
```

Figure 6: MPI calls in MAIN

```
void shortest_paths(int n, int* restrict l, int irank, int imin, int imax, int jmin, int jmax)
{
    // Generate l_{ij}^0 from adjacency matrix representation
    infinitize(n, l);
    for (int i = 0; i < n*n; i += n+1)
        l[i] = 0;

    // Repeated squaring until nothing changes
    int* restrict lnew = (int*) calloc(n*n, sizeof(int));
    MPI_Allreduce(l,lnew,n*n,MPI_INT,MPI_MIN,MPI_COMM_WORLD);

    for (int done = 0; !done; ) {
        int idone = square(irank,imin,imax,jmin,jmax,n, l, lnew);
        MPI_Allreduce(&idone,&done,1,MPI_INT,MPI_MIN,MPI_COMM_WORLD);
        MPI_Allreduce(lnew,l,n*n,MPI_INT,MPI_MIN,MPI_COMM_WORLD);
    }

    free(lnew);
    deinfinitize(n, l);
}
```

Figure 7: MPI calls in shortest paths algorithm

3.1 Scaling

Strong and weak scaling studies were also performed with the MPI parallelization, similarly to those done for the baseline code in section 2.2.

Overall, the current MPI parallelization method has better strong scaling than the openMP parallelization. Weak scaling (which can be used to investigate the cost of communications) in both cases is similar, 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.

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

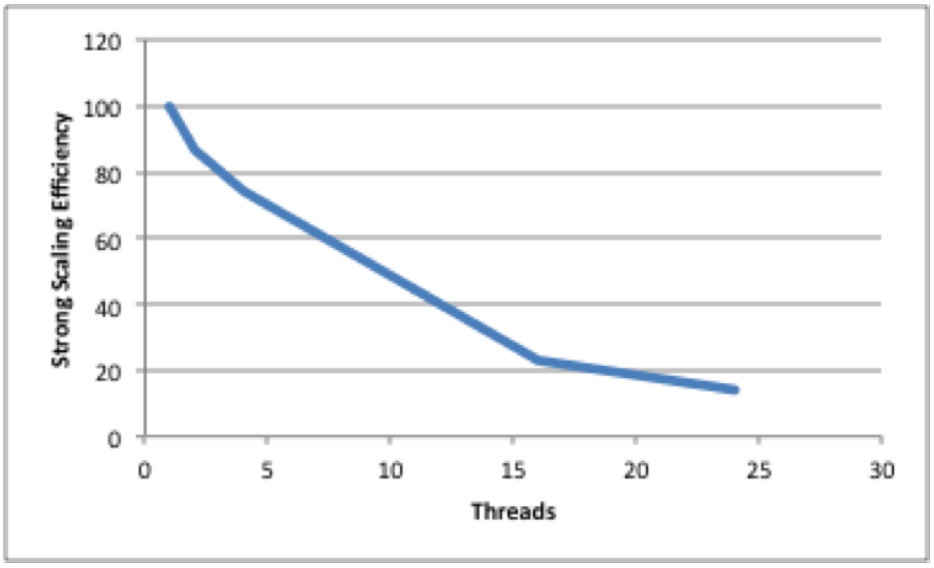

Figure 9: Strong Scaling Efficiency, openMPI

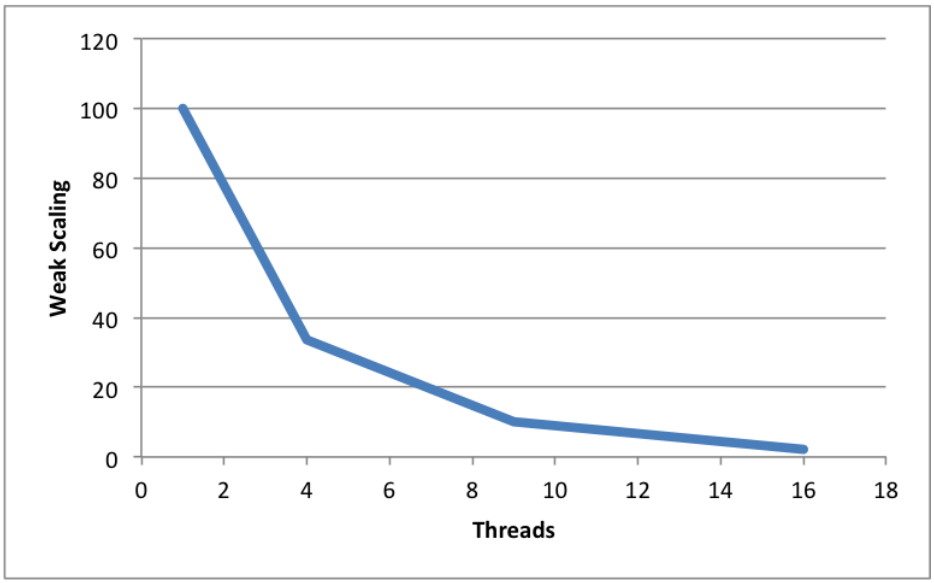

Figure 10: Weak Scaling (%), openMPI

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.