CS 5220 Floyd-Warshall Initial Report

Team 11

Guo Yu (gy63), Sania Nagpal (sn579), Scott Wu (ssw74)

1 Introduction

The purpose of this project is to analyze and improve performance of the Floyd-Warshall shortest path algorithm. The algorithm is very similar to matrix multiplication, with a branch in the kernel instead of a multiplication. We can apply the same blocking and cache locality tactics to the Floyd-Warshall algorithm. In addition, we will attempt to use OpenMP and MPI to further improve performance by utilizing multiple processors.

2 Profiling

2.1 Intel VTune

We used Intel VTune (amplxe-cl) to identify the runtime bottlenecks in the code. As per the report, the functions taking big chunks of the runtime are as follows:

Function	Module	CPU Time	CPU Time:Idle
square	path.x	43.688s	0s
__kmp_barrier	libiomp5.so	13.473s	0.100s
__kmpc_reduce_nowait	libiomp5.so	4.528s	0.020s
__kmp_fork_barrier	libiomp5.so	4.324s	3.011s
__kmp_join_call	libiomp5.so	0.040s	0.030s
__kmp_launch_thread	libiomp5.so	0.030s	0.020s
__intel_ssse3_rep_memcpy	path.x	0.030s	0s
fletcher16	path.x	0.030s	0s
gen_graph	path.x	0.010s	0.010s
pthread_create	libpthread-2.12.so	0.010s	0s
genrand	path.x	0.010s	0s

The function taking most of the time is the square function. Since there is not much of computation involved (one addition and one branch), the memory accesses should be taking most of the time. Hence, the code should be optimised by using good memory access pattern and decreasing the cache misses. Hence, in the initial report, we have mostly tried to tune the square function code. The OpenMP options barrier and reduce_nowait also seem to be taking huge time. It was also observed that this overhead increases further with increase in load.

3 Parallelization

3.1 Copy Optimization

In order to minimize cache misses, we tried to implement copy optimization in the square function by copying the matrix into a transposed block, giving us better cache locality. Below are the profiling results afterwards:

3.1.1 Profiling results

Function	Module	CPU Time	CPU Time:Idle
square	path.x	10.185s	0s
__kmp_fork_barrier	libiomp5.so	8.625s	7.042s
__kmp_barrier	libiomp5.so	4.340s	0.070s
__kmpc_reduce_nowait	libiomp5.so	1.238s	0.020s
shortest_paths	path.x	0.030s	0s
fletcher16	path.x	0.030s	0s
__intel_ssse3_rep_memcpy	path.x	0.020s	0s
genrand	path.x	0.020s	0.010s
__kmp_join_barrier	libiomp5.so	0.010s	0s
infinitize	path.x	0.010s	0s

We observed a large decrease in CPU time taken in the square function. Also we appear to have a decrease in some of the OpenMP overhead.

3.2 Blocking and Copy Optimization

Next, we applied blocking in the square function. Here, we reused the code written in Matrix Multiplication with a block size of 64. The profiling results after blocking are as follows:

3.2.1 Profiling Results

Function	Module	CPU Time	CPU Time:Idle
basic_square	path.x	9.794s	0.020s
__kmpc_barrier	libiomp5.so	5.191s	0.769s
__kmp_fork_barrier	libiomp5.so	4.426s	2.080s
__kmpc_critical	libiomp5.so	0.164s	0s
square	path.x	0.050s	0s
__intel_ssse3_rep_memcpy	path.x	0.050s	0s
fletcher16	path.x	0.020s	0s
gen_graph	path.x	0.010s	0s
deinfinitize	path.x	0.010s	0s
infinitize	path.x	0.010s	0s

Here we have basic_square (serial kernel) doing the main work. We also changed the done flag to be shared and modified with the critical pragma, which lowered overhead compared to reduction.

3.2.2 Vectorization Report

Aside from a couple initialization and output portions of the code, all the loops in the blocking function and kernel function were vectorized. Combined with the AVX2 compiler flag, we can utilize the vector instructions on each core.

3.2.3 OpenMP

The Floyd-Warshall algorithm, as with matrix multiplication, is an embarrassingly parallel problem. We can easily apply parallel for pragmas to fork tasks. Since our kernel is tuned for single core execution, we focus on parallelizing block by block rather than within the kernel. In our profiling results, we see that our wall clock times improve greatly, but we seem to spend quite a bit of time in an implicit barrier at the end of the parallel for.

3.2.4 Performance Model

Let P = number of processors
- N = size of the problem (number of vertices)
- T_A = time to perform the core atomic operation (add and branch)
- T_C = time to perform a copy
- T_B = overhead time for blocking
- T_P = overhead time for forking and barriers

We can represent the performance of the algorithm as an equation using these variables.

Serial Performance

$$N^3 \cdot T_A$$

Parallel Performance

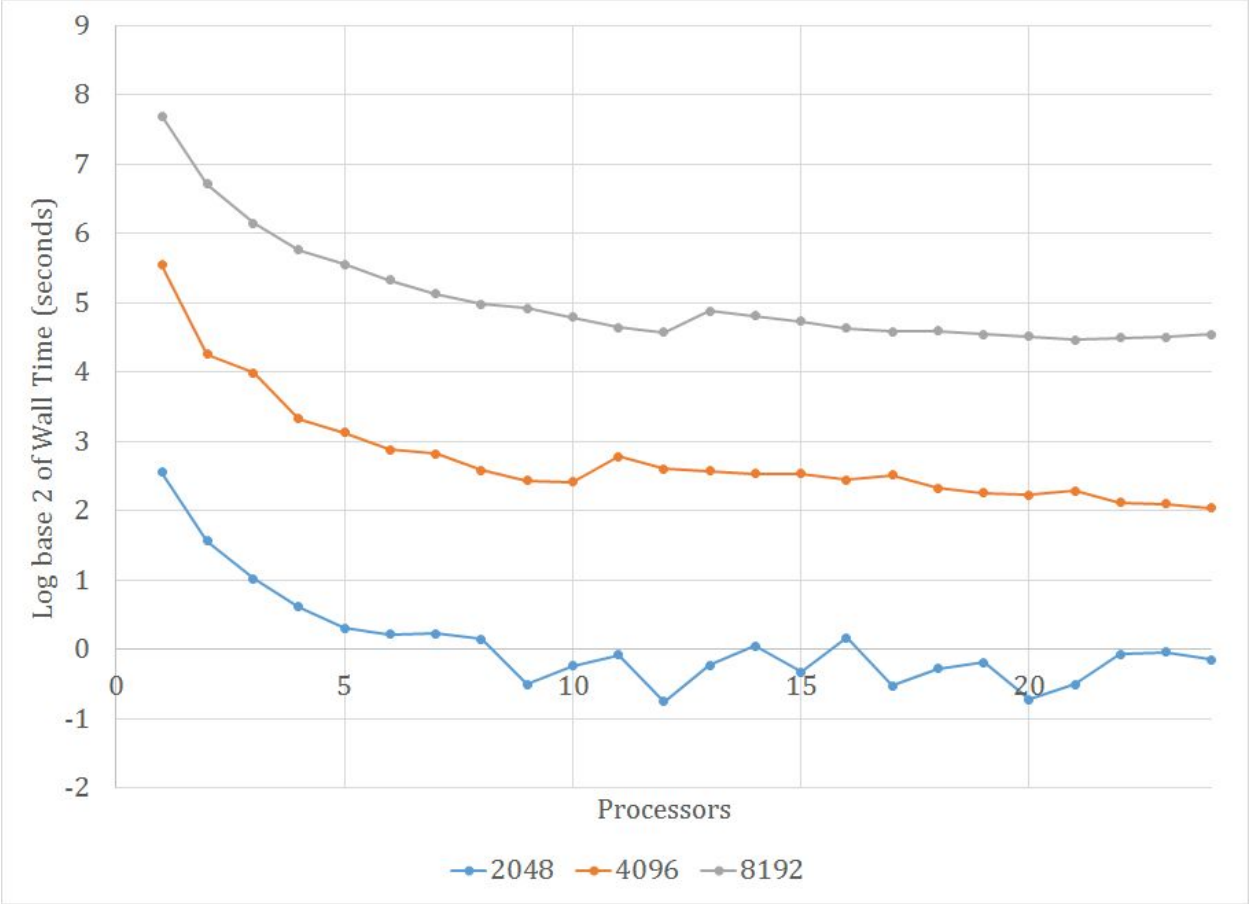
$$N^2 \cdot T_C + T_B + T_P + \frac{N^3 \cdot T_A}{P} + N^2 \cdot T_C$$

Speedup

$$\frac{N^3 \cdot T_A}{N^2 \cdot T_C + T_B + T_P + \frac{N^3 \cdot T_A}{P} + N^2 \cdot T_C} = \frac{T_A}{\frac{2 \cdot T_C}{N} + \frac{T_B + T_P}{N^3} + \frac{T_A}{P}}$$

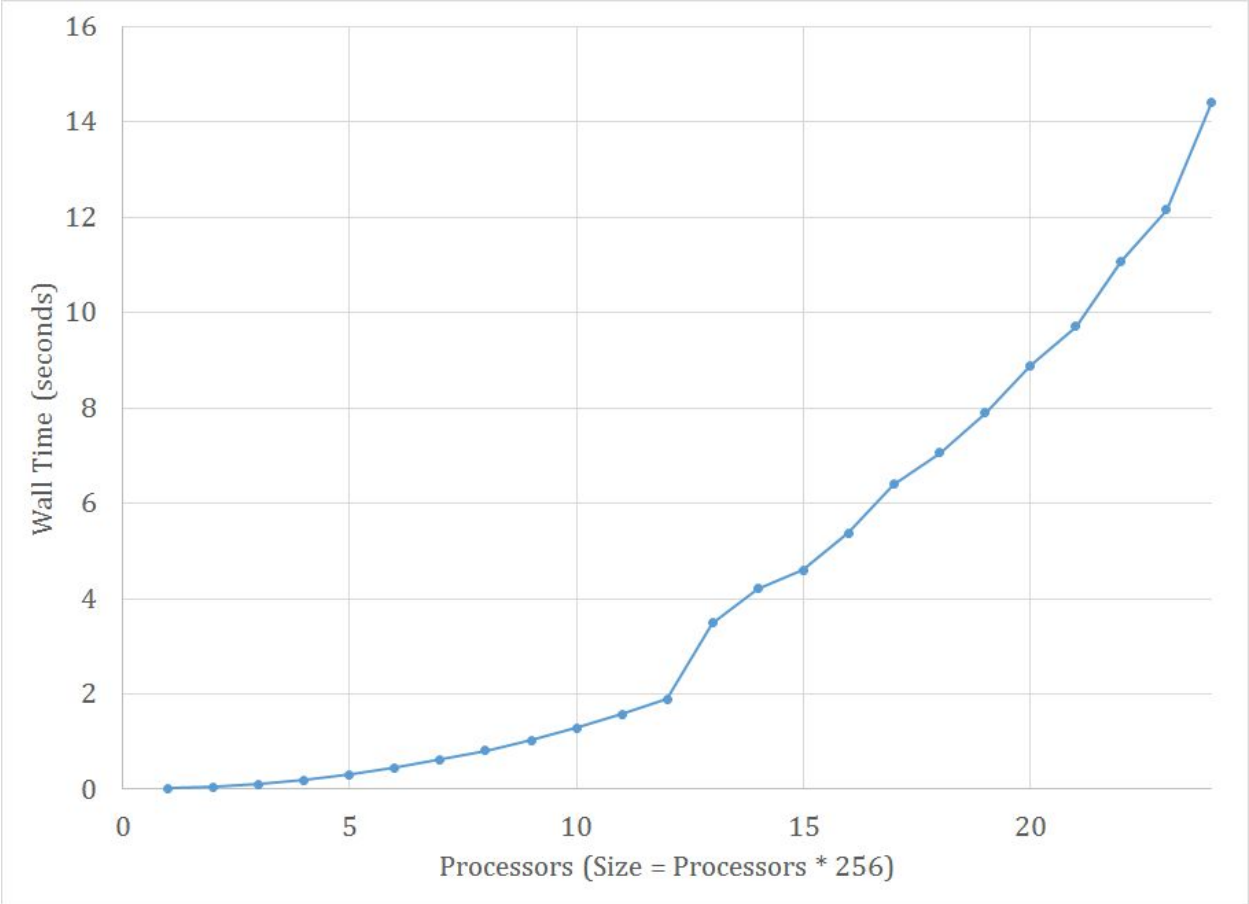
This model gives us a couple of ideas. First, as our problem size increases, the overhead from copying, blocking and OpenMP decreases. Second, we would ideally obtain a speedup of P as N goes to infinity.

3.2.4 Strong Scaling



In our strong scaling study, we consider three settings where the number of vertices are 2048, 4096 and 8192 respectively. We see there is an approximately exponential decrease of wall time against the number of processors. Later in stage 2 we will also study the plot of speed up against the number of processors.

3.2.5 Weak Scaling



In the weak scaling study, we set the number of vertices equal to 256 times the number of processors. We see that the wall time increases nearly-exponentially with the number of working processors. Later in stage 2 we will also study the plot of speed up against the number of processors in weak scaling.

3.3 MPI

We could not get MPI working on the cluster on multiple nodes nor the Xeon Phis. We also reasoned that if we were only using a single computer, message passing has more overhead than shared memory and OpenMP.

4 Next Steps

For the final iteration of the project, we will continue investigating MPI, and possibly some other solutions to message passing (unix sockets, network sockets, filesystem). Additionally we will try OpenMP again with the Xeon Phis.