CS 5220 Floyd-Warshall Final 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.

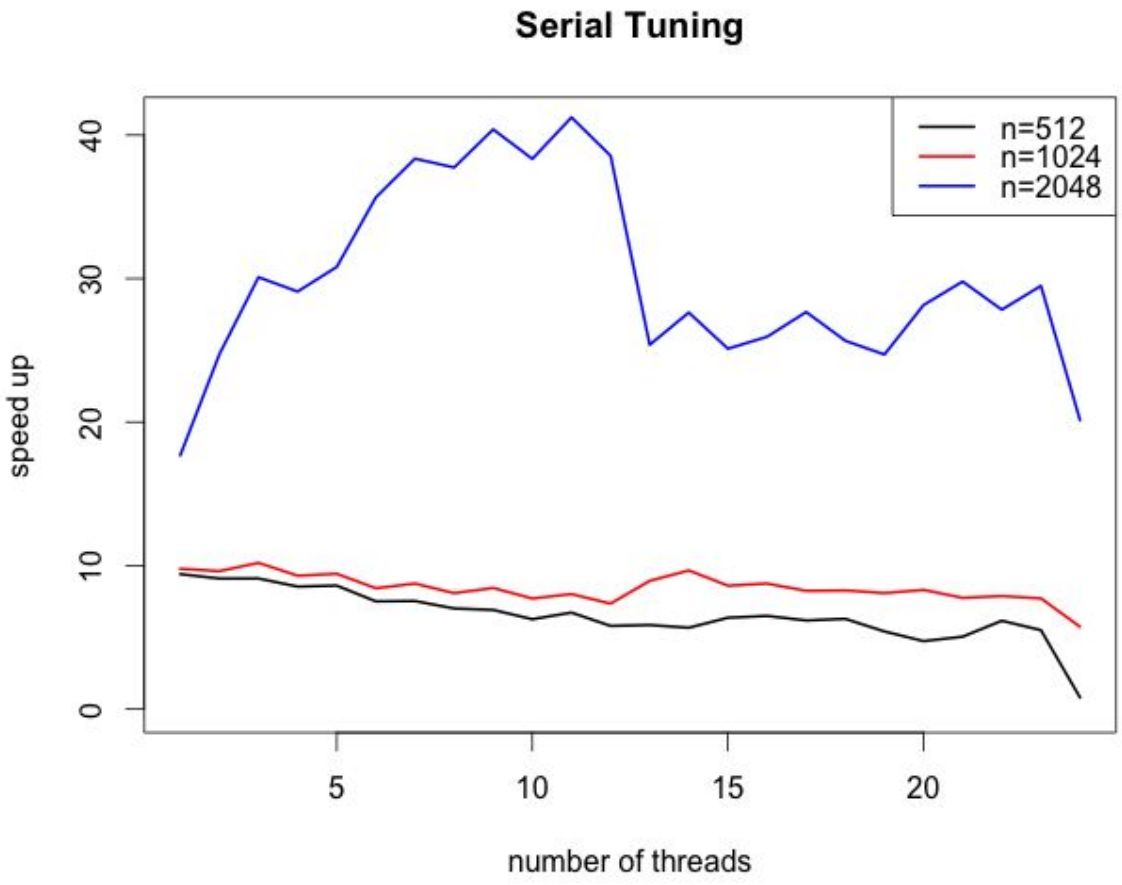
3.2.2 Vectorization Report

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.

The following plots shows the comparison results of the tuned version with OpenMP with the naive implementation with OpenMP (the originally given implementation). We observe a significant improvement of performance. In specific, for $n > 3000$, a naive implementation takes so long that the job got killed.



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

$$N^3 \cdot T_A$$

For the naive implementation:
Parallel Performance

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

Potential speedup

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

For the serial-tuned implementation:

Parallel Performance

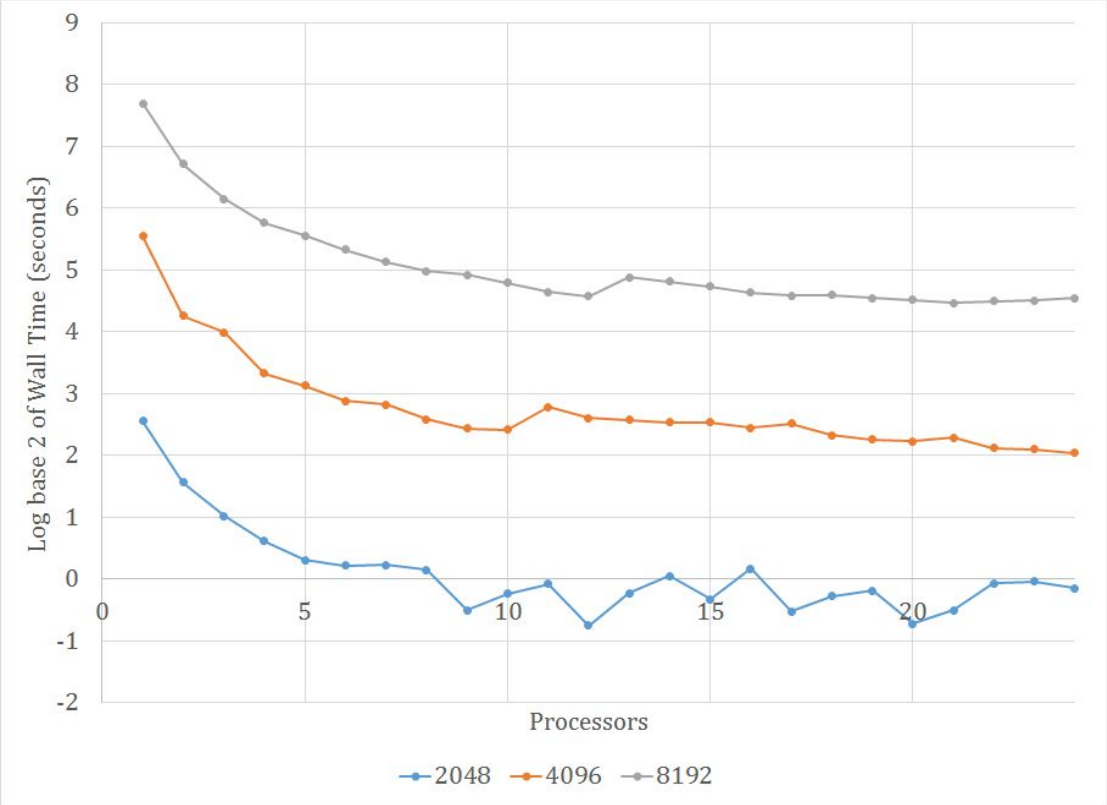
$$N^2 \cdot T_C + T_B + T_P + \frac{N^3 \cdot T_A}{P} + N^2 \cdot T_C$$

Speedup

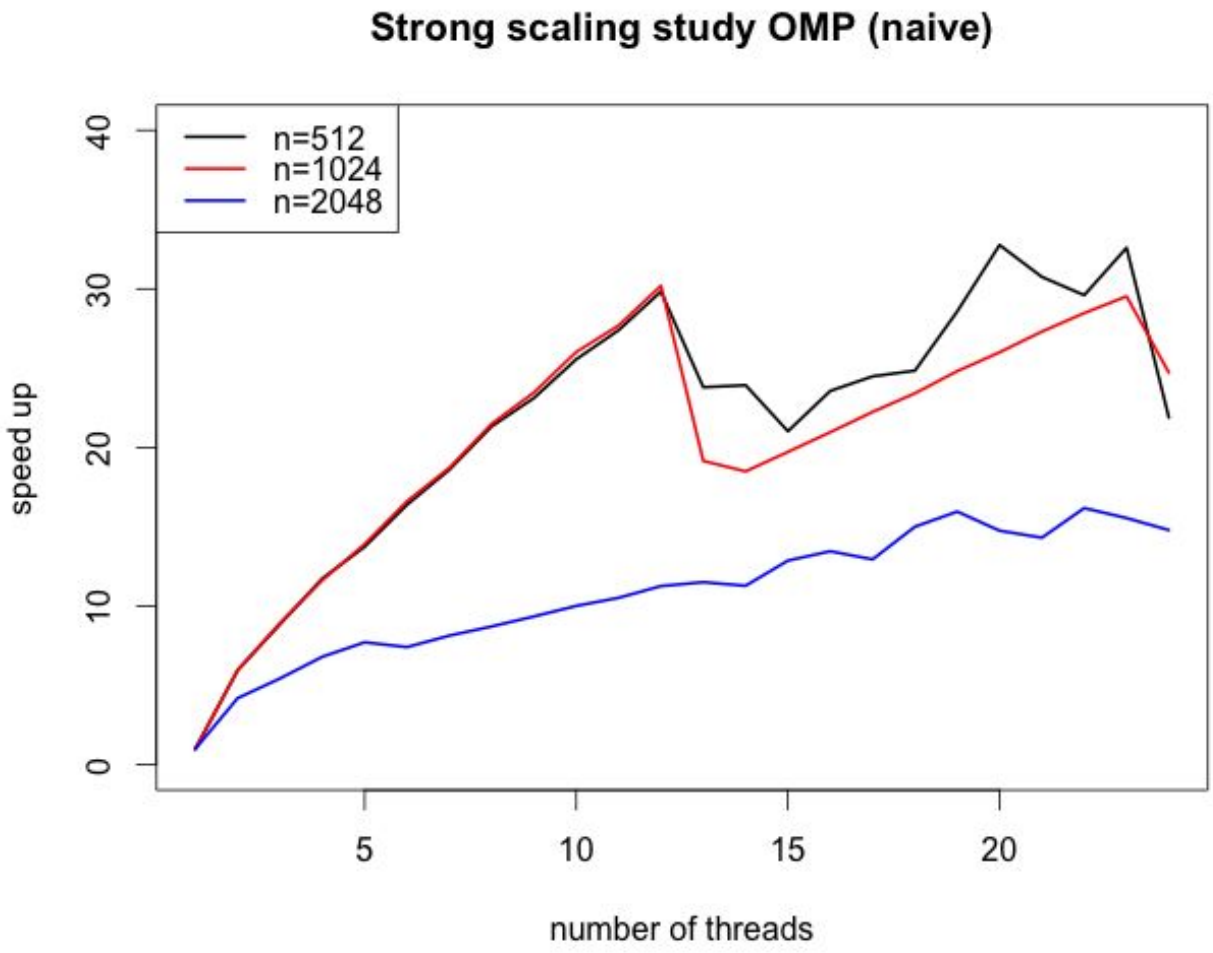
$$\frac{N^3 \cdot T_A}{N^2 \cdot T_C + T_B + T_P + \frac{N^3 \cdot T_A}{P} + N^2 \cdot T_C} = \frac{T_A}{\frac{2 \cdot T_C}{N} + \frac{T_B + T_P}{N^3} + \frac{T_A}{P}}$$

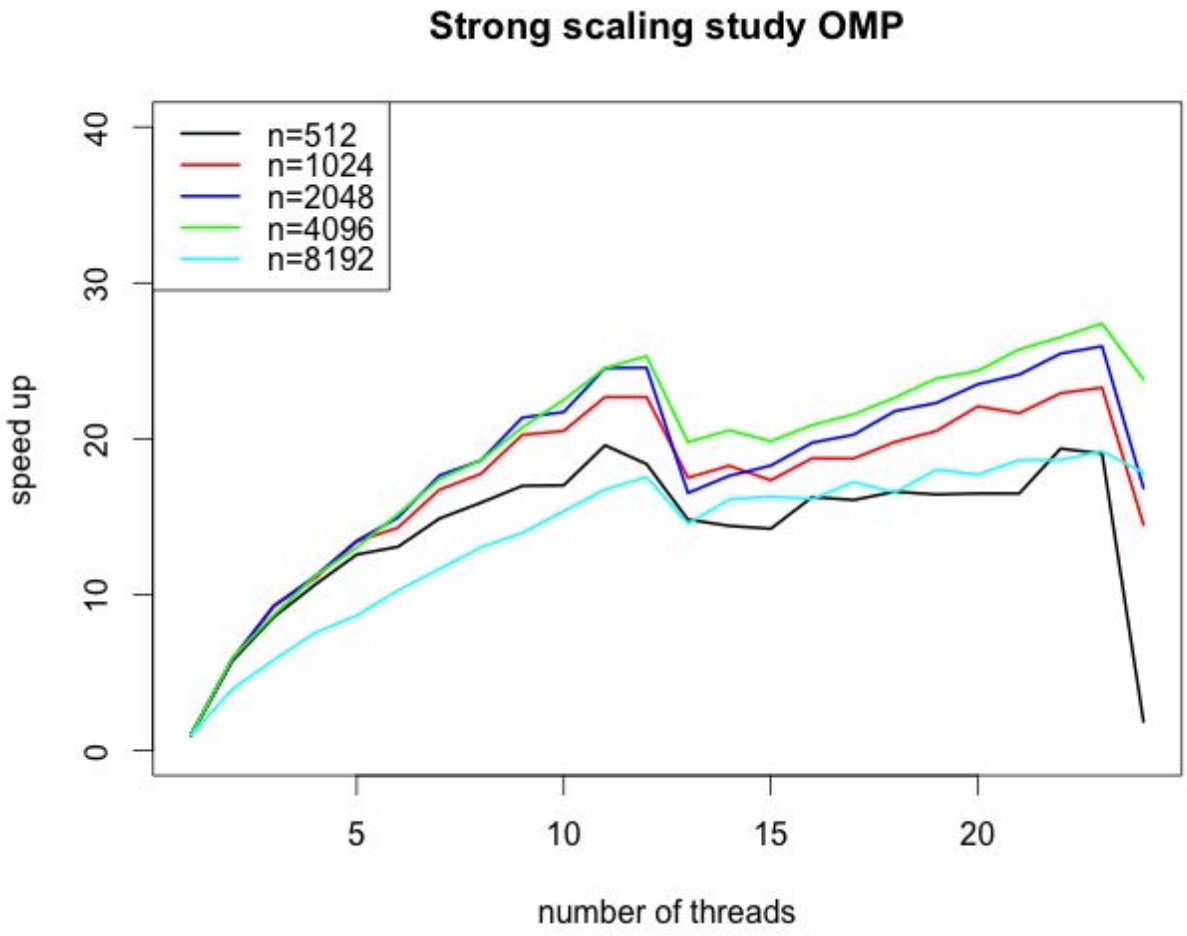
This model gives us a couple of ideas. First, as our problem size increases, the overhead from copying, blocking and OpenMP decreases. Second, we would ideally obtain a speedup of P as N goes to infinity. Third, ideally the naive implementation would give a better potential speed-up.

3.2.4 Strong Scaling



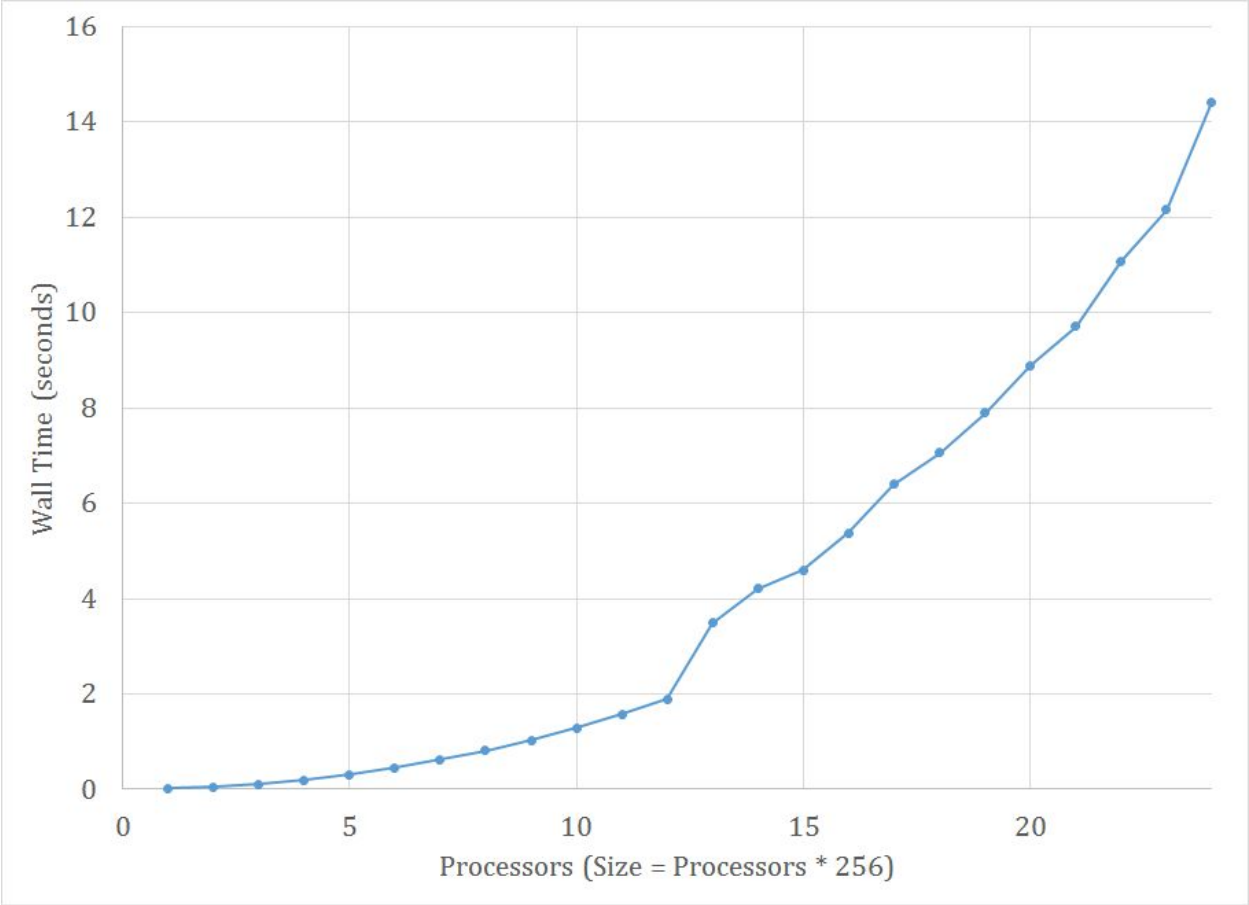
In our strong scaling study, we consider three settings where the number of vertices are 2048, 4096 and 8192 respectively. We see there is an approximately exponential decrease of wall time against the number of processors. We also studied the speed-ups of naive implementation as well as our tuned version against the number of threads using OpenMP.



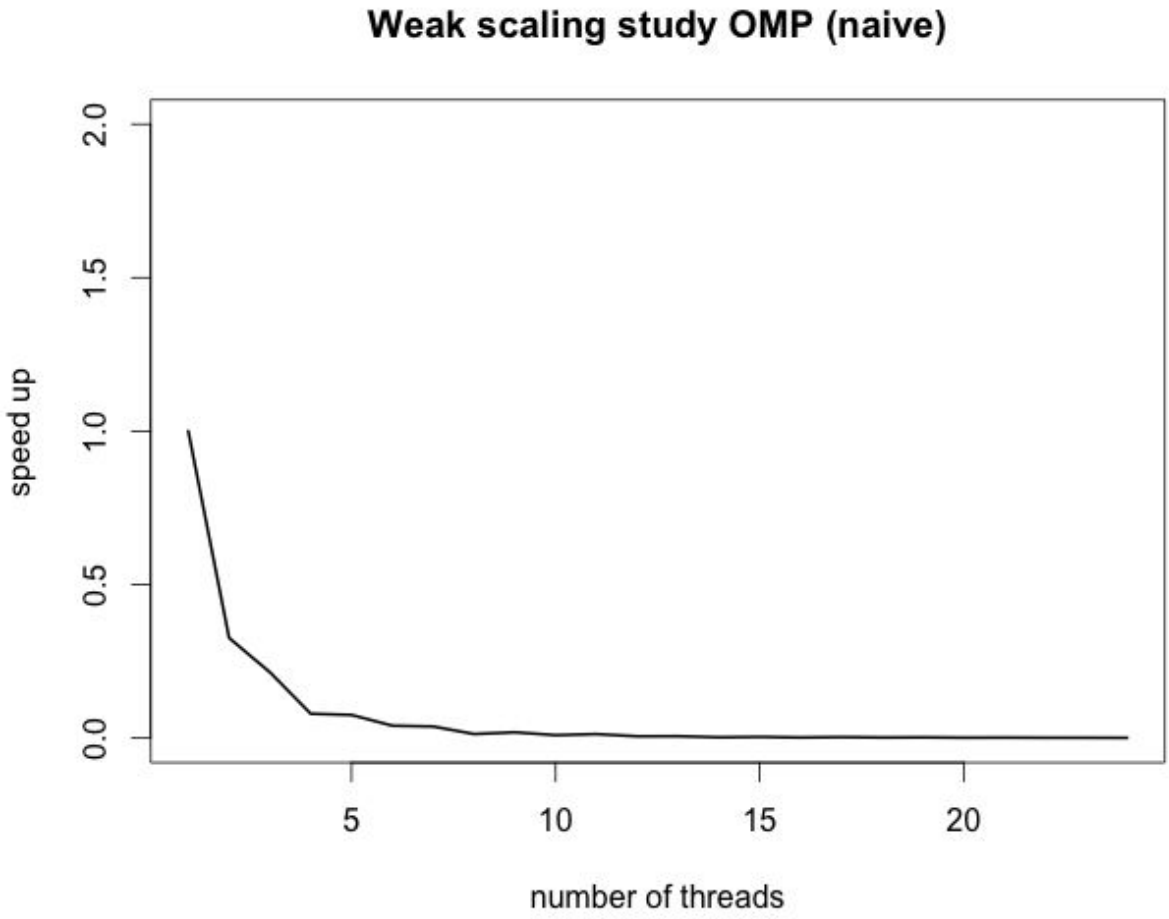


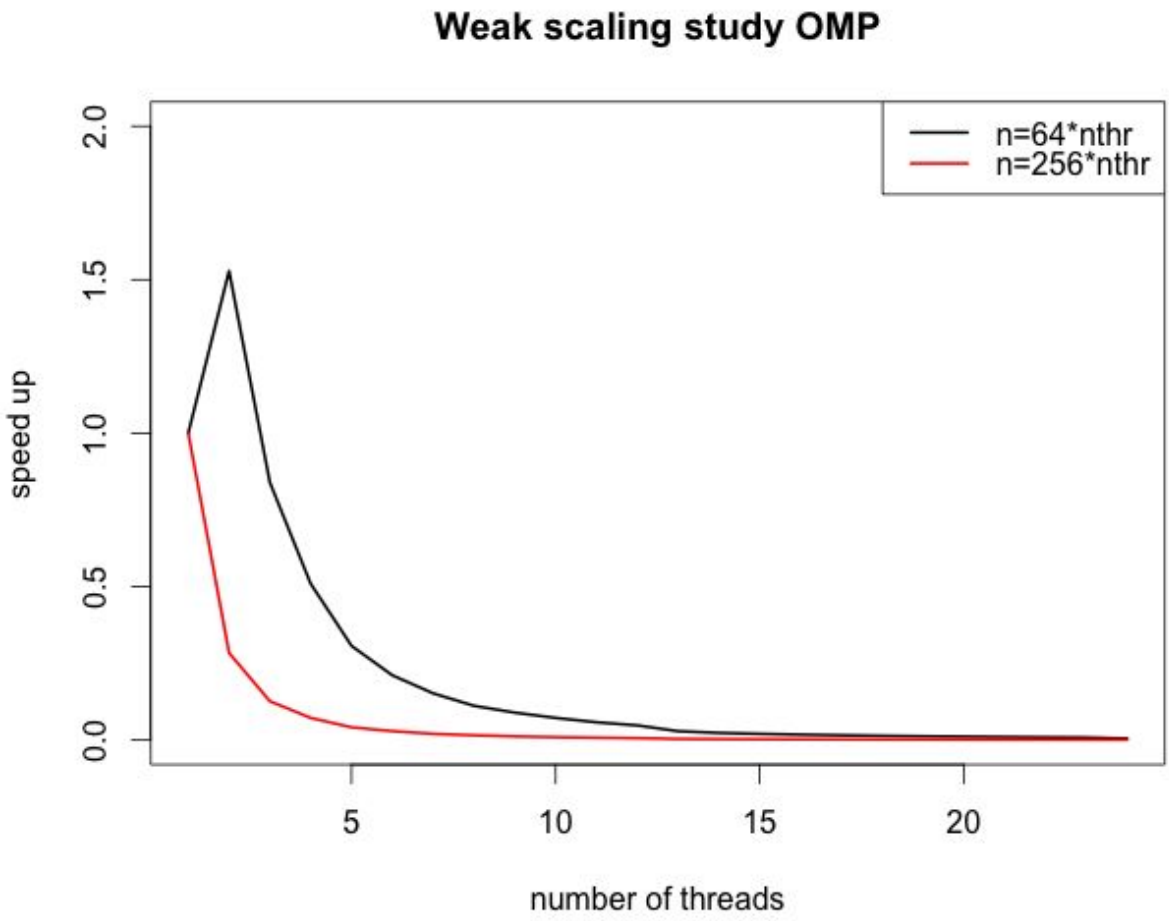
Comparing two plots, we found that the speed up patterns for both implementations of Floyd-Warshall algorithm are pretty similar. This observation verifies in some sense the performance models we established earlier in the following ways: First as the model predicts, the speed up of naive implementation is generally larger than the speed up of the tuned implementation. Second, as the formula which predicts the speed up shows, the pattern looks essentially like a linear function of number of threads.

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. The following plots gives the speed-ups plots of weak scaling. Again we both studied the naive implementation and our tuned version.


Weak scaling study OMP (naive)
speed up
0.0
0.5
1.0
1.5
2.0
5
10
15
20
number of threads



Unfortunately, the numeric results for weak scaling study do not explain our performance model very well. Note that in our setting where n = 64 * num_threads or n = 256 * num_threads, the performance model predicts an increasing speed up w.r.t. number of threads in a way that is similar to a quadratic function. However, the numeric results show decreasing pattern of speed up as the number of threads increases. This might results from the fact that our performance model is very simplified and many factors might be ignored.

3.3 Offloading

We tried to offload the parallel part of the Floyd-Warshall algorithm in OpenMP on to the Xeon Phis. We used a straightforward approach of offloading the whole shortest path computation to Phi nodes. However, there was no improvement to offloading the computation, either as a result of data transfer overhead or inefficient work loads. A sample of results is shown below:

n	OpenMP	Offload + OpenMP
2047	0.879967	1.40739
2048	0.900537	1.38295
2049	0.835211	1.50368
4095	4.28529	7.08943
4096	4.11324	6.99912
4097	5.10643	7.23956
8191	22.6708	37.2516
8192	23.3544	37.0134
8193	31.646	50.8505

3.4 MPI

On a single node, we are limited to 24 threads with OpenMP. Although offloading is one option, it did not give very good results. The other option is to use multiple nodes and pass data between them using MPI.

To run an MPI program with multiple nodes, we first compile with `mpicc` instead of `icc`.

In the PBS script, we specify the number of nodes and processors we want to use:

```
# Gives 4 x 24 = 96 processors available for MPI
#PBS -l nodes=4:ppn=24
```

Finally, we use `mpirun` to actually execute the program.

```
mpirun -n 82 ./path_cannon.x -n 4000
```

One problem we ran into was that mpirun used ssh to distribute tasks. Using mpirun either requested login info, or asked if the host was trustable. Sometimes it would result in a neverending loop. To fix this problem, we set up passwordless login on the head node itself (which would propagate to the other nodes on the network file system). Within an interactive qsub shell, we also logged into each of the other nodes once to trust the fingerprints. After these steps, we could run mpirun with multiple nodes without a problem.

3.5 Cannon's algorithm

A naive use of MPI on the Floyd-Warshall algorithm would be imitating the shared memory construct of OpenMP. However, as the size of the problem increases, the amount of data we must send grows into the hundreds of megabytes. One important thing to consider is the amount of communication between nodes and processors, since that could result in a large overhead. To remedy this, we looked at Cannon's algorithm.

Cannon's algorithm is a distributed algorithm for two-dimensional matrix multiplication, boasting relatively low communication overhead. Since the algorithm assumes we have an efficient dgemm kernel, we can easily adapt it to work with the Floyd-Warshall algorithm. Given p^2 processors, we give blocks A and B of the matrix to each processor. The processors compute retain and compute the result C. Then they send the blocks A and B to adjacent matrices. Block A is sent left and B sent up. If the initial A and B blocks sent are skewed in a staircase fashion, after p steps the accumulated result block C will contains the result for the corresponding part of the matrix.

One caveat of Cannon's algorithm is that it requires a square number of processors. In our implementation, we used $p^2 + 1$ processors. The extra processor served as a root node which handled the input / output and logistics. Another implementation difficulty is that processors must send and receive data at the same time. When done incorrectly this may result in a deadlock or unsent data, and thus incorrect results.

Once again we did not see any improvement, and in fact rather unpredictable result:

n	OpenMP	Offload + OpenMP	MPI
2047	0.879967	1.40739	6.41029
2048	0.900537	1.38295	3.1904
2049	0.835211	1.50368	3.06637
4095	4.28529	7.08943	51.461
4096	4.11324	6.99912	56.4659
4097	5.10643	7.23956	74.9398
8191	22.6708	37.2516	
8192	23.3544	37.0134	
8193	31.646	50.8505	

4 Last Thoughts

Our OpenMP results turned out well, but offloading and MPI were tricky to work with and did not give the performance we expected. Those are two areas to put more analysis and tuning into because theoretically they should enable us to achieve larger degrees of speedup.

Something we have not touched at all was the probability argument. Varying the probability to different ends of the spectrum, we could have studied the difference between very sparse and very dense graphs.