Project 3 - All-Pairs Shortest Paths

Team 1 – Bob Chen (kc853), Nimit Sohoni (nss66), Xiangyu Zhang (xz388)

1 Introduction

In this project our object was to optimize an implementation of the *Floyd-Warshall algorithm* to compute all pairwise shortest path lengths in a graph. We were provided with a reference OpenMP implementation with naive approach for the core. We started with profiling the given code and analyzing the most significant bottleneck to be accessing individual elements in the matrix. From there we employed strategies from matrix computation project including blocking, vectorization and copy optimization to enhance the performance.

By the end of the first stage we were able to significantly minimize the cost on accessing elements. However we did not intensively investigated the incorporation of OpenMP or MPI, which in theory should further optimize the performance. We plan to continue tuning the code with those approaches in the next stage of this project.

2 Profiling and Analysis

We used `amplxe` tool to profile the provided basic implementation of Floyd-Warshall algorithm. The hotspots report shows that the `square` function is the most expensive step in the computation

Function	CPU Time (s)
`square`	54.854
`__kmp_barrier`	10.136
`__kmpc_reduce_nowait`	4.940

Here we only listed the slowest three functions, two of which are OpenMP calls. Looking at the detailed profiling of `square` function:

Lines in `square`	CPU Time (s)
47 `#pragma omp parallel for shared(l, lnew) reduction(&& : done)`	
48 `for (int j = 0; j < n; ++j) {`	
49 `for (int i = 0; i < n; ++i) {`	
50 `int lij = lnew[j*n+i];`	0.040
51 `for (int k = 0; k < n; ++k) {`	8.249
52 `int lik = l[k*n+i];`	25.808
53 `int lkj = l[j*n+k]`	
54 `if (lik + lkj < lij) {`	3.321
55 `lij = lik+lkj;`	
56 `done = 0;`	4.437
57 `}`	
58 `}`	
59 `lnew[j*n+i] = lij;`	
60 `}`	
61 `}`	

Notabaly the most costly step is line 52 when retreiving the `lik` element from the matrix. We find this scenario quite falimiar as this happened in the matrix computation project. It can also be noticed that the cost on OpenMP should happen in this block as well since the barrier synchronization and reduction happen at the end of for loops. We suggest that after minimizing the computational cost within the core of this nested for loops, the cost of OpenMP should be ideally significantly reduced as well.

3 Matrix Access Strategies

In order to minimize the cost of accessing individual elements in a matrix, we bring up the strategies we used in the matrix computation project, namely blocking, vectorization and copy optimization. We have tried all three approaches in the first stage.

3.1 Blocking

The L1 cache of the computing nodes of the cluster is 64 KB large. So the largest matrix it can accommodate is

$$\sqrt{\frac{64\text{KB}}{4\text{B}}} = 128$$

Thus in our implementation, we divide the given matrix into blocks of 128 to reduce the cost of memory access.

However we found that this implementation does not enhance the performance as much as we expected. On the contrary it slows down the computation due to communication between blocks. We believe this would be a good scenanio where either OpenMP or MPI should make contribution to, however due to time constraints we have not finished tuning yet.

3.2 Vectorization and Copy Optimization

Another important strategy we employed is the copy optimization and vectorization. Knowing that for the computing nodes on cluster, vector size for each register is 256 bits long and there are 4 vectors per core, we align columns of the matrix into vectors of 64 Bytes. To enforce the alignment and enforce the column access we first transpose the matrix then use the `__assume_aligned` function.

With the copy optimization and pre-tuned OpenMP implementation, we are able to significantly accelerate the computation, signifying that this approach works well in this set-up.

3.3 Profiling for Different Approaches

We have profiled three experiments: the blocking approach which is not parallel with OpenMP; the copy optimization and vectorization on one thread; and the copy optimization and vectorization on 24 threads. In comparison, note that in the basic implementation, `square` function takes 54.9 seconds, in which the slowest step takes 25.8 seconds in total.

The following are a few highlights from the performance analysis.[1]

Blocking Approach without OpenMP/MPI[2]

Function	CPU Time (s)
`basic_square`	6.084
———	
Slowest Steps	
57 `int lik = l_transpose[i*BLOCK_SIZE + k];`	3.298
58 `int lkj = l[j*BLOCK_SIZE + k];`	2.337

Vectorization and Copy Optimization Using One Thread[3]

Function	CPU Time (s)
`square`	3.587
———	
Slowest Steps	
188 `lij = lik + lkj;`	1.798

Vectorization and Copy Optimization Using 24 Threads[4]

Function	CPU Time (s)
`square`	6.675
`__kmp_barrier`	5.336
`__kmp_fork_barrier`	2.108
———	
Slowest Steps	
188 `lij = lik + lkj;`	3.784

It can be observed that both three approaches enhance the performance, while the vectorization and copy optimization have the best performance. Note that we still have not intensively tune the parallel portion of the code and we do expect to see better results once we have better tuned OpenMP and MPI implementation.

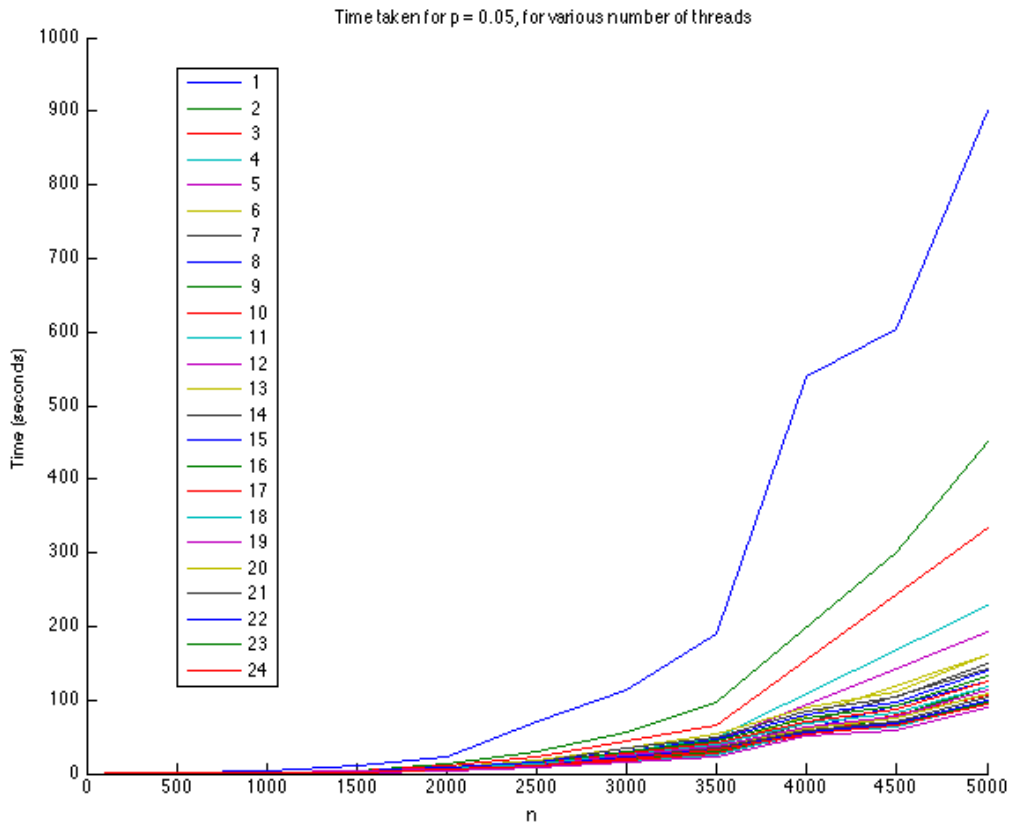
[1]To access the full report, check the Github repository `report` folder.
[2]See the code `path.c`, function `square_block` and `basic_square`
[3]See the code `path.c`, function `square`. To run the code, use flag "-t 1".
[4]See the code `path.c`, function `square`. To run the code, use flag "-t 24".

4 MPI

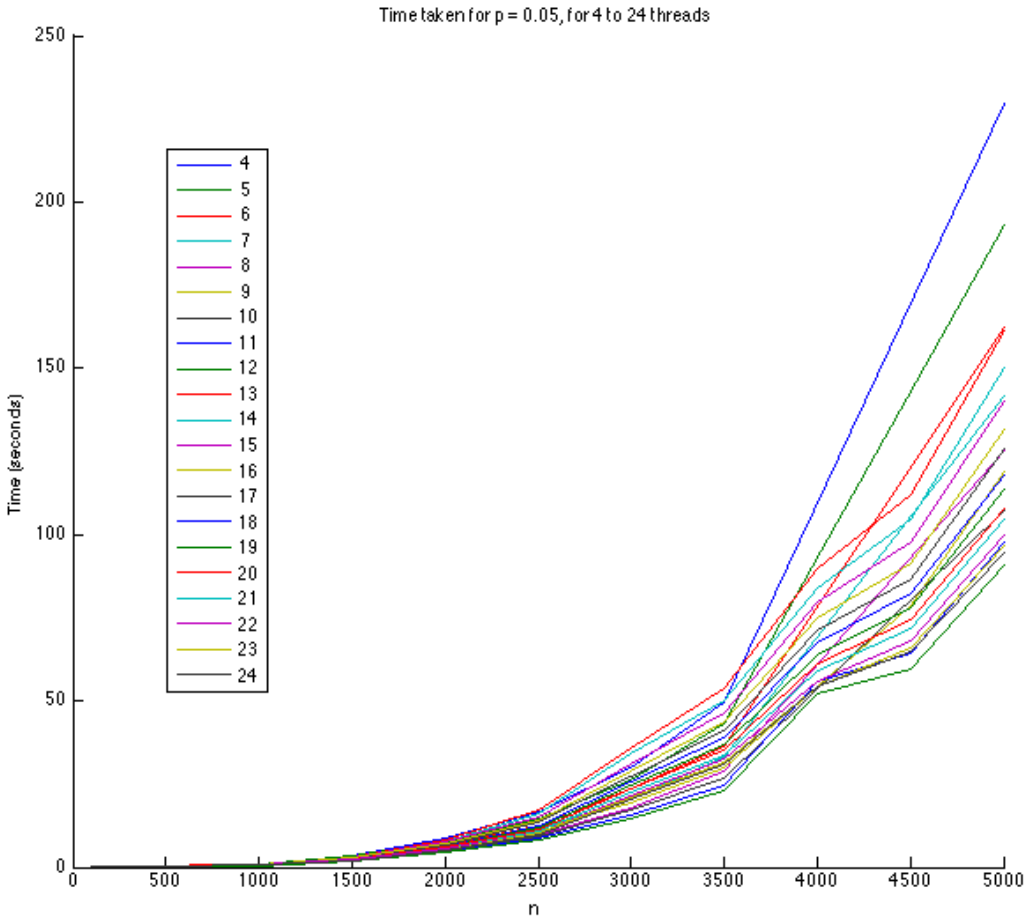
We also began to work on rewriting the code in an MPI fashion. We eventually decided to just focus on parallelizing the repeated squaring step with MPI, as the remainder of the steps run only once (gen_graph) and/or are much less costly. Each thread is assigned a block of columns to handle. In the square function, the thread multiplies its assigned columns by the entire L matrix on the left and, checks if the result is unequal to the result from the previous iteration, as in the original code. If any of the entries are unequal to the previous iteration, done is set to false. At each loop iteration, the results of 'done' from each thread are ANDed together. The blocks from each thread are gathered back into the new L matrix. We used the MPI_Allgatherv function because it is possible that n is not evenly divisible by the number of threads, in which case the leftover columns are assigned to the last thread.

The code for the MPI version is in `https://github.com/nimz/path/blob/master/path.c`

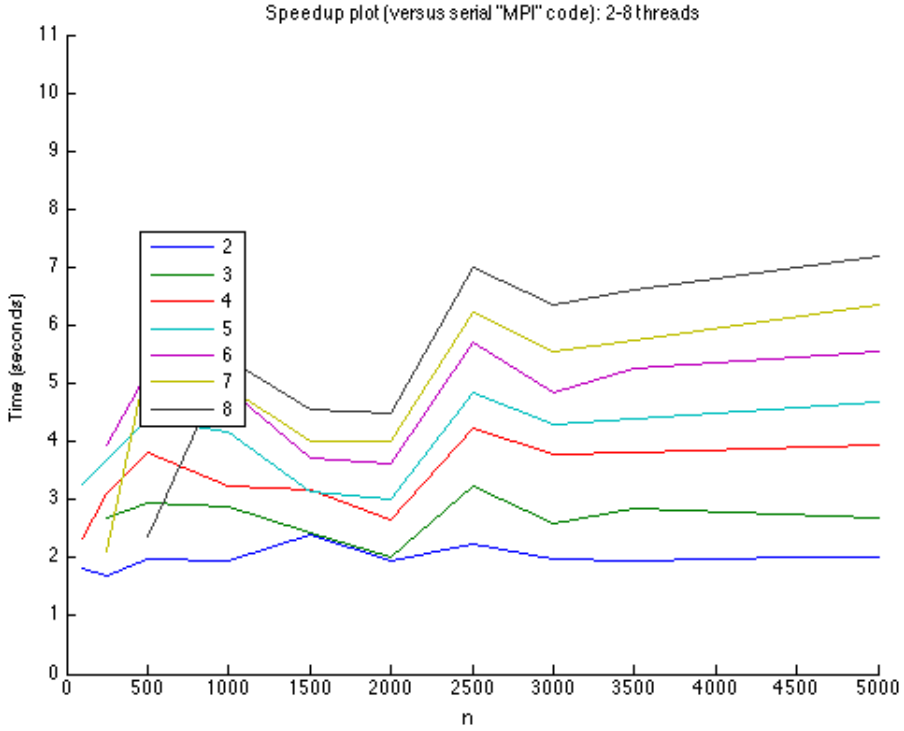
Timing results for varying number of threads:

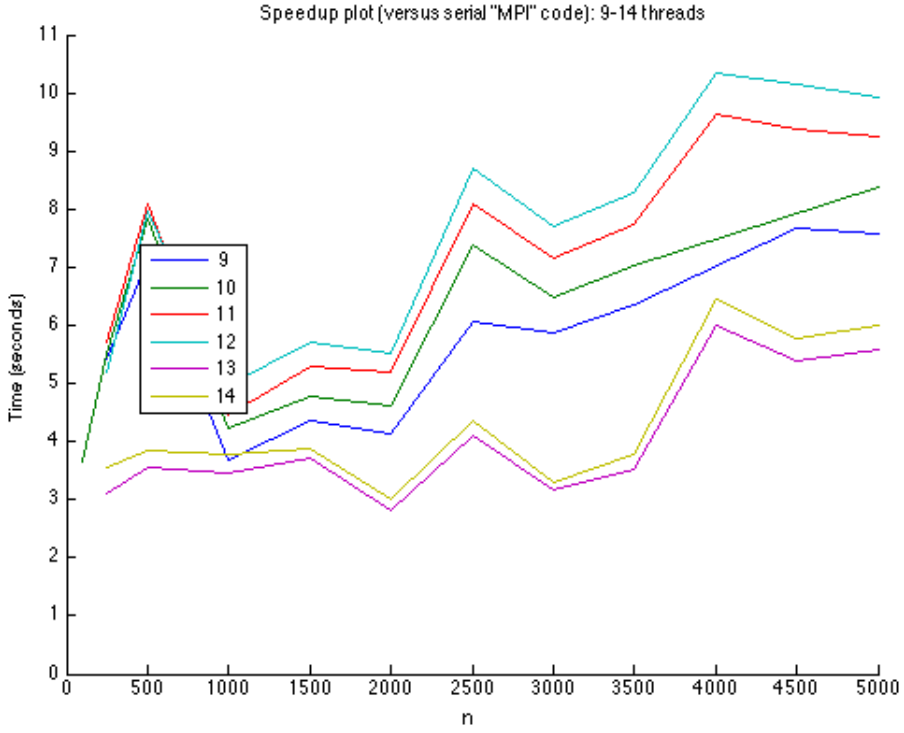


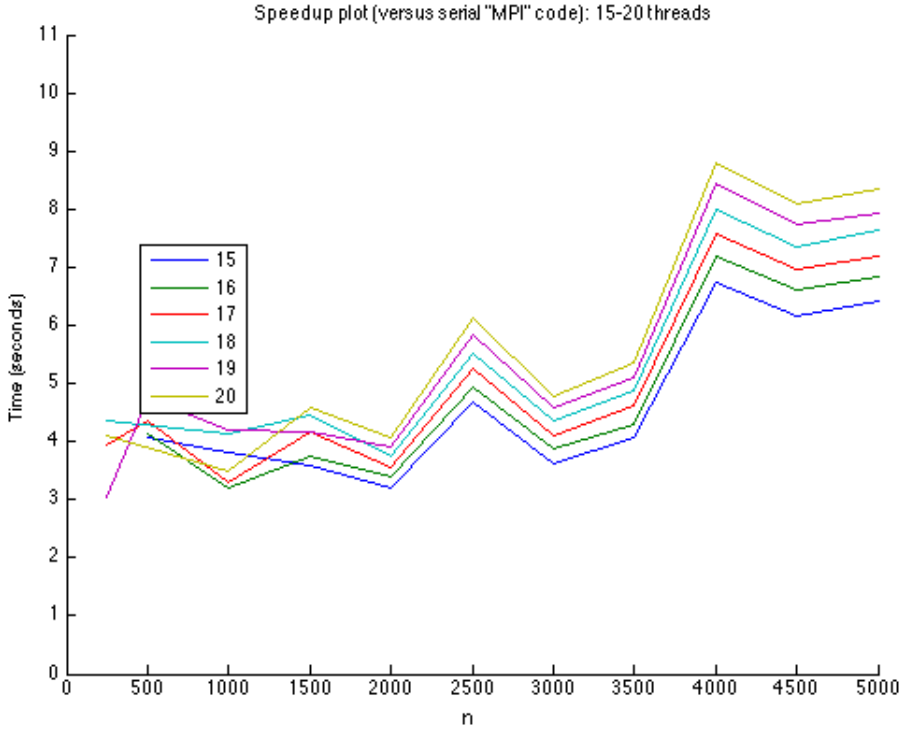
Timing results showing only with 4 to 24 threads for better visibility:

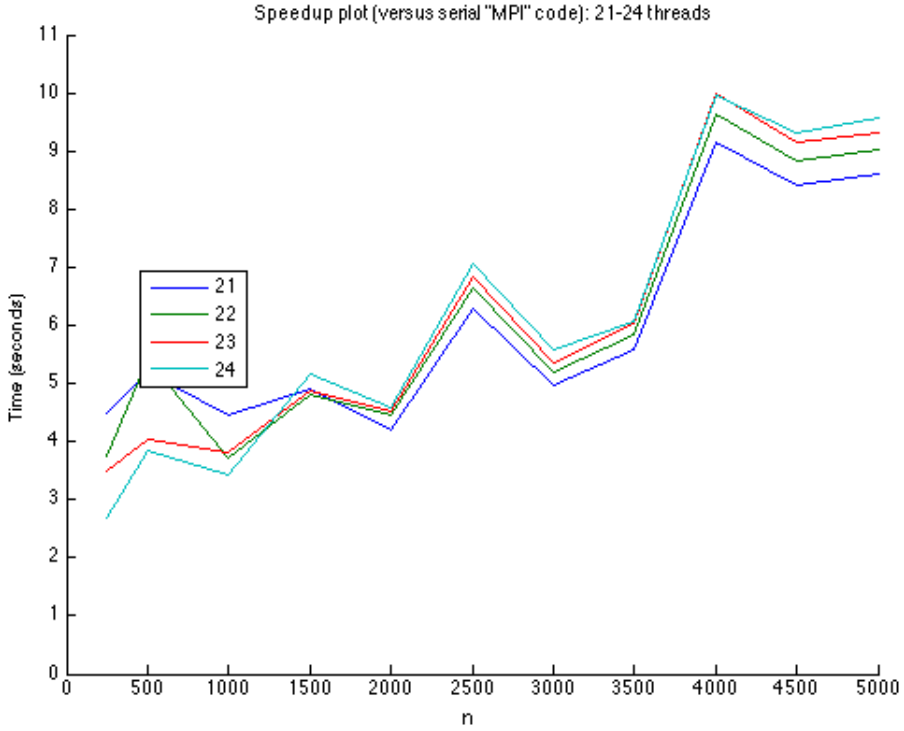


Speedup plots (split into several for readability):


Speedup plot (versus serial "MPI" code): 2-8 threads
Time (seconds)
n
2
3
4
5
6
7
8
0
1
2
3
4
5
6
7
8
9
10
11
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000


Speedup plot (versus serial "MPI" code): 9-14 threads
Time (seconds)
n
9
10
11
12
13
14
0
1
2
3
4
5
6
7
8
9
10
11
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000





Unsurprisingly, as we increase from 1 to 12 threads the speedup keeps increasing. However, once we go from 12 to 13 nodes we see a significant drop in speed. This is

actually also unsurprising, because we have gone from 1 to 2 different chips (we have 12 cores / threads per chip), so there is communication overhead. After this, however, speedup continues as we increase the number of threads, as expected. With all 24 threads we get around 5x-10x speedup depending on the size of the problem.

5 Future Work

There are a few of things we will be doing in the next stage.

1. Further implement OpenMP and MPI in our code and tune it to give better performance.

2. Once the parallel is well set up, we will do the strong scaling and weak scaling experiment to examine the speedup of our code.

3. We only briefly adapted the copy optimization in our blocking approach without further polishing. We will introduce parallelism to this strategy.

4. We have tried a few compiler flags but did not observe significant change to the performance, but we are going to investigate that as well.

6 Acknowledgement

Part of the work referenced our work in *Project 1 - Matrix computation* shared by Sijia Ma (Xiangyu's partner in Project 1) and Amiraj Dhawan (Bob and Nimit's partner in Project 1). Scott Wu's Project 1 report also provided inspiring ideas to this work.