# CS 5220 Homework 3 Initial Report

Dylan Foster (djf244), Saul Toscano-Palmerin (st684), Vikram Thapar (vt87)

November 10, 2015

## 1 Introduction

We sought to optimize the performance of a parallel implementation of the Floyd-Warshall all-pair shortest paths algorithm. This is a simple combinatorial algorithm for finding the shortest path between every pair of vertices in a directed graph. The algorithm has a data access pattern identical to that of square matrix multiplication, and thus is very amenable to optimization and parallelization. We profiled a baseline implementation of the algorithm in which the main loop is parallelized using OpenMP, then developed our own parallel implementation based on the MPI framework.

## 2 Profiling

As an initial step, it is useful to understand how the performance of the OpenMP implementation grows with the input size in practice, as this can inform our intuition regarding, say, whether communication costs begin to dominate runtime for certain input sizes. We ran the OpenMP implementation with a range of input sizes between 1000 and 10000. The results are plotted in Figure 2.

The optimization flags used in the baseline implementation (as well as our MPI implementation) are:

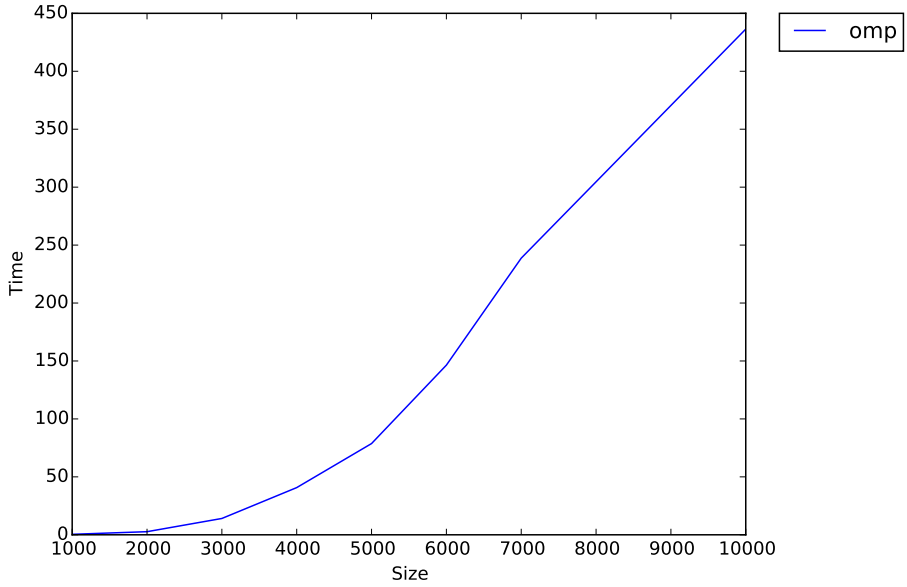
```
-O3 -no-prec-div -xcore-avx2 -ipo -qopt-report=5
-qopt-report-phase=vec-qopt-report=5 -qopt-report-phase=vec
```

We analyzed the OpenMP implementation's performance with the Intel VTune Amplifier tool. Because advanced hotspot collection is not currently working, we used the basic hotspot tool instead. The results are shown in Figure 1. Clearly most of the computation time is spent in the `square` function, but this analysis shows that a sizable chunk of this time is spent at the barrier. In some sense this is not surprising because the OpenMP implementation just parallelizes the outermost loop of the Floyd-Warshall computation.

Figure 1: VTune basic hotspot analysis of OpenMP implementation.

| Function | Module | CPU Time | CPU Time:Idle | CPU Time:Poor | CPU Time:Ok | CPU Time:Ideal | CPU Time:Over | Spin Time | Overhead Time:Other |
|---|---|---|---|---|---|---|---|---|---|
| __kmp_fork_barrier | libiomp5.so | 2.179s | 2.179s | 0s | 0s | 0s | 0s | 2.17899 | 0s |
| __kmp_barrier | libiomp5.so | 1.421s | 1.421s | 0s | 0s | 0s | 0s | 1.42099 | 0s |
| __kmpc_reduce_nowait | libiomp5.so | 0.169s | 0.169s | 0s | 0s | 0s | 0s | 0.169011 | 0s |
| __kmp_launch_thread | libiomp5.so | 0.071s | 0.071s | 0s | 0s | 0s | 0s | 0.0710027 | 0s |
| __kmp_join_call | libiomp5.so | 0.060s | 0.060s | 0s | 0s | 0s | 0s | 0.0599999 | 0s |

Figure 2: Scaling performance of the baseline OpenMP implementation.




# 3 Parallelizing with MPI

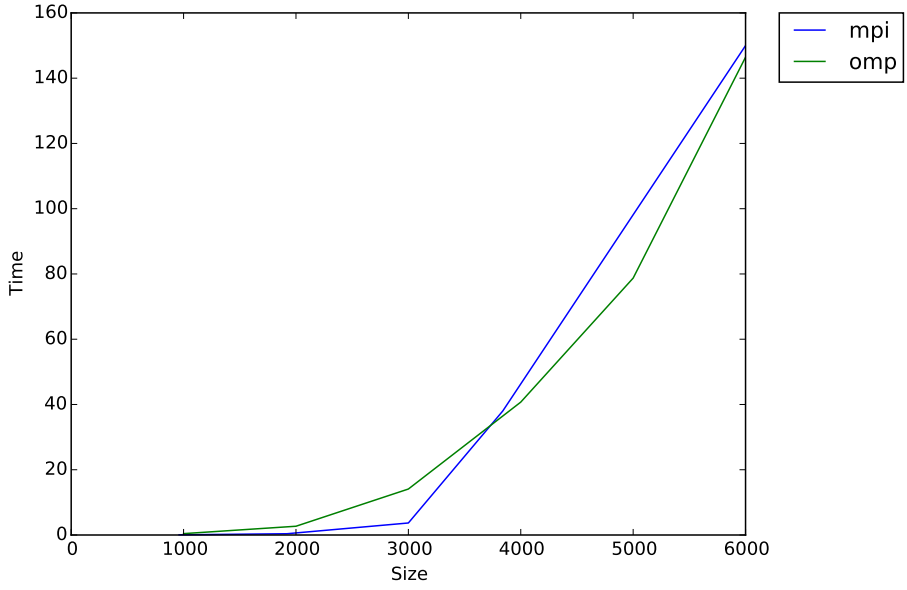
We have used two different ways of parallelizing the code using MPI. In the first approach, we split the matrix into different blocks,lblock where each block is handled by a different processor. We first initialize each block and infinitize it. Then we use MPIAllgather function to generate the updated full matrix,l and then broadcast it to very processor. The square operation is then performed where each block gets updated using the data from the full matrix,l. Then, we again use MPIAllgather to generate,l. We change the loop ordering to k,j,i in square operation. These square operations are then done iteratively until the value of done for each block is 1. To check this condition, we have used MPIAllreduce. For this approach, the code is written in a way that matrix sizes

should be chosen in a way that it evenly divides the number of processors. Also, the operation of deinfinitizing the loop and computation of the total time taken is only performed at processor 0.

In the second approach, we divide the array l into local arrays and we assign each local array to the processors using MPI_Scatterv. We also change the order of the loops in the square function: k,j,i. At each iteration of the outer loop (k), we broadcast the kth column of the matrix, which completes all the data needed to make the computation for that iteration. At the end we use MPI_Gatherv to gather the final results.

The initial performance results for our MPI implementation are shown in Figure 3. We plot timing results for only the first approach here as their runtimes are very similar. For small graph sizes the performance of this implementation is superior, but it becomes slightly worse than the OpenMP implementation for larger graphs. This may be due to the overhead in communication in our naive parallelization strategy.

Figure 3: Scaling performance of our MPI implementation.




# 4 Future Work

There are many future directions to consider for this project. We are particularly interested in further optimizing our parallel MPI implementation. The current implementation maintains a copy of the matrix on every processor, which is unnecessary and increases communication costs. Going forward, we plan to use

a more sophisticated scheme to pass data between processes, in which processes selectively communicate small chunks of the matrix between each other, which should decrease the asymptotic complexity of our implementation in terms of dependence on the number of blocks we partition the matrix into.

Of course, we will also spend extensive time tuning our final implementation. For instance, we will look into improving the vectorization of the innermost shortest path computation loop. The current implementation relies on icc automatix AVX vectorization, but there is likely room for improvement. We will also look into reducing branching.