CS 5220 11/14/2011

Project #3: All Pairs Shortest Paths

Professor Bindel *tbe4, dsj36*

Introduction

Given a graph specified by an adjacency matrix, the Floyd-Warshall algorithm computes a matrix A where A_{ij} is the length of the shortest path from node i to node j. This algorithm computes the shortest path between each pair of nodes at once, but it does not tell us anything else about the paths. Using the dynamic programming idea, we have the following recurrence, where A^s_{ij} is the length of the shortest path of at most 2^s steps.

$$A^{s+1}_{ij} = \min_k \{A^s_{ik} + A^s_{kj}\}$$

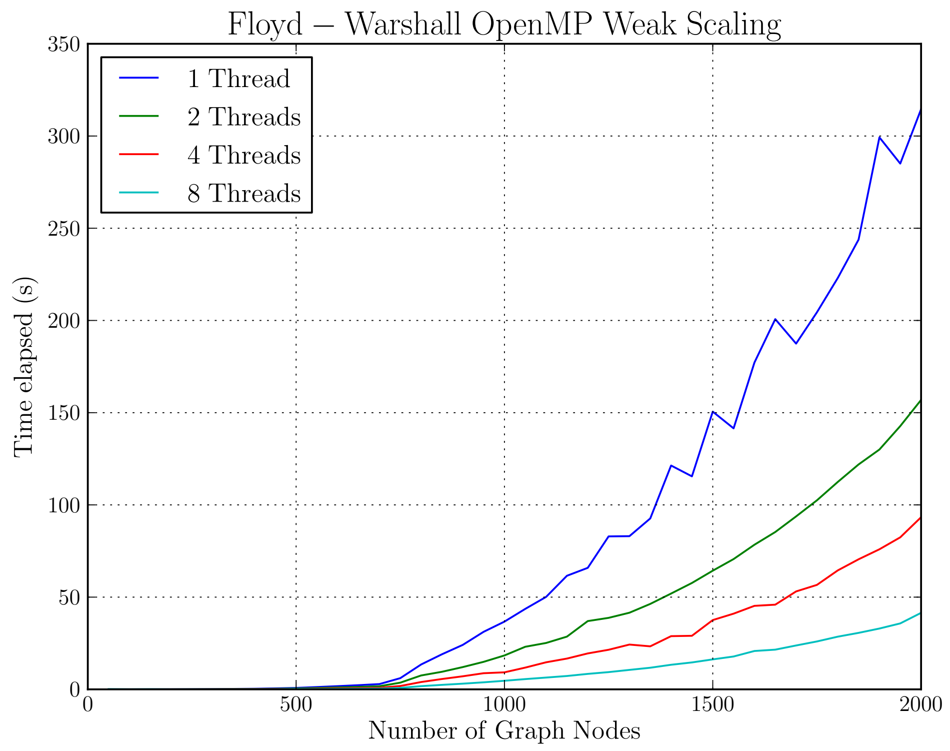
Therefore, we may start with $s = 0$, which corresponds to the adjacency matrix of the graph with the distance of unconnected vertices set to infinity, and iterate until we reach a fixed point. Since we are working with unweighted edges, we do not have to worry about negative cycles. Once we are done iterating, we set the edges of distance infinity to zero to return to the adjacency matrix representation.

Reference Implementation

The reference implementation of this algorithm is a straightforward parallelization in OpenMP. The loop over the columns, indexed by `j`, is split among the threads with an `omp parallel` pragma. After compiling with the `-O3` optimization flag, the scaling of this implementation was quite passable. We ran the program on randomly generated graphs of sizes 50 to 2000 nodes in steps of 50. Here is the data from the weak scaling study in both linear and log-log plots.

From the logarithmic plot, we observe that as the number of nodes becomes large, the slope of the plot roughly settles to around three, as we would expect from the $O(n^3 \log n)$ asymptotic behavior of our algorithm. Curiously, the number of iterations needed per experiment changed little. Past 200 nodes, each experiment used exactly three iterations. The randomly generated graph of 50 nodes required five iterations, the most of any graph.

Similarly, we fixed the problem size at 2000 nodes and varied the number of OpenMP threads from one to eight to collect data for the strong scaling experiment. The ratio between the time taken with one thread is plotted below.



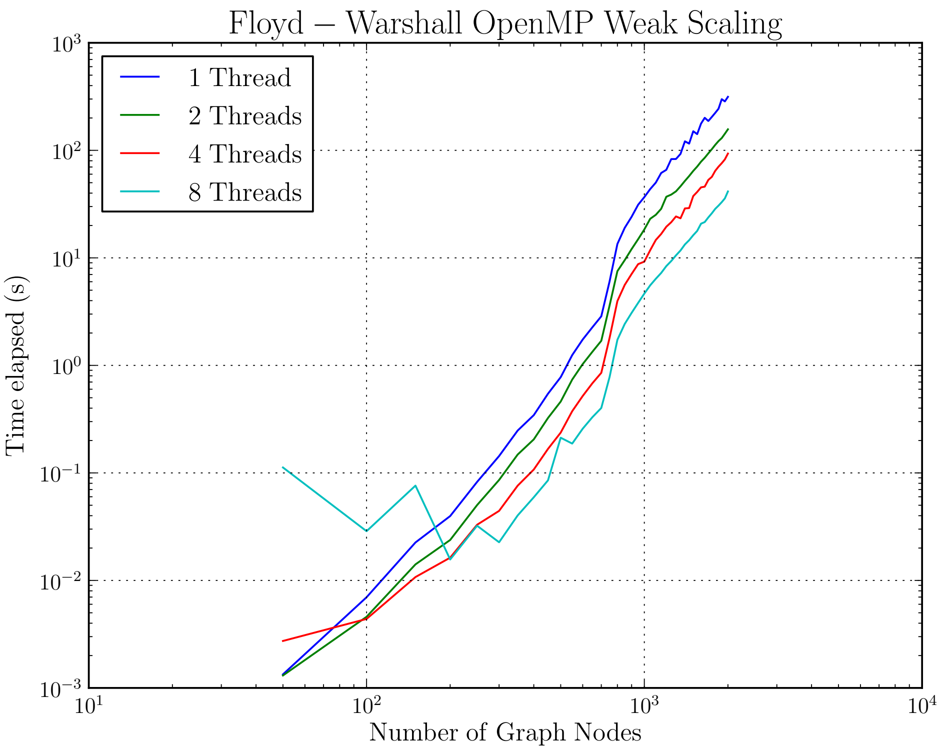


Figure 1: Weak scaling results from the reference OpenMP implementation

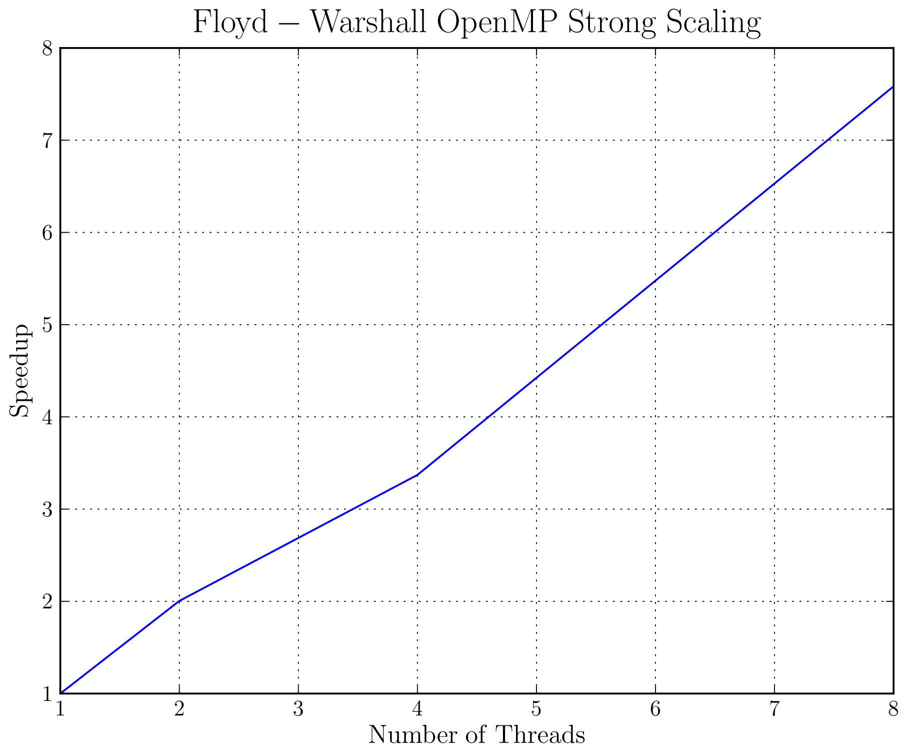


Figure 2: Strong scaling results from the reference OpenMP implementation

The OpenMP codes scale just below the p times speedup, which is good. Furthermore, we were not able to improve on the asymptotic behavior of the code, as the algorithm itself is $O(n^3 \log n)$. However, we were able to significantly improve the constant factor with some optimizations.

We profiled the OpenMP implementation with HPC Toolkit, but we found that after providing `gcc` the `-O3` optimization flag, the profiler did not give us much useful data. Specifically, almost all our time was reported spent in the comparison between the l_{ik} and l_{kj} and l_{ij} distances, which makes sense as it is the main computation in our inner loop. We were not able to diagnose any bottlenecks with the profiler.

Naive MPI Implementation

Since it does not rely on shared memory access, MPI is natural choice for improving the scaling of the code units. Our first implementation simply maintains a full copy of the distance matrix at each thread and synchronizes after each iteration with an `MPI_AllGatherv` command. As expected, each thread must broadcast its columns of the matrix to every other thread, incurring a roughly $O(n^2)$ cost. Our main aim in the remainder of this project will be to decrease the constant factor of these communication costs.

We collected similar data on the performance of the naive implementation. Given the amount of communication needed, it is no surprise that the naive code did not scale well.

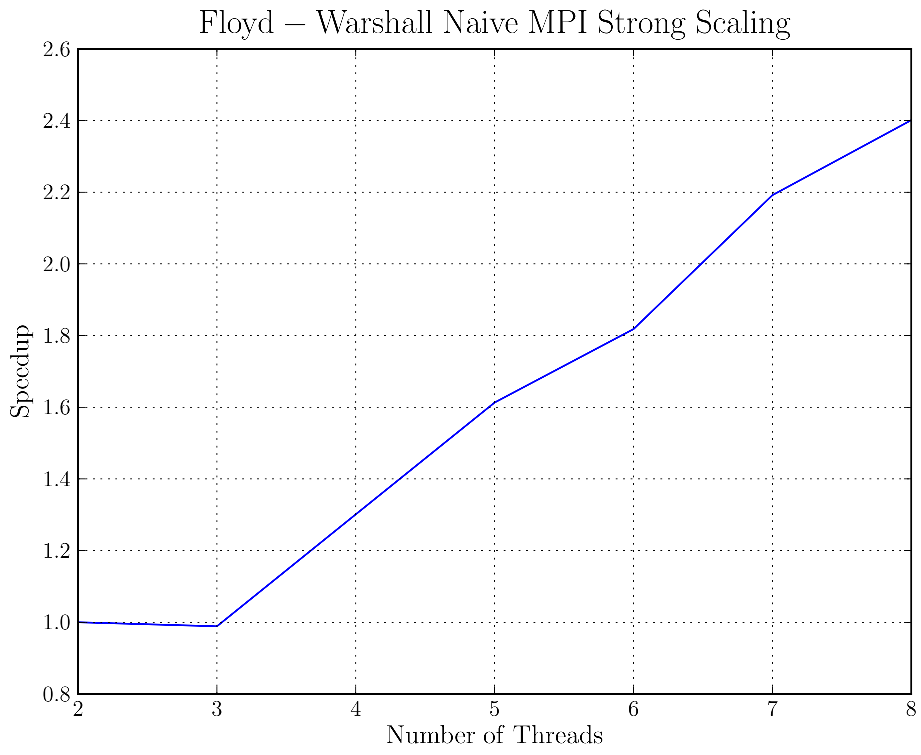


Figure 3: Strong scaling results from the naive MPI implementation. Note that running our code with eight threads is only a modest improvement over running it with two threads.

Ring MPI Implementation

The main idea behind this project is that we can rearrange the recurrence so that each thread only needs to see a few columns at once. Therefore, we may pass the columns of the original matrix around round-robin style, incurring much less communication overhead. Note, however, that our asymptotics do not improve, as $O(n^2/p)$ entries still need to be communicated.

Let $\{c_1, c_2, \ldots, c_p\}$ indicate a partition of the columns among the processors, and rewrite the recurrence as follows.

$$A_{ij}^{s+1} = \min \left\{ \min_{k=0}^{c_1} \{A_{ik}^s + A_{kj}^s\}, \ldots, \min_{k=c_p}^{n} \{A_{ik}^s + A_{kj}^s\} \right\}$$

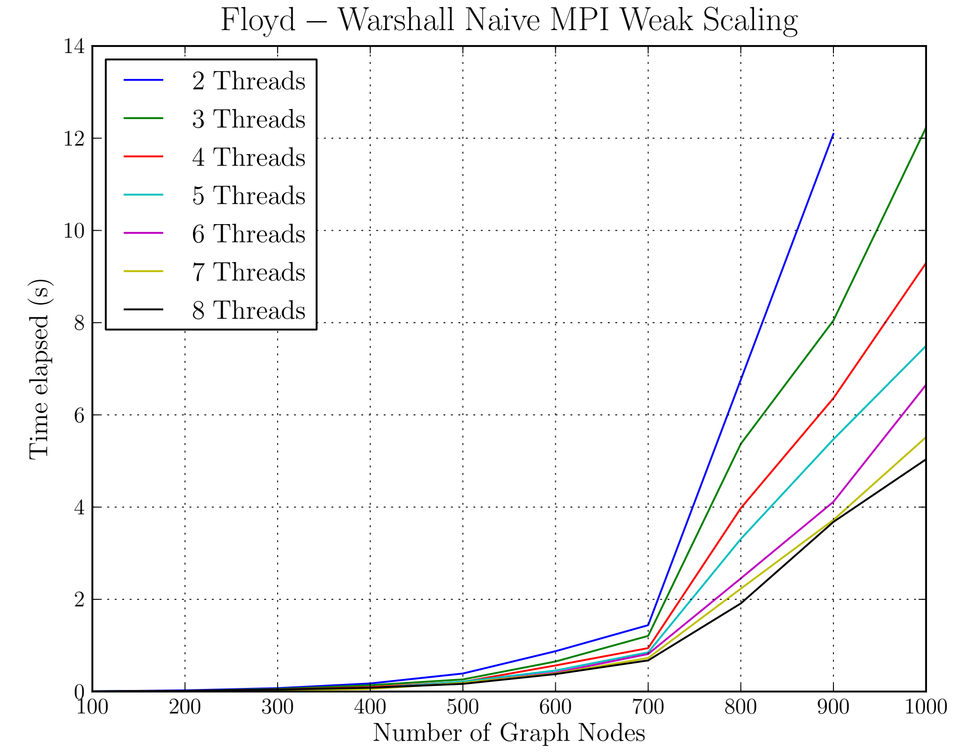
That is, we may compute the minima over only a few k, which correspond to columns of the matrix, and then keep a running minimum in our final matrix. The communication pattern starts with each thread doing the computations that involve only its local columns. Each thread then passes its chunk of columns to the right and receives from the left, wrapping around if needed. The computations involving these columns are then performed, and the process continues until each thread has seen every column. We maintain a flag on each thread indicating whether any of its columns have changed and then use `MPI_Allreduce` to determine whether all the threads have completed.

As we would hope, the performance of this implementation was an improvement. We ran the usual set of strong and weak scaling experiments on this implementation as well.
In addition, we manually instrumented our code to determine the amount of time spent in communication versus computation. We simply used `MPI_Wtime()` before and after the major computation and communication events and kept a running total of the time spent in each section.

The communication time is almost an order of magnitude smaller than the computation time once we have a sufficiently large graph. However, note that the slope of the communication curve is still two, indicating a quadratic amount of communication. This result stems from our previous observation that we did not actually improve the asymptotic behavior of the code.

In terms of explaining the communication costs in terms of, say, the $\alpha - \beta$ model, the linear factors for latency and bandwidth are hidden behind the quadratic behavior of the communication time. However, the results from our manual profiling do confirm our suspicions that the computation is indeed $O(n^3 \log n)$ and the communication is $O(n^2)$.

We conclude with some comparisons between the OpenMP and ring MPI implementations. For large graphs, the MPI implementation is about two times faster than the OpenMP implementation.



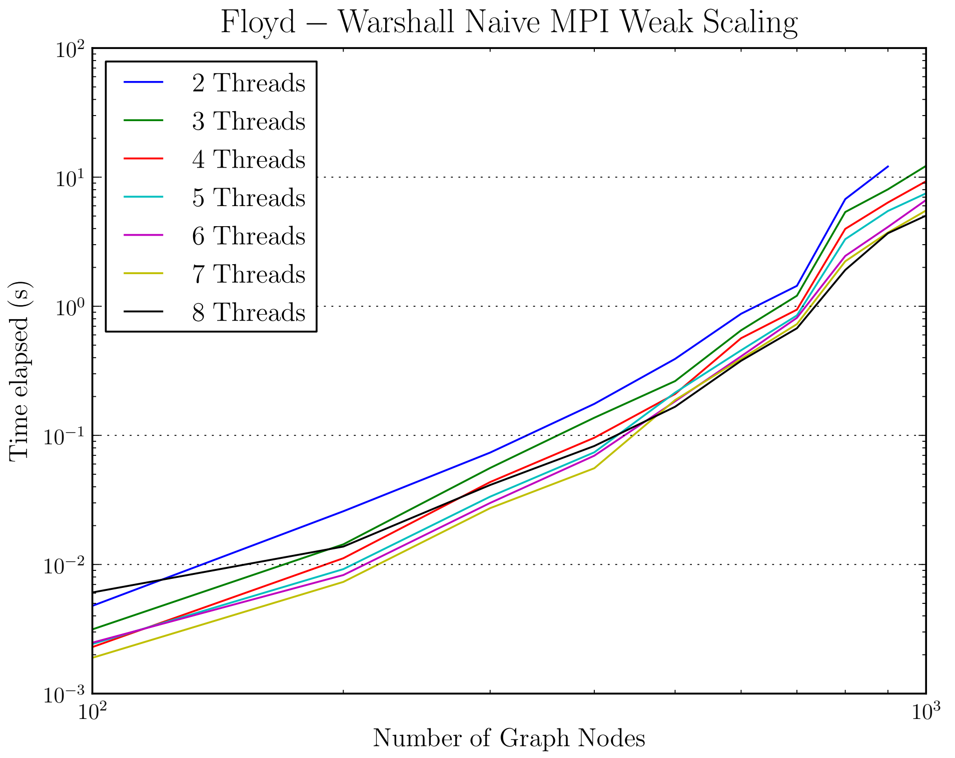
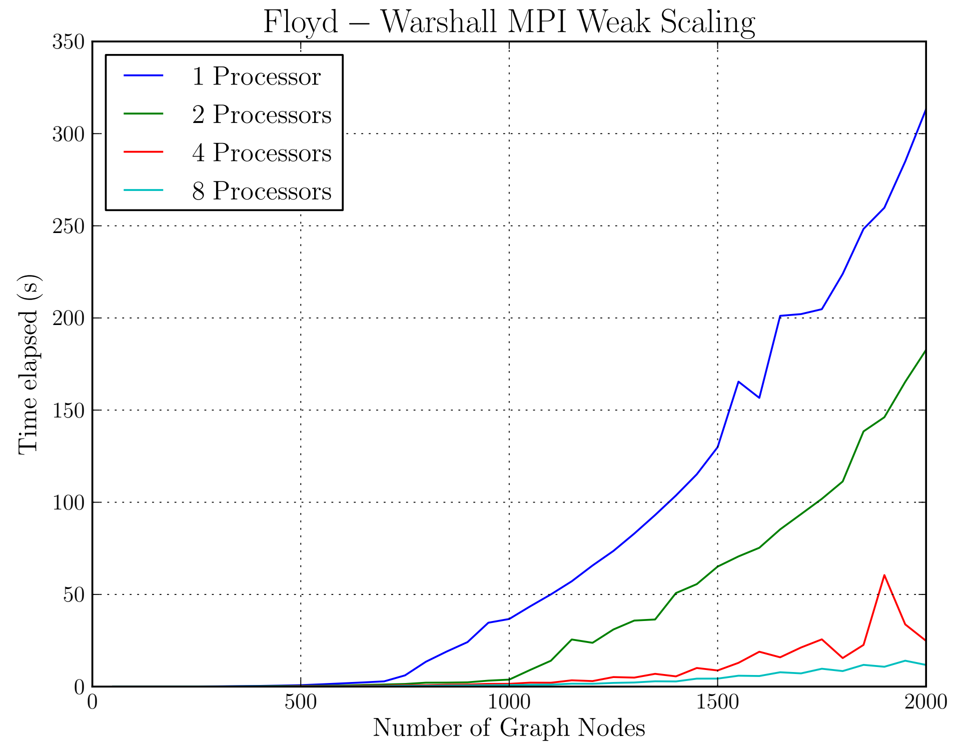


Figure 4: Weak scaling results from the naive MPI implementation. Note that in the log-log plot, the curves are only displaced vertically from each other by a small amount: This indicates that the constant factor of improvement is not much from increasing the number of threads.



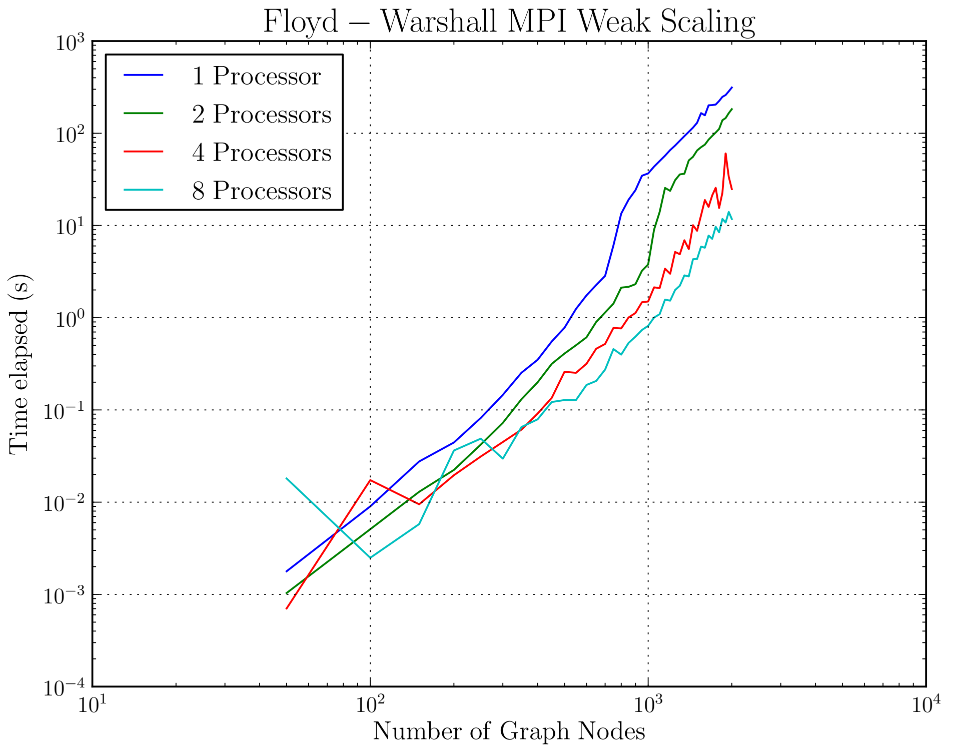


Figure 5: The usual weak scaling experiments for the optimized MPI implementation. The log-log plot shows a good improvement in the constant factor with increased processor count, but, as expected, the asymptotic behavior remains the same.

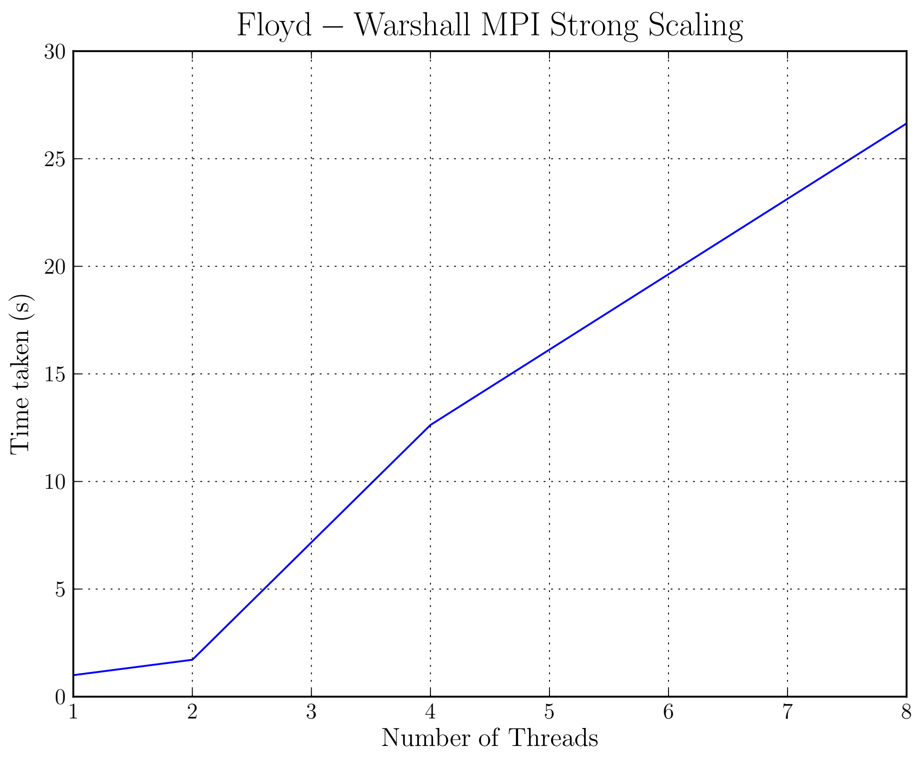
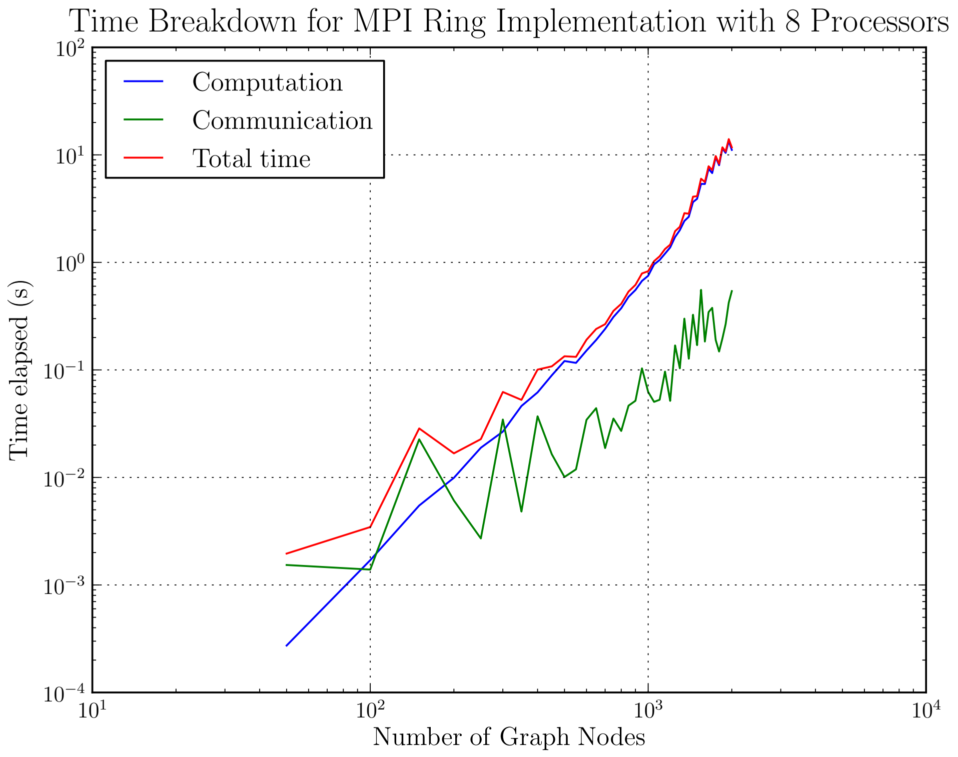


Figure 6: This strong scaling plot is a little misleading. There is a significant amount of overhead when we run the code with a few number of processors, so the 25 times speedup with eight processors isn't quite genuine.



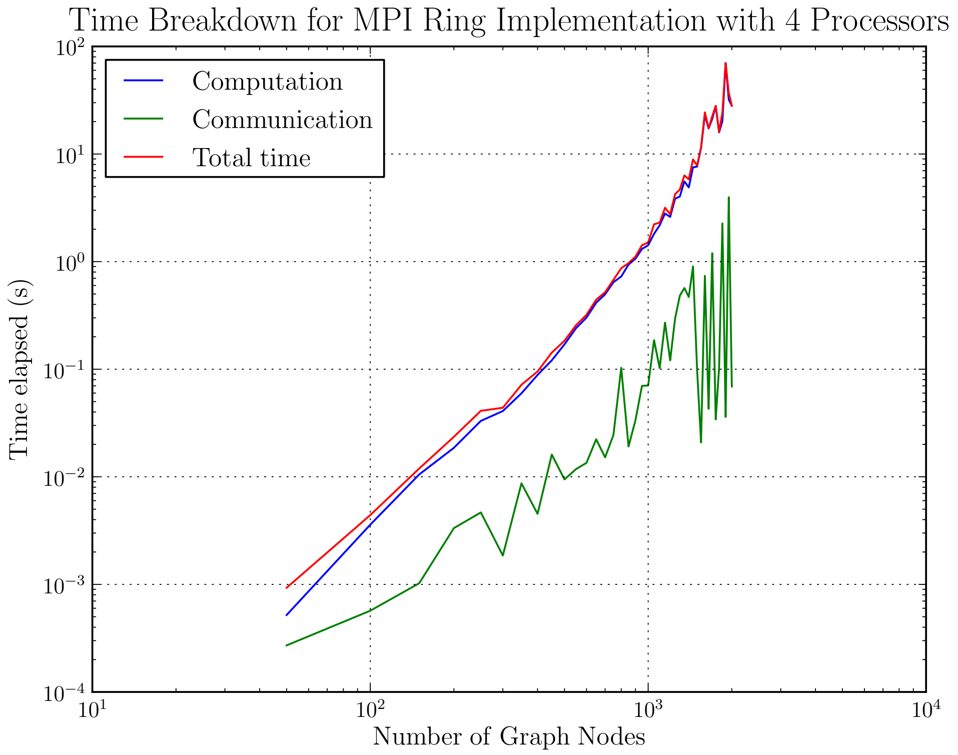
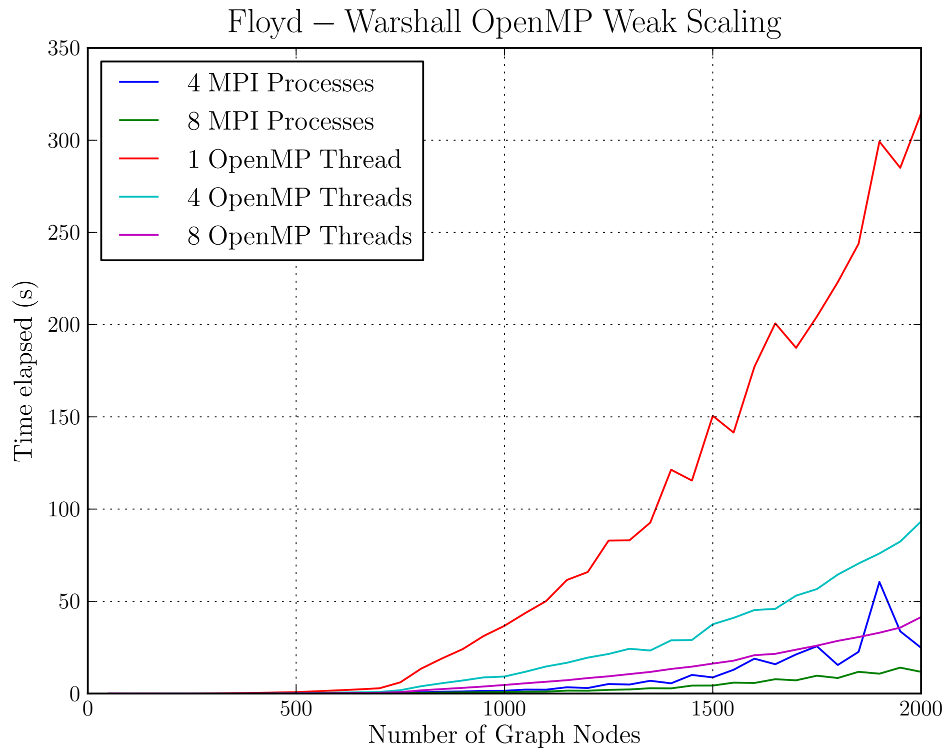


Figure 7: Breakdown of time spent in computation and communication on a log-log plot for the ring MPI implementation.



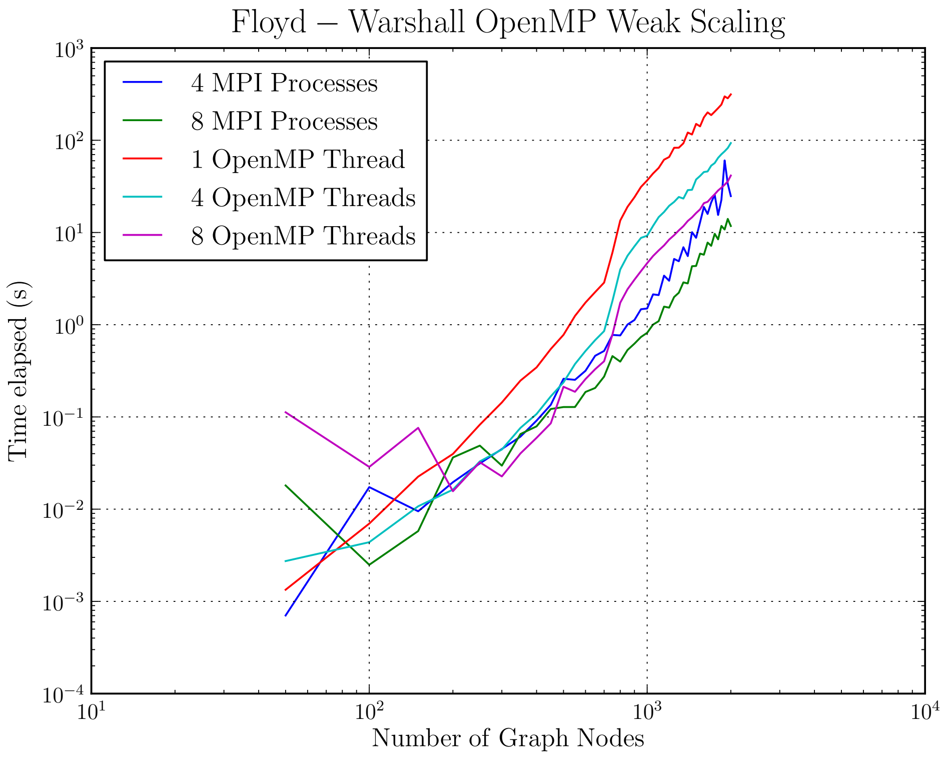


Figure 8: A comparison of our timing experiments for the optimized MPI and OpenMP implementations. Note that the fastest MPI code runs roughly twice as fast as the OpenMP code for large graphs.