CS 5220 Homework 3 Final Report

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

November 19, 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. We finally experimented with tuning both our MPI implementation and our OpenMP implementation.

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:

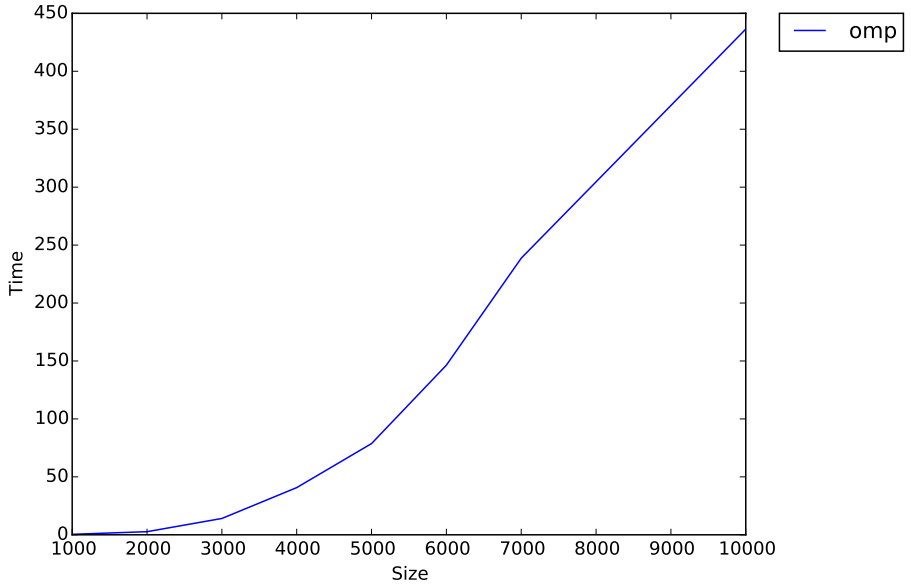
```
-O3 -no-prec-div -xcore-avx2 -ipo -qopt-report=5
-qopt-report-phase=vec-qopt-report=5 -qopt-report-phase=vec
```

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

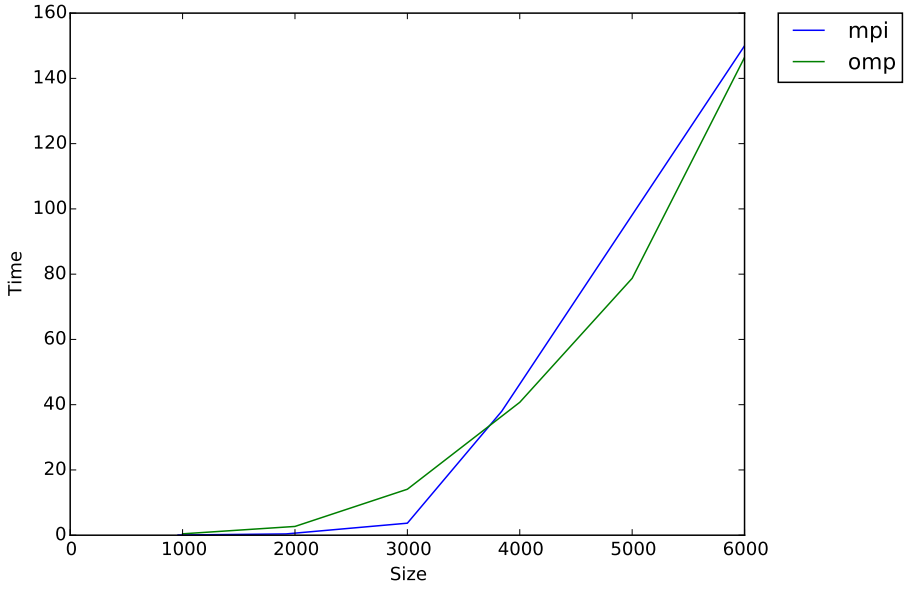
We have used two different ways of parallelizing the code using MPI. In the first approach, we split the matrix into different blocks,lblock where each block is handled by a different processor. We first initialize each block and infinitize it. Then we use MPIAllgather function to generate the updated 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. Like the first approach, the chosen matrix sizes evenly divides the number of processors.

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.

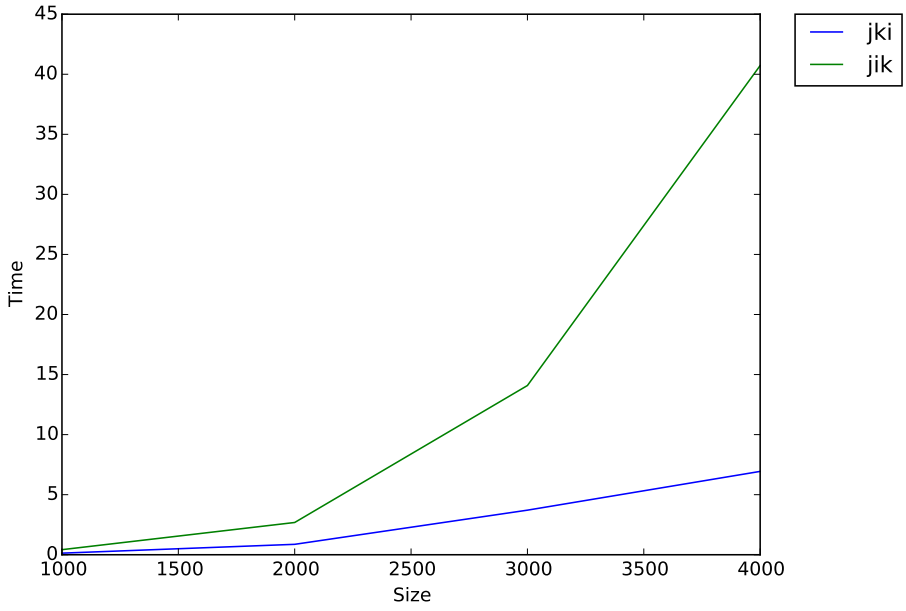


4 Tuning OpenMP Implementation

4.1 Loop Reordering

The first technique we used to tune our OpenMP Floyd-Warshall implementation was loop reordering. For the matmul project, we found that the "jki" loop ordering had superior performance because it effectively has unit stride when we work with column-major matrices. Since our Floyd-Warshall has the same access pattern and also uses column-major matrices, it seemed plausible that the "jki" ordering would have superior performance in this setting as well. Indeed, this is the case, as is shown in Figure 4. The improvement due to use of the "jki" ordering becomes increasingly prominent as the matrix becomes larger and time spent on cache misses becomes more pronounced.

Figure 4: OpenMP performance with loop reordering.

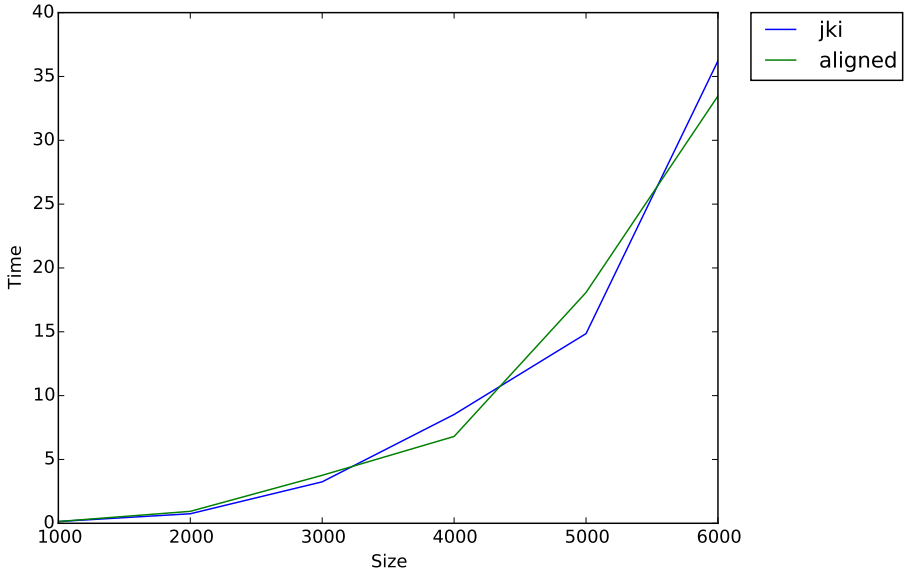


4.2 Improving Vectorization

We attempted to guide icc in improving the vectorization of the inner loop of the Floyd-Warshall computation. In particular, we aligned the arrays in memory and used the `#pragma vector aligned` and `#pragma ivdep` directives to guide vectorization. The vectorization report did not indicate any issue with the if statement in our inner loop, so we didn't bother trying to improve this aspect. Our efforts resulted in a slightly nicer looking vectorization report, but had little

effect on the overall computation time. This result is shown in Figure 5.

Figure 5: OpenMP performance with memory alignment.



4.3 Preprocessor Directives

We experimented with a few new preprocessor directives in addition to the baseline. In particular, we were interested in the `-fstrict-aliasing` keyword, as this improved our performance by about 1 GFlop/s on the matmul project. The full set of preprocessor directive we used was:

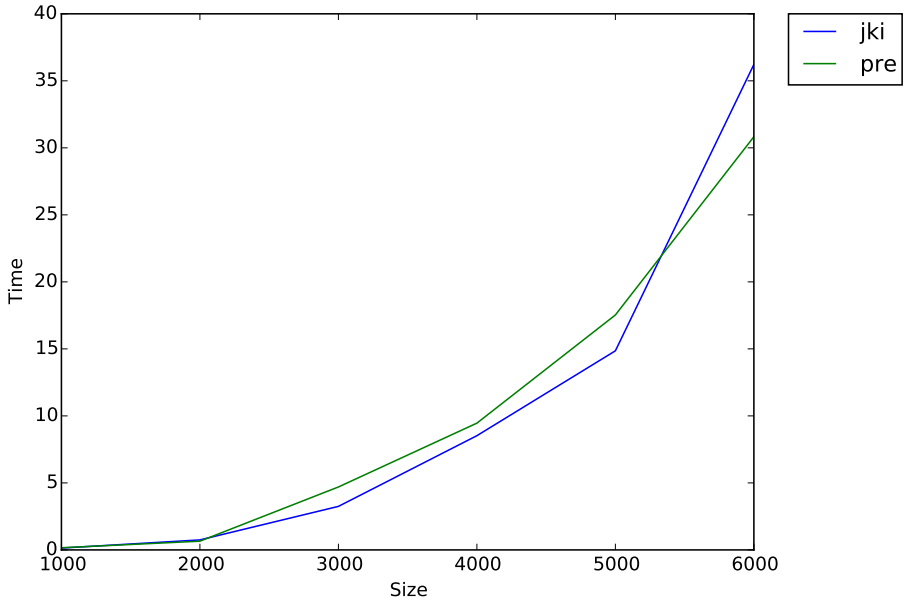
```
-O3 -no-prec-div -xcore-avx2 -ipo -qopt-report=5
-qopt-report-phase=vec -fstrict-aliasing -unroll-agressive
```

The results are plotted in Figure 6. The improvement with the extra flags was marginal, but this might because we did a good job indicating aliasing and aligment to the compiler without the addition of the extra flags.

4.4 Offloading

An optimization that ended up being very helpful for our OpenMP Floyd-Warshall implementation was offloading the parallel portion of the computation to the Xeon Phi coprocessor. Our strategy was to simply offload the entire shortest path computation - the loop in which we repeatedly call the `square` function is offloaded so that we don't waste time offloading after each `square` call. There is some overhead to offloading which dominates for small input sizes, but

Figure 6: OpenMP performance with additional preprocessor directives.

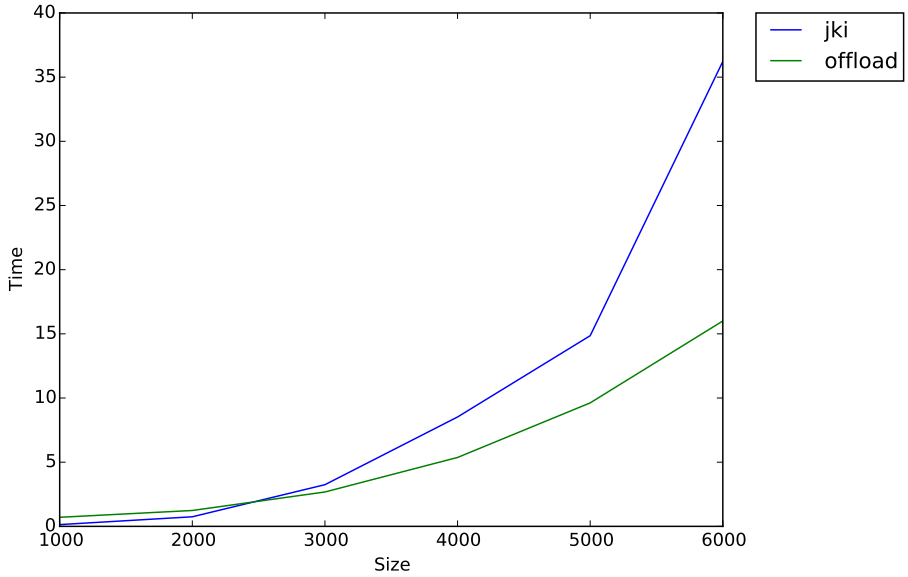


for larger sizes the advantage of having 236 threads available on the coprocessor vs. 24 threads on the main board becomes significant. This is shown in Figure 7, where we plot the performance of the offloaded implementation with 200 threads against the non-offloaded version with 24 threads.

5 Tuning MPI Implementation

We have attempted to tune the MPI version written using the second approach (as mentioned earlier, this approach uses matrix sizes which evenly divides the number of processors). Loop ordering jki will not be optimal for the MPI version as broadcasting operation is performed in the k loop. So, we have used kji ordering for the MPI version. We have also tried manual vectorization using _mm256 operators. We used _mm256_set_epi32 for broadcasting, _mm256_load_si256 for loading variables, _mm256_add_epi32 for addition of two vectors, _mm256_min_epi32 for taking minimum of each component of the vector, _mm256_testz_si256 for comparison and _mm256_store_si256 for storing the vector in a given variable. Here, we would like to acknowledge group 5 for helping us finding relevant mm256 operations. We compared the vectorized code timing with the original code for 1920by1920 matrix and we were not able to see any siginificant change in the timing (0.329 second for the vectorized code and 0.322 for our original code). We have tried to improve the performance of vectorized

Figure 7: OpenMP performance with basic offloading.



code by reducing the number of operations but we faced the problem of getting an incorrect answer. Also, our vectorized code will only run for matrix sizes which are divisible by both 8 and number of processors. We thought of generalizing it but our initial attempt did not provide any significant gain.

6 Scaling Studies

6.1 OpenMP Implementation

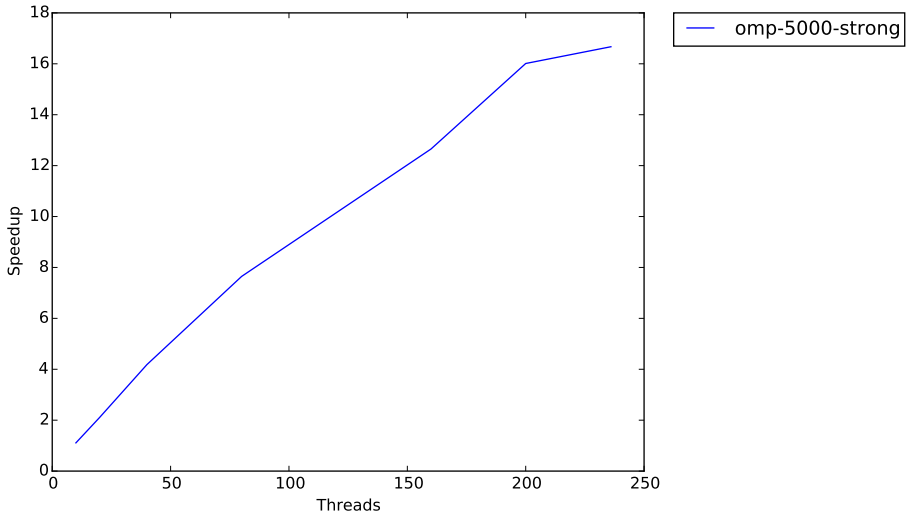
We performed scaling studies with the tuned OpenMP version of our code. This implementation features a full set of compiler flags, memory alignment, loop reordering, and offloads computation to the Xeon Phi coprocessor. We ran the strong scaling study with a 5000-node graph and compared to our best serial code, which features the improvements described above minus parallelization and offloading. The serial code took 135 seconds to complete the computation. The strong scaling study, which shows speedup relative to this serial benchmark, is shown in Figure 8.

We also performed weak scaling studies for the OpenMP implementation. The computation time of the Floyd-Warshall algorithm scales with the dimension n of the input graph as

$$T(n) = O(n^3 \log n).$$

This means that if $n(1)$ is the input size we consider with 1 processor, the input

Figure 8: Strong scaling performance of tuned OpenMP implementation.



size to compare against as a function of the number p of processors should be

$$n(p) \approx n(1)p^{1/3}.$$

We used $n(1) = 2000$ for our weak scaling study. The results are shown in Figure 9. The weak scaling performance is actually quite good. This may be because, unlike the water assignment, there is relatively low communication overhead to offloading computation to the Phi.

6.2 MPI Implementation

We perform weak and strong scaling studies for the MPI implementtation. The results are showin in Figure 11 and 10. The strong scaling performance is decent, which it may indicate that overhead is compensated by the division of the work into processors. However, the weak scaling performance is not good.

7 Future Work

An obvious direction to explore would be exploiting the graph structure of our problem for better parallelism. For instance one could partition the graph, compute the all-pair shortest paths in each partition, then synchronize between the partitions at the end. They key to making this approach work well would be to find a good separator for the graph, i.e., a set of edges to cut that is sparse, yet partitions the graph into components of comparable sizes. This would make it relatively easy to synchronize, but would also make it so that computing the

Figure 9: Weak scaling performance of tuned OpenMP implementation.

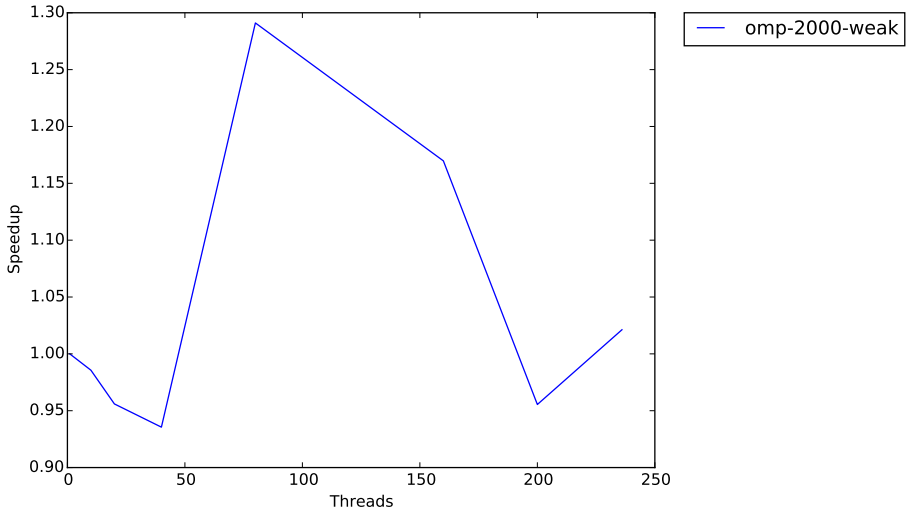
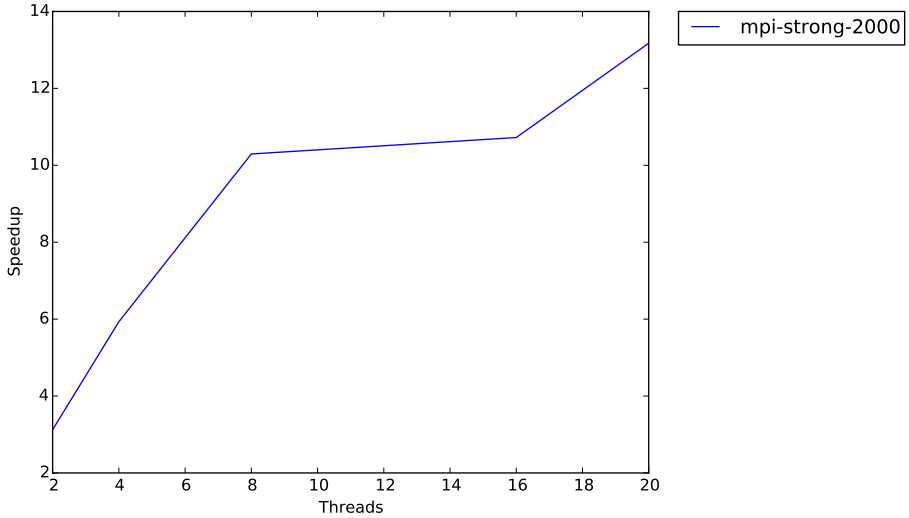


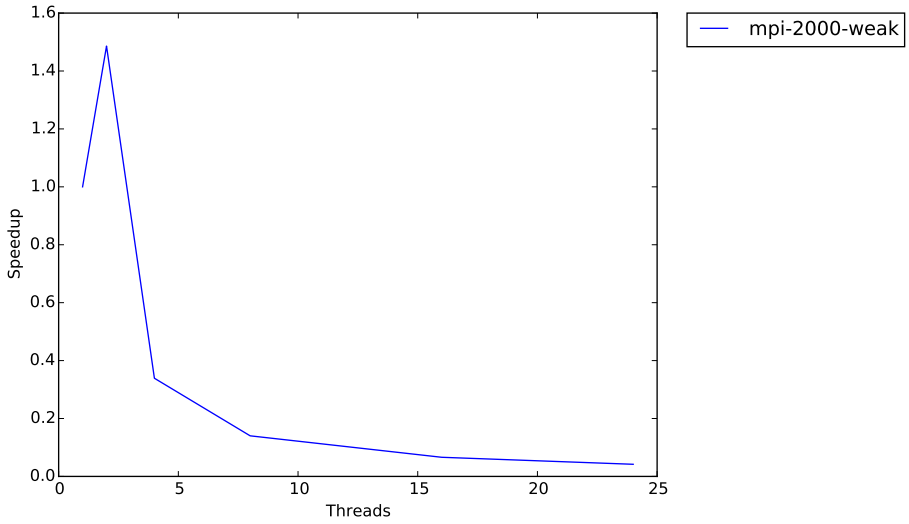
Figure 10: Strong scaling performance of tuned MPI implementation.



components separately is actually beneficial. With this sort of approach, we could exploit a type of parallelization where synchronizing between processes is very costly; this would be the case if we parallelized across multiple nodes.

Another promising approach would be to consider more sophisticated shortest

Figure 11: Weak scaling performance of tuned MPI implementation.



path algorithms such as Cannon's algorithm.