Project 3 - Final Report

Team 18 - Eric Lee and David Eckman

The Floyd-Warshall algorithm, which calculates all pairwise minimum-length paths in a graph, is implemented using tropical matrix multiplication, i.e., matrix multiply under the min-plus algebra. Under this arithmetic, the normal matrix multiplication formula

$$A_{ij} = \sum_k A_{ik} A_{kj}$$

is replaced with

$$A_{ij} = min_k \left\{ A_{ik} + A_{kj} \right\} .$$

If A is the adjacency matrix, then A_{ij} is one if there is an edge connecting node i and node j and zero otherwise. By squaring the matrix A using tropical matrix multiplication, the resulting matrix has elements A_{ij} which equal the length of the shortest path of at most 2 edges between nodes i and j. Notice that in the squaring update, the new A_{ij} is the shortest length of all possible paths from node i to some intermediary node k, and then to node j where each subpath consists of at most one edge.

Proceeding in this fashion, we can calculate the shortest paths of at most 2^m edges for any integer $m > 0$. To calculate all pairwise shortest paths, we need only compute the $(n-1)$th power of A, as any path of length n must have a cycle in it, and thus not be the shortest path. If the element A_{ij} in the $(n-1)$th power of A is zero, then there is no path from node i to node j. However, if we actually use zero literally, our tropical arithmetic will be incorrect because the zero will factor into the minimum operation; this necessitates the "infinitize" and "definitize" operations.

Benchmarking/Profiling

We profiled the OpenMP version of the code using the Intel VTune Amplifier, in particular the amplxe hotspots reports. We tested a range of graph sizes, varying the number of nodes from 500 to 8000 to observe how the OpenMP version scales (holding the number of threads fixed at 24). As expected, the CPU time of the square function increases on roughly the order of n^3 although the variability of runtimes makes it hard to clearly observe this relationship. There is a major increase in CPU time from 2000 nodes to 4000 nodes which may be explained by the "smart" cache size for the Xeon-E5 2650, listed as 20MB. Since each int is 4 bytes the maximum square matrix of ints we can store in cache is approximately 2289 by 2289. Therefore for $n > 2290,$ the matrix A can no longer fit in cache and thus fetches from main memory slow down the implementation.

CPU time (s)	Number of nodes (n)				
	500	1000	2000	4000	8000
square	0.357	4.760	39.880	726.863	4282.106
barriers	2.835	3.781	5.956	23.045	18.736
reduce	0.348	0.240	1.700	8.456	8.328

We also recorded the time used by the OpenMP barrier and reduce operations to see how they scale with the number of nodes. It appears that the overhead associated with the barrier functions and the reduce operation increases as the total amount of work increases, although at a slower rate than the square function. There is again a large increase in CPU times between graph sizes of 2000 and 4000.

Implementation Details

Our task was to implement this algorithm using MPI. Unlike the second project, where we optimized both with vectorization and parallelization, a change in arithmetic means we cannot easily use vectorization. We therefore focus on parallelization.

First, we note that by representing n in binary, we can cut down heavily on the amount of tropical matrix multiplies needed:

$$n = \sum bool_i 2^i$$

$$A^{12} = A^{\sum bool_i 2^i} = A^{0*1} A^{0*2} A^{1*4} A^{1*8}$$

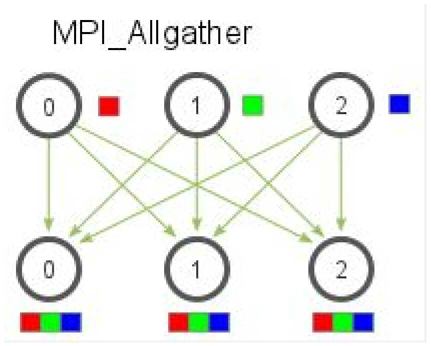
The key to making the MPI implementation perform well is choosing the best method of communication. Since we now assume there exists no shared memory, what is the best way to communicate information? Before we answer this question, we must figure out how to decompose the matrix A among processors.

We decided to have each processor hold a copy of A and calculate 1/p of the product columns. As long as A is not too large, decomposing A by columns will not be an issue. Because A is stored in column major form, each processor makes updates to a continuous section of memory. After each step, each processor broadcasts its piece of the product matrix to its neighbors.

We considered four ways of handling this communication:

-Send updated section of product matrix to neighbor and repeat p times
-Do a gather and then a scatter operation
-Do a hybrid (combination of the previous two).
-Use MPI_Allgather()

We tried all four methods on a toy problem of redistributing a large array around processors. Predictably, MPI_Allgather() was the fastest method and is illustrated below:



(picture taken from mpi-tutorials.com)

If you imagine Processors *P0*, *P1*, and *P2* holding the 0th column block, 1st column block, and 2nd column block respectively, we do an MPI_Allgather() at the end of a matrix-multiply to ensure that *P0*, *P1*, and *P2* all have the correct copy of the total matrix product at the end.

We also discussed decomposing the matrix into blocks rather than column chunks or using something fancier like Cannon's Algorithm to do the matrix multiplication, but did not implement these ideas.

Expected Speed-Ups

If we abbreviate "tropical arithmetic operation" as "Trop" (for fun), then a tropical matmul does $2n^3$ Trops. Since we are doing this $log(n)$ times, we are doing a total of $2log(n)n^3$ Trops. Then the total serial time is

$$Time_{Trop} * 2log(n)n^3 .$$

Given p processors, the total time will be

$$1/p \, * \, Time_{Trop} * 2log(n)n^3 \, + \, TotalTime_{\,communication} \, .$$

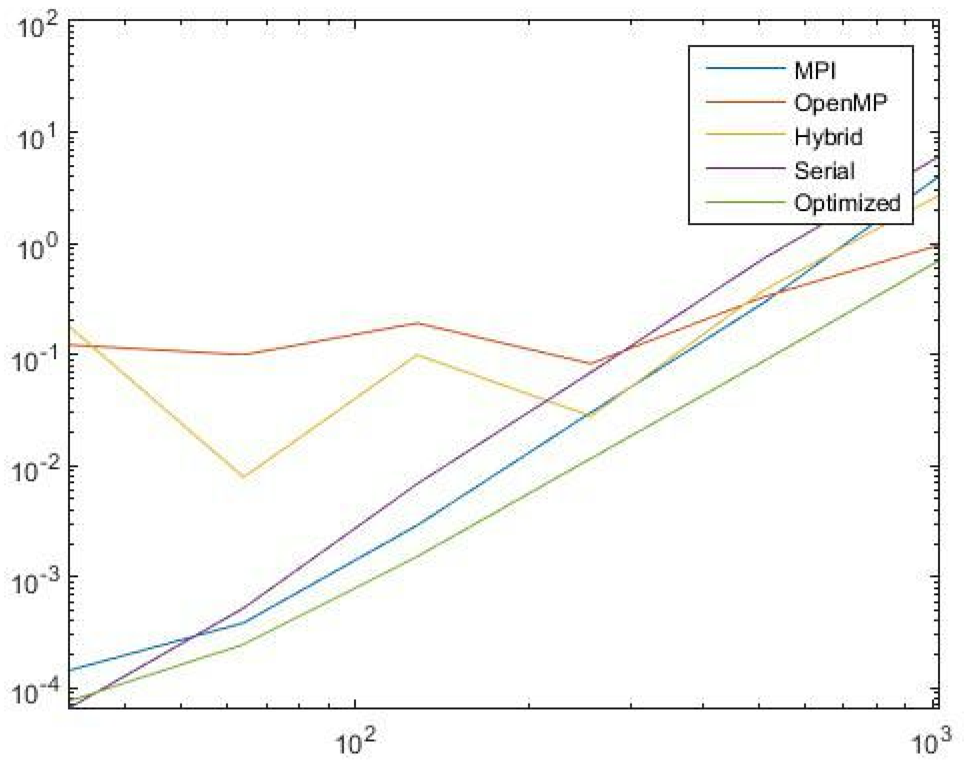
What is the total communication cost? Suppose it takes d seconds to send a single integer. Then our message cost is $Time_{size} = dn^2/p$ and our message overhead is $Time_{overhead} = C$ (some constant). Assuming receiving is free and synchronization is not necessary (which is not true in practice), then each processor is sending $log(n)$ messages. So our speed-up is roughly:

$$1/(1/p \, + \, d/(2 * Time_{trop} * n * p)) \, .$$

As long as our problem is large enough, we should ideally get substantial speedups.

Weak Scaling Results

In practice, achieving high performance using MPI on a single processor is challenging, as communication costs are massive relative to the number of Trops we are doing.



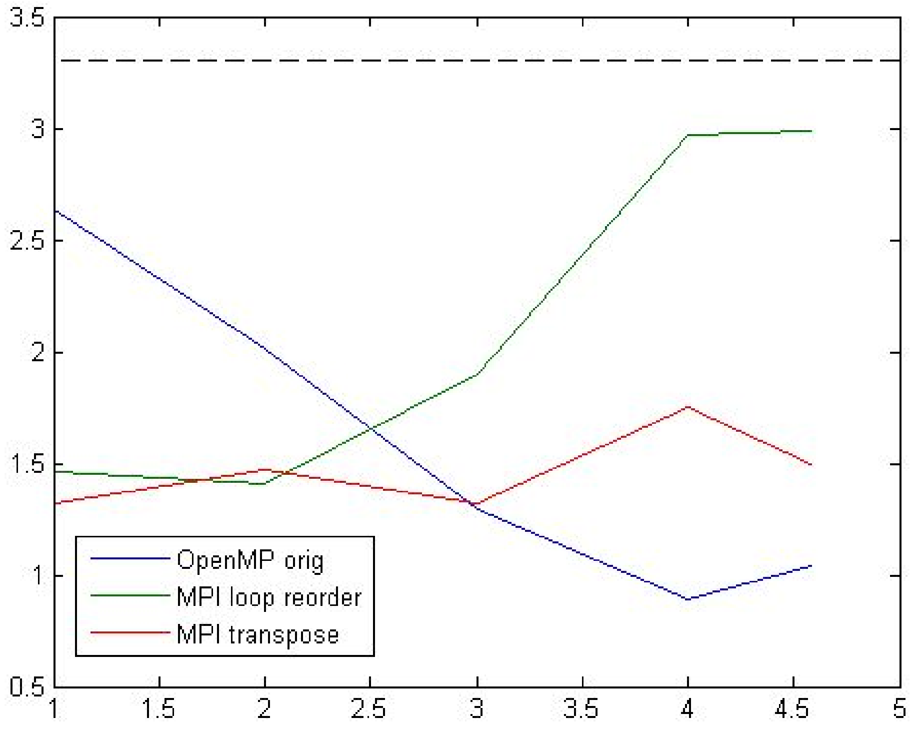
In this loglog plot of our code (log of problem size on the x-axis, log of wall clock time on the y-axis), we see that our MPI version scales rather poorly in problem size compared to the OpenMP version (on 24 processes and threads, respectively). This is simply because, as mentioned before, there’s really no point in message passing on shared memory; MPI is effective for communication *between* compute nodes, not *between threads* on the same processor. For smaller problem sizes, we found that MPI performs much better due to the larger overheads associated with OpenMP. Using a hybrid combination of both MPI and OpenMP gives us speed-ups a little-bit better.

Our previous work focused on optimizing the communication patterns between processes. Following the spirit of the previous assignment, we optimized the matrix-multiply itself by improving memory locality via copy optimizations and improving memory alignment with manual alignments in the code. More precisely, we copy row-blocks to our buffer to achieve unit stride. We also experimented with transposing the entire matrix in one go, but found that this was slower than buffering.

Our optimized version uses 6 MPI processes and 4 OpenMP threads per process. Decent speed-ups were achieved, especially as the problem size increased (not shown in the graphs).

Strong Scaling Results

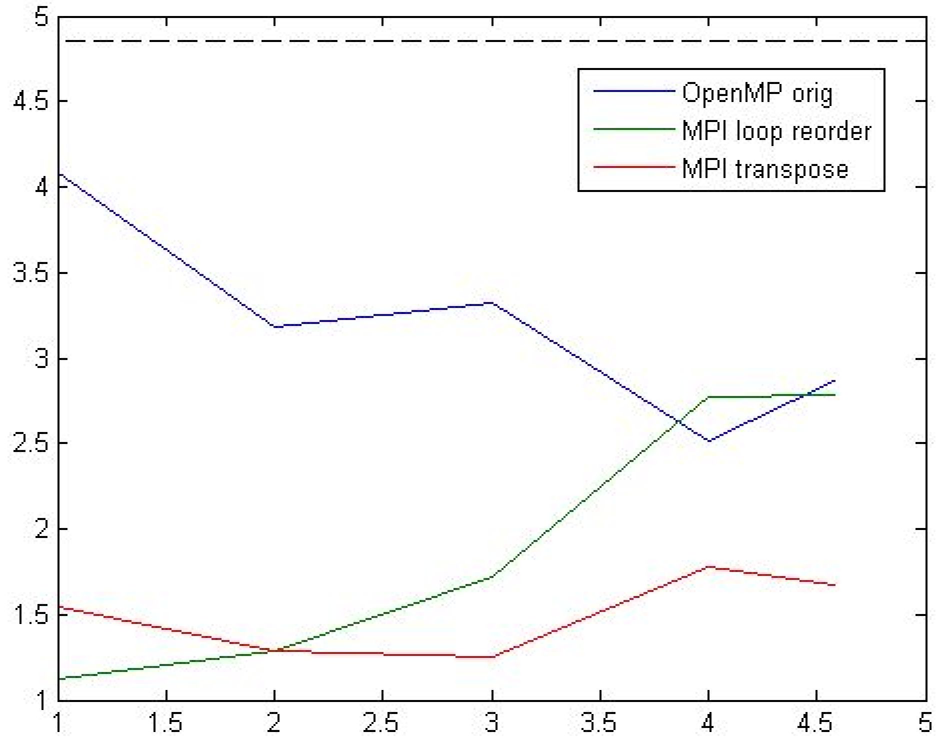
We also tested a couple of versions of our code against the serial implementation for a fixed problem sizes.



The above plot (log of number of processes/threads on x-axis, log of wall clock time (seconds) on y-axis) is for a problem size of n = 2000 nodes. The three implementations that were tested were the original OpenMP version, and early version of our MPI implementation with (with loops ordered to improve access patterns), and our optimized MPI implementation (with 4 OpenMP threads per process) wherein we use the transpose of A (in row-blocks) to improve access patterns in the matrix multiply. The dashed line at the top is the wall-clock time of the serial implementation.

The OpenMP version scales well and appears to achieve the best performance when using 16 threads. Among the MPI implementations, the performance of the "transpose" version is more consistent, unlike the loop reordered version which suffers poor performance for large numbers of processes.

The plot below shows the same comparison for a problem size of $n = 2048$ nodes. Note that the serial version runs a great deal slower for this problem size, which may be explained by issues with cache associativity given that this problem size is a power of two. The general trends seen for the case of $n = 2000$ appear here as well, except that the OpenMP implementation runs much slower whereas the performance of the two MPI versions is relatively unchanged.



Performance Analysis

We fit a line to each of the implementation results to see the estimated computational complexity of each. Note that we have an $O(log(n) * n^3)$ algorithm.

Type (with 24 threads/processes)	Complexity
Serial	$O(n^{3.35})$
OpenMP	$O(n^{0.54})$
MPI	$O(n^{3.02})$
Hybrid	$O(n^{1.01})$
Optimized	$O(n^{2.69})$

This line fitting reveals that OpenMP still scales much better than our other methods. However, we didn't think this was completely accurate, as the high overheads associated with thread-spawning probably skewed the line fitting. We compared the OpenMP version with our Optimized version for incredibly larger problem sizes, and found that the Optimized version outperforms the OpenMP version by a magnitude in order. Of course, this is a bit unfair as the OpenMP version isn't optimized, and doesn't handle large problem sizes well.

The question is: do these estimated computational complexities for our MPI implementations match our initial estimate? (given by $1/p \; * \; Time_{Trop} * 2log(n)n^3 \; + \; TotalTime_{\;communication}$ if you'll recall). Certainly, the serial version fits, as communication time is 0. If we interpret the $TotalTime_{\;communication}$ to be much larger than the other term in the summand, than our results fit as well. We simply aren't doing enough work per process.

Possible Improvements

Apart from improvements currently out of reach (requesting multiple nodes and offloading, both of which aren't currently supported on Totient), two other things we investigated were Cannon's Algorithm and SIMD comparison instructions. Cannon's algorithm is used mainly for matrices far too large to fit inside the memory of one processor. We wanted to build Cannon's algorithm on top of a multi-node matmul for very large matrices, but without the ability to use multiple nodes, Cannon's algorithm is sort of moot. We also didn't have the time to correctly implement SIMD comparison instructions (asking ICC to auto-vectorize didn't yield any noticeable speed-ups).