Initial Report

Team 18 - Eric Lee and David Eckman

The Floyd-Warshall algorithm, which is used to calculate all min-length paths, is implemented using Tropical Matrix Multiplication (that is, matrix multiply under the min-plus algebra). Consider the normal matrix multiplication formula

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

If A is the adjacency matrix, then A_{ij} is the length of the path from node i to node j. If we change from normal arithmetic, then A_{ij} is the shortest length of all possible paths from node i to some intermediary node k, and then to node j.

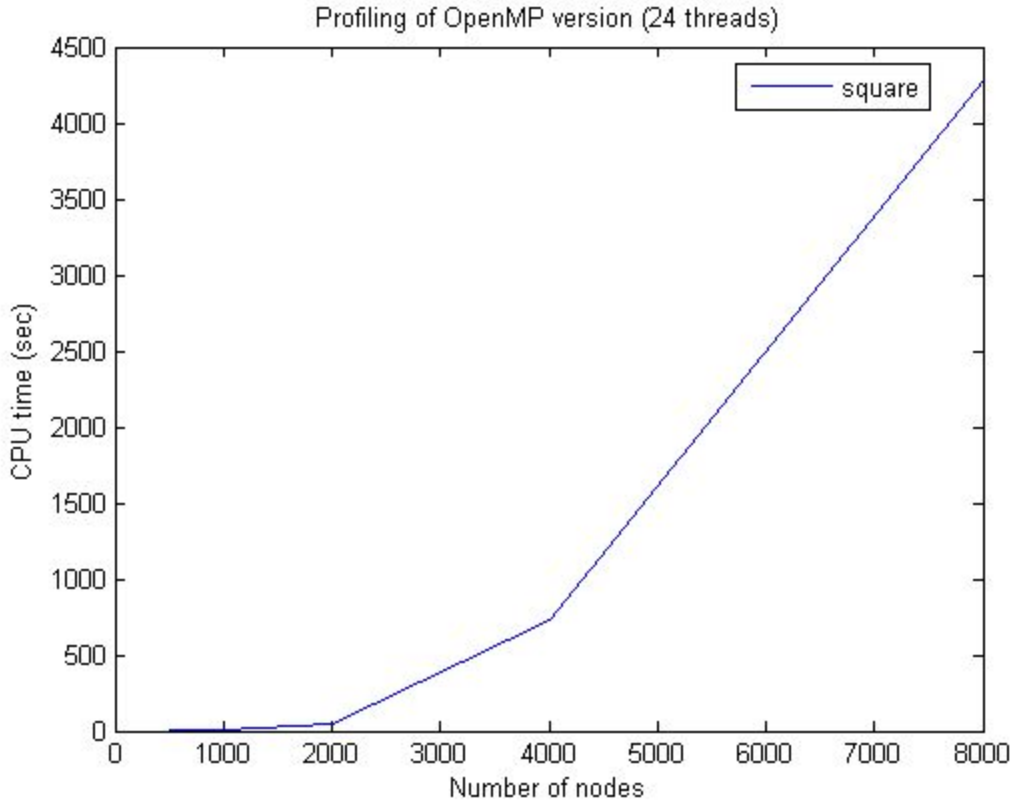
$$A_{ij} = min(\{A_{ik} + A_{kj}\}_k)$$

If A_{ij} is zero, then there is no path from node i to node j. However, if we actually use zero literally, our tropical arithmetic won't be correct; this necessitates the "infinitize" and "definitize" operations.

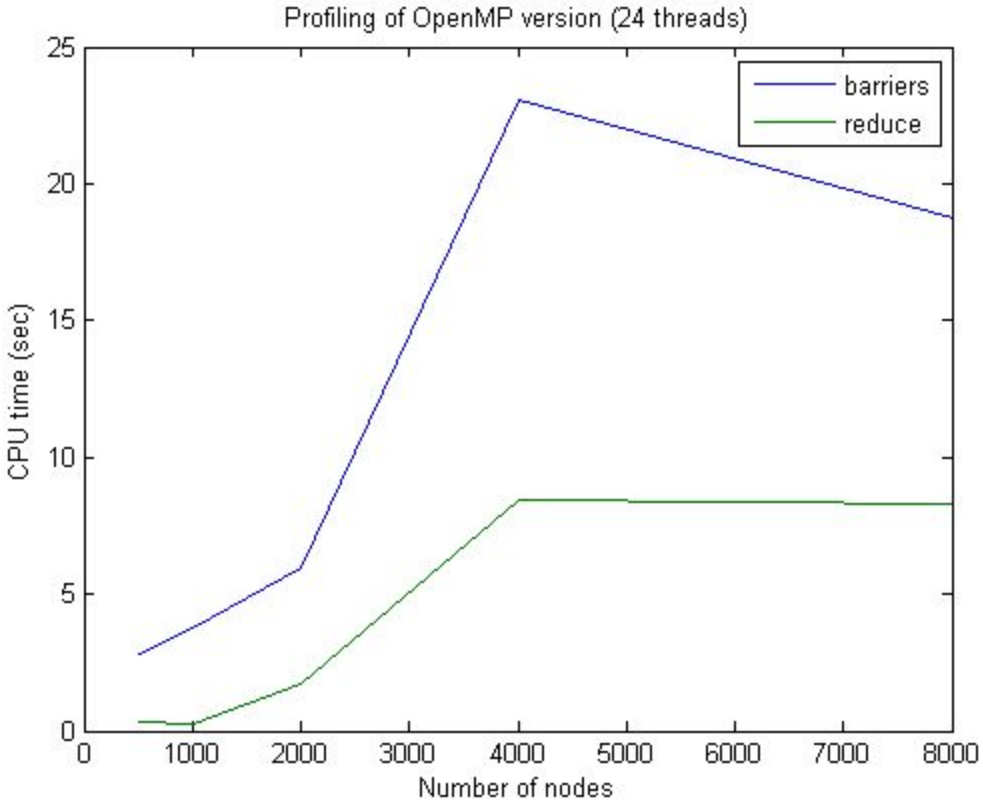
Then all we really need to compute is the (n-1)th power of A, as any path of length n must have a cycle in it, and thus not be the shortest path.

Benchmarking/Profiling

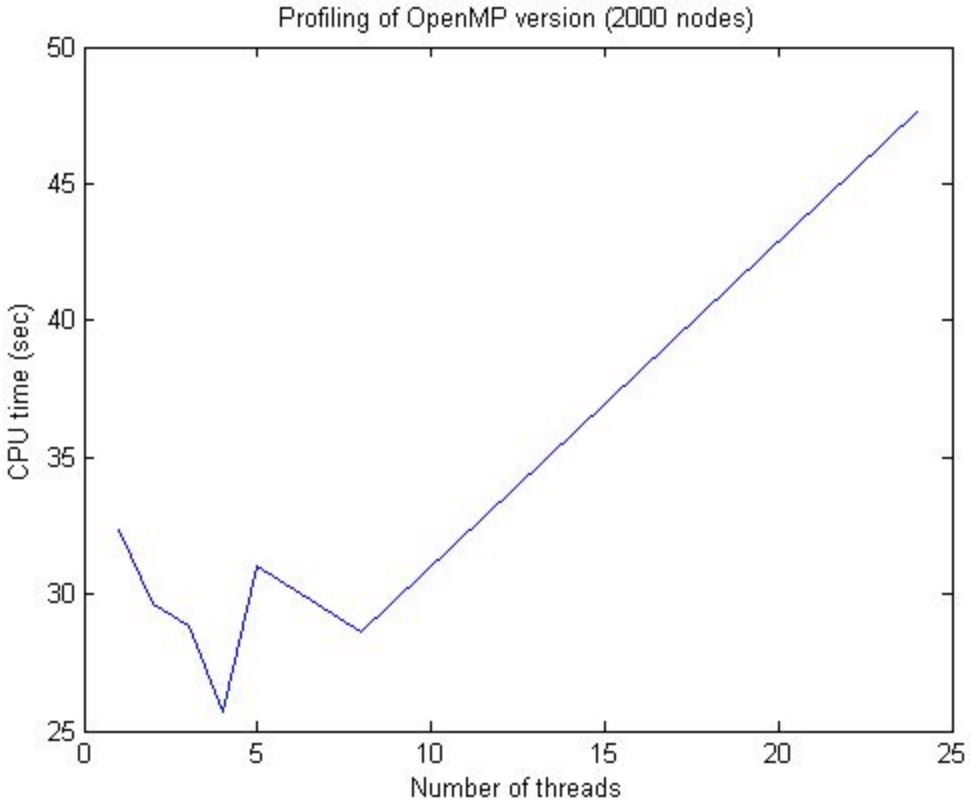
We profiled the MPI version of the code using the amplxe hotspots reports. We tested a range of graph sizes, varying the number of nodes from 500 to 8000 to observe how the MPI version scales in the number of nodes. As expected, the time increases on roughly the order of n^3 although the variability of runtimes makes it hard to clearly observe this relationship.



We also plotted how the time used by the OpenMP barrier and reduce operations scale with the number of nodes. Here the picture is less clear, though on the whole it appears that the overhead associated with the barrier functions and the reduce operation increase as the total amount of work increases, although at what would appear to be a slower rate than the time of the square function.



We held the graphs size fixed at 2000 nodes and varied the number of processors. The plot of CPU times suggests that the optimum number of threads to use for a 2000-node graphs is somewhere between four and eight, after which the overhead associated with more threads cuts into the savings in computational time.



Implementation Details

Our task is to do this via MPI. Unlike the previous assignment, where we optimized both with vectorization and parallelization, a change in arithmetic means we can't use vectorization, unless there exists a really hacky method. 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:

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

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

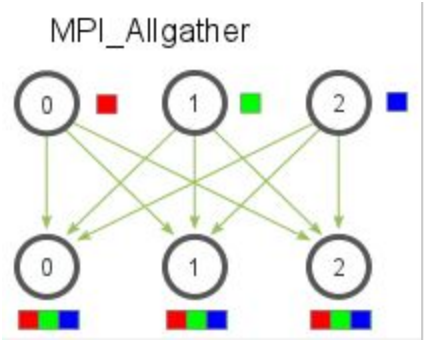
The key here is the best method of communication. Since we now assume there exists no shared memory, what's the best way to communicate information? Before we answer this question, we must figure out how to actually cut up our matrix.

While we would like to do something fancier like Cannon's Algorithm, we initially settled for having each processor hold a copy of A and calculate 1/p of the product columns. This isn't an issue as long as the A isn't too large. Then after each step, we need each processor to broadcast its piece to its neighbors.

There are four ways of doing this:

-Send 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 -redistributing a large array around processors. Predictably, MPI_Allgather() was the fastest method. It works like this:



(picture taken from mpi-tutorials.com)

So 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.

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 total. Then serially, the total time is

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

Given p processors, the total time will be

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

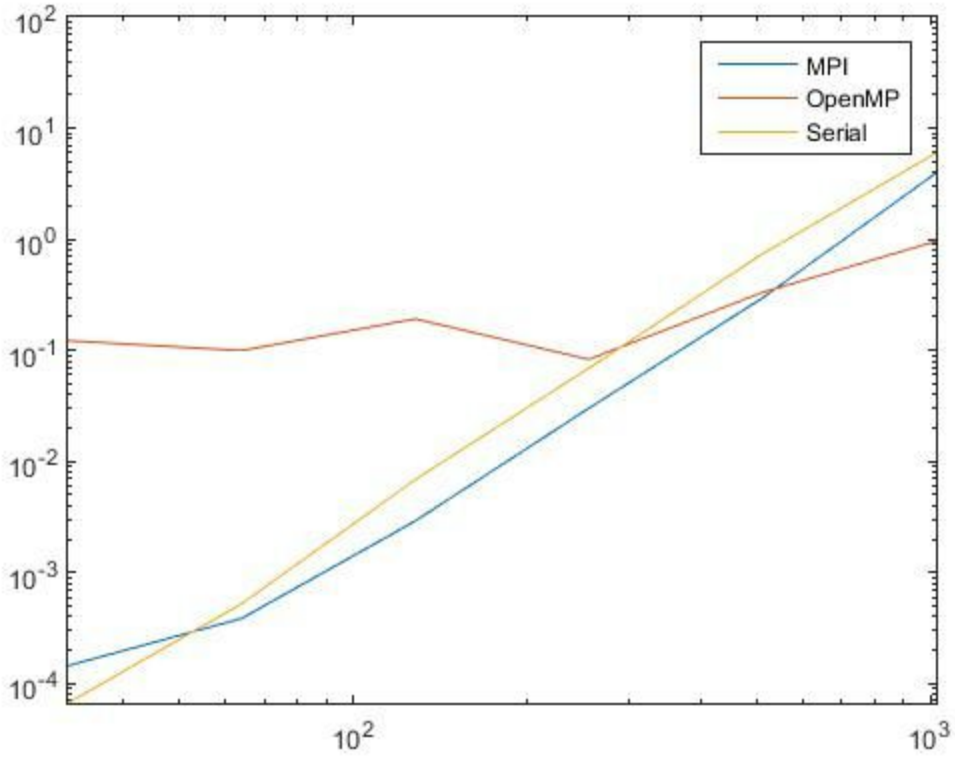
What is the total communication cost? Well, let's say that the it takes d seconds to send a single integer. Then our message cost is $Time_{size} = dn^2/p$ and our message overhead is $Time_{overhead} = C$ (some constant). Assuming receiving is free and synchronization isn't necessary (which isn't true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

$$p + d/(2 * Time_{trop} * n * p)$$

So as long as our problem is large enough, we ideally get substantial speedups

Results

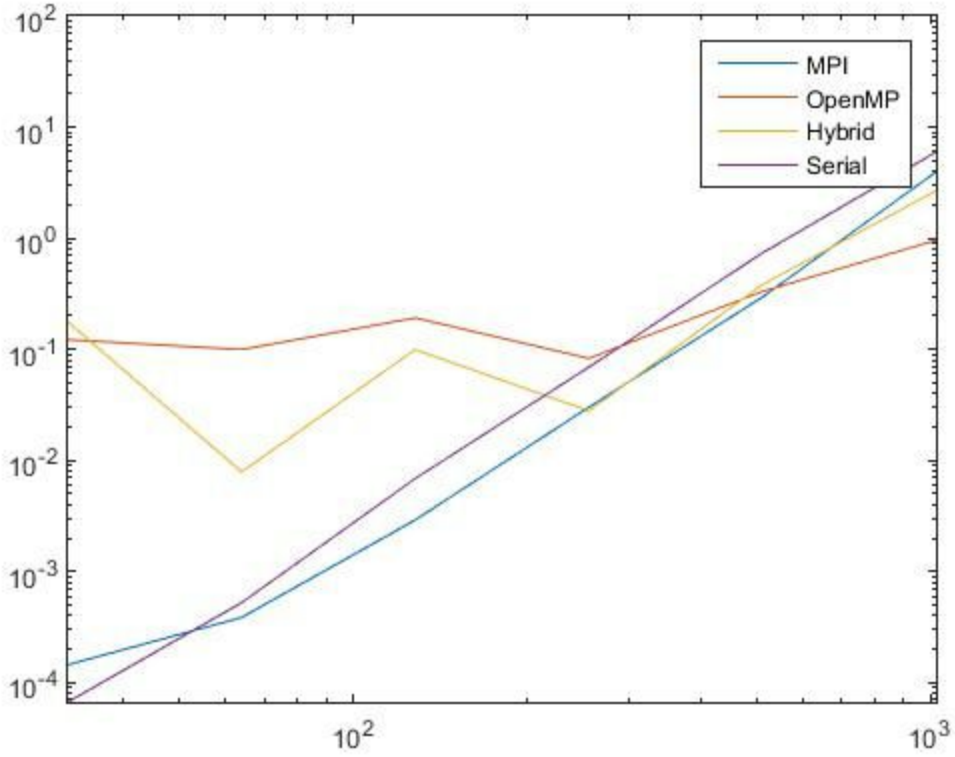
In practice, using MPI on a single processor sucks, as communication costs are comparatively massive relative to the number of Trops we are doing.



In this loglog plot of our code, 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, MPI performs much better due to the larger overheads associated with OpenMP, but this advantage disappears on larger problem sizes.

Using a hybrid combination of both MPI and OpenMP gives us speed-ups a little-bit better.



Future Work

We'd like to offload to the Phis, but we probably won't get any better speed-ups, as we are going to have the same problems as before. A much cooler thing to do would be having multiple nodes running OpenMP, with MPI communicating between them. This might crowd the cluster, but it's worth a try. For larger matrices, we'll also have to change our implementation to use a more memory-efficient algorithm… most likely Cannon's Algorithm.

We can also trying a different implementation of Floyd-Warshall where on iteration k, we keep track of the shortest paths which use only nodes 1 through k. Look into variant of Floyd-Warshall from Wikipedia here.