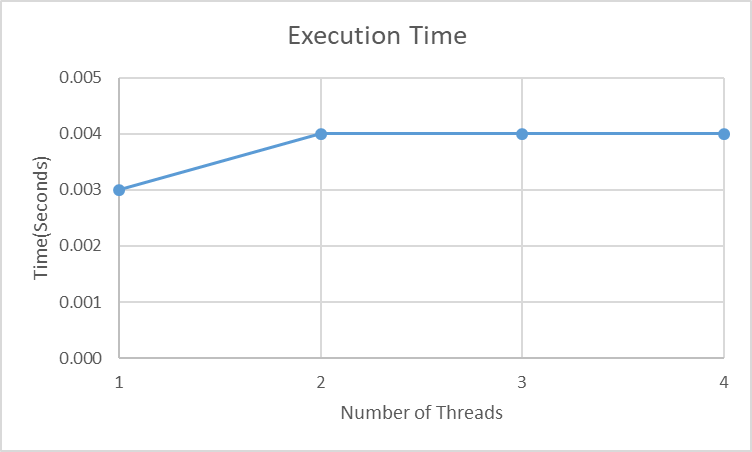
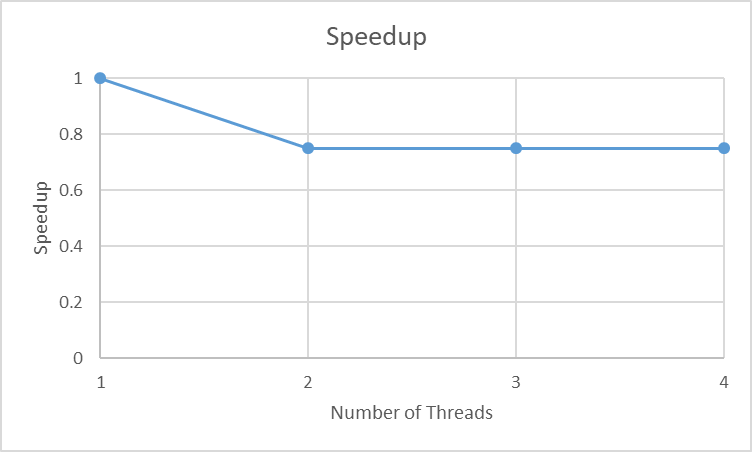
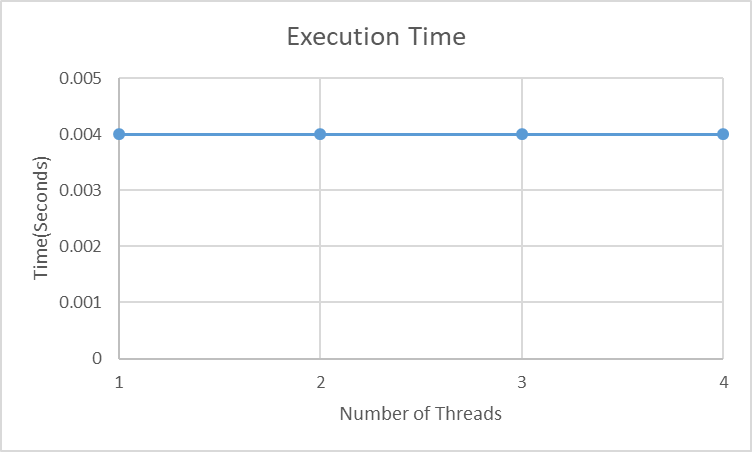
Lab 1: Parallel Travelling Salesman

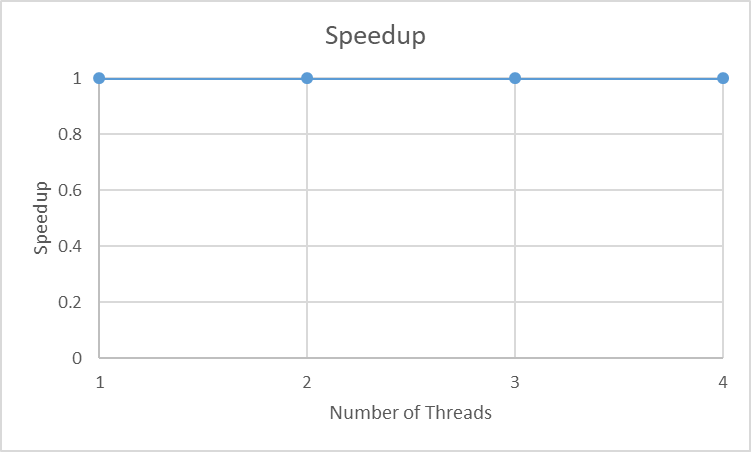
Jiyuan Lu [jl11046@nyu.edu](mailto:jl11046@nyu.edu)

Speedup over single-thread version for a problem of size 4 cities:

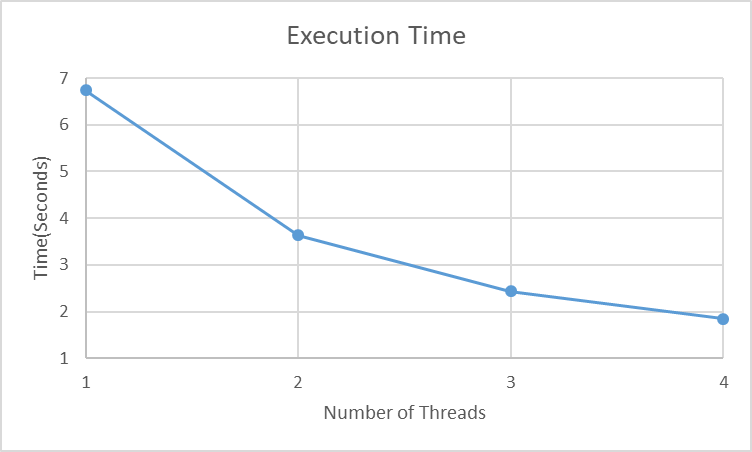


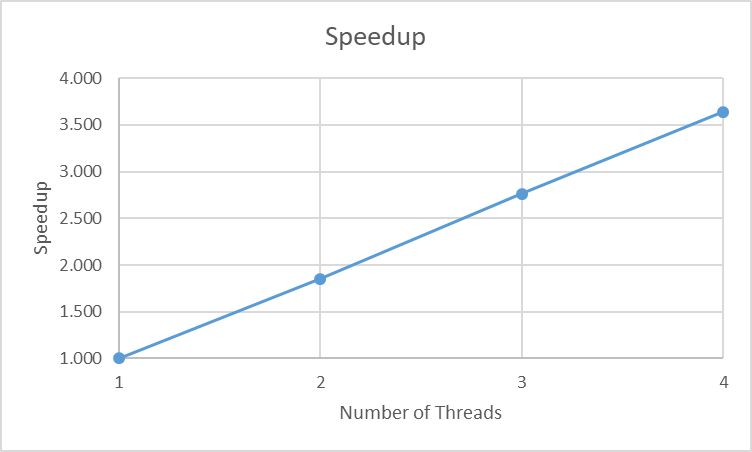


Speedup over single-thread version for a problem of size 8 cities:



Speedup over single-thread version for a problem of size 12 cities:

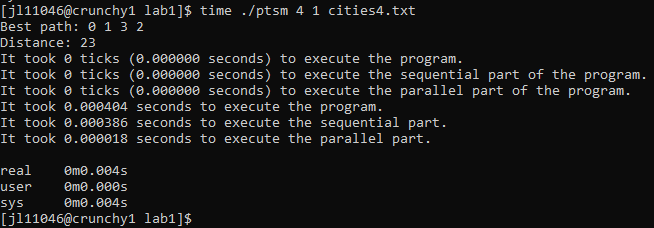




Analysis of the graphs

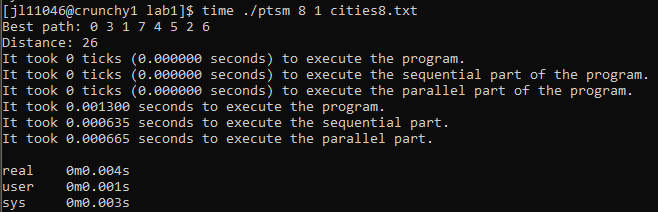
1. For 4 cities, there is no speedup. In fact, it takes more time when using multiple threads instead of a single thread. This is because the problem size is too small, and there is not too much to be parallelized (most of the program is the sequential part). And there is more time wasted on things like creating threads and synchronizations than the time saved from parallel processing

The following shows that the most part of the program is sequential for a problem size of 4:



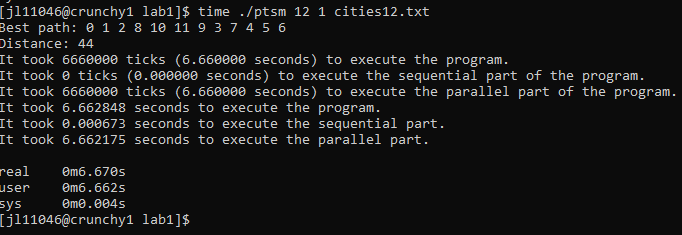
1. For 8 cities, there is no speedup. In fact, it takes about the same time when using multiple threads as using a single thread. This is because the problem size is still small. And the gain from parallel processing are close to the loss incurred from things like creating threads and synchronizations.

The following shows that the sequential part and the parallel part of the program are close for a problem size of 8:



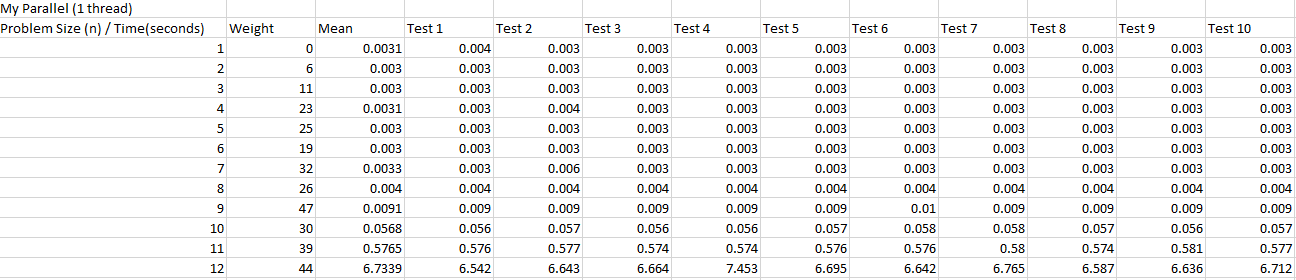
1. For 12 cities, there is significant speedup. The speedup is close to the number of threads used. (1.852 for 2 threads, 2.764 for 3 threads, and 3.640 for 4 threads). We can also calculate the efficiency of the program (0.926 for 2 threads, 0.921 for 3 threads, and 0.910 for 4 threads). This indicates that the problem is strongly scalable for a problem size of 12. This is because the problem size is now big enough and the most part of the program is parallelizable.

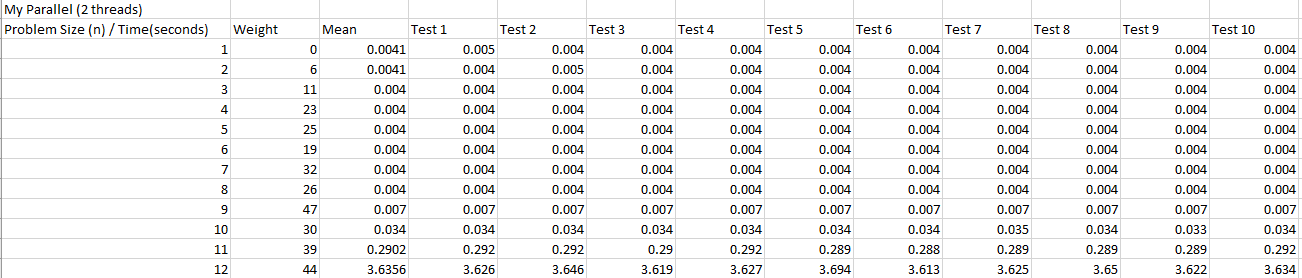
The following shows that the most part of the program is parallelizable for a problem size of 12:

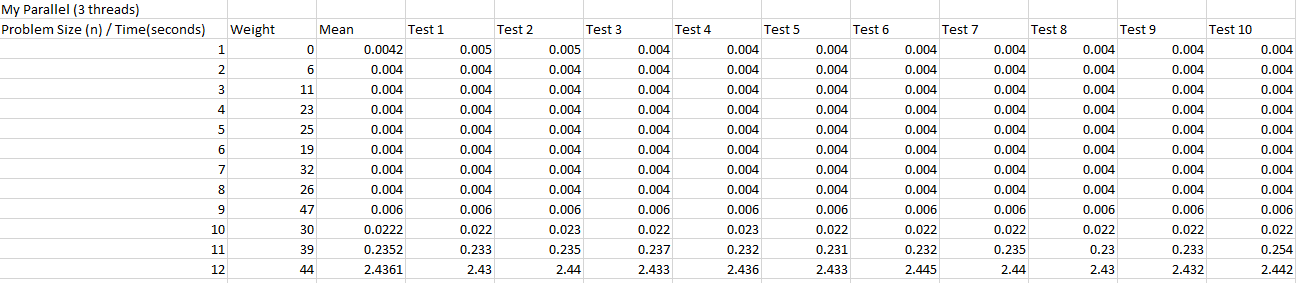


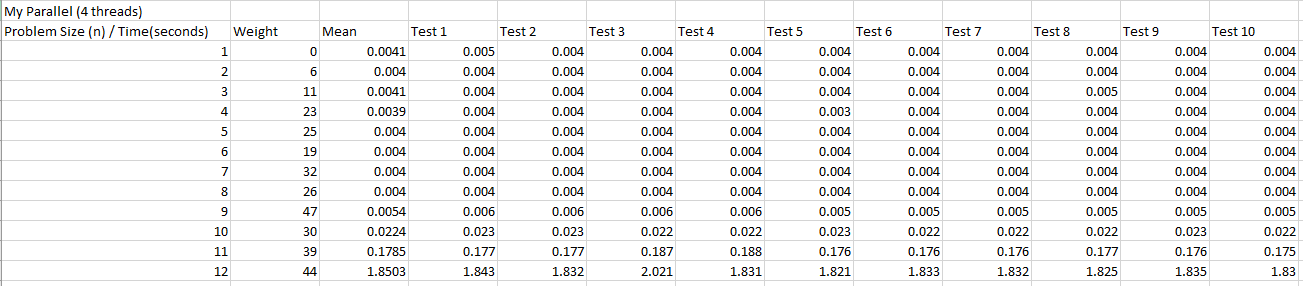
Data

* For execution time, I conducted 10 experiments for each (n, t) pair where n is the number of cities and t is the number of threads. Then I took the average of the 10 experiments.









Some Notes

* The program was built on crunchy1.cims.nyu.edu and compiled with gcc-9.2.
* Problem about the clock() function:
  + The clock() function as given in the Hint does not work well because the problem size is too small. It always gives me 0 ticks estimation for the sequential execution time. Therefore, I used omp\_get\_wtime() instead to get a finer-grained estimation of both the sequential and the parallel execution time of the program.
* Advantages over the reference implementation:
  + My program, even when executed sequentially, is much faster than the reference program when the problem size is above 10.
  + The reference program has invalid output for distance when the problem size is 1. I fixed the problem by outputting 0 for the distance when the problem size is 1.
* How I iterated on building the program:
  + I first implemented a serial version, then built upon it to get a parallelized version.
  + At first, there was a bug in my parallel program because I did not create private variables for each thread, and all threads were competing for the shared variable. The program was essentially sequentially executed, and the performance was even worse than a sequential version.