Thy Vu

CSS 534 Autumn 2016

Professor Munehiro Fukuda

Program Assignment #1: Documentation, Outputs, and Discussion

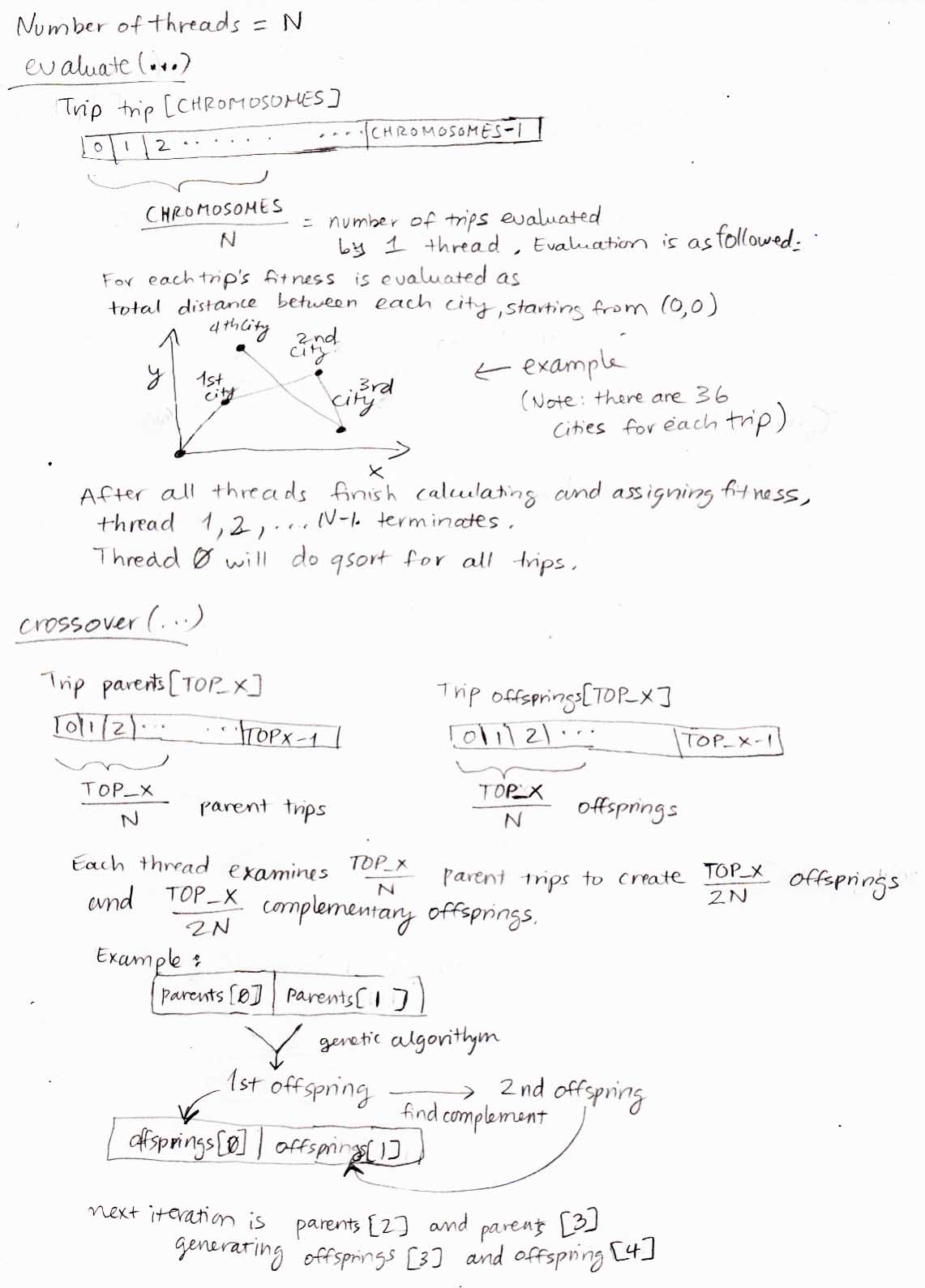
Documentation

This program gives a semi-optimal trip solution to traveling salesman problem based on genetic algorithms. In this scenario, the salesman has to travel to cities 'A' to 'Z' and '0' to '9'.

My parallelization strategy involves parallelizing the most outer loop with N threads for evaluate() function and the most outer loop with N threads for crossover() function. This is because those two functions are the most computational-intensive. Function mutate() is not parallelized because it uses rand() function, which makes a system call, every for loop iteration. Only one thread can make a system call at a time, so performance cannot be improved with multiple threads. Although crossover() function involves rand() when it needs to get a random unvisited city for child[i], that only occurs sporadically (not every iteration), specifically when child[i] has visited both parent[i] and parent[i+1]'s next cities. Hence, parallelizing still improves performance because the chance a thread has to wait for its turn to make system call is low and the wait is negligibly short.

In function evaluate(), each thread takes care of CHROMOSOMES/N trips. CHROMOSOMES is defined in Trip.cpp as 50000 by default, so if there are N=4 threads, thread 0 will evaluate trip[0] to trip[12499], and thread 1 will evaluate trip[12500] to trip[24999], and so on. There is variable trip[] is shared among threads because they update the fitness variables for each trip. After updating fitness variables, thread 1 to N-1 terminates, thread 0 (master) calls qsort() for the resulting trip[] array. Function qsort() cannot be parallelized because sorting requires valid fitness values.

In function crossover(), each thread takes care of TOP\_X/N parent trips to generate TOP\_X/(2N) offsprings and TOP\_X/(2N) complement offsprings. TOP\_X is defined in Trip.cpp as 25000 by default. If there are N=4 threads, thread 0 examines parents[0] to parents[12499] to generate offsprings[0] to offspring[12499]. Please see illustration. Shared variables between each thread is offsprings[] array because they will populate this array with Trip objects. Each thread has a copy of cities string and compCities string, which are complementary of each other and are used to generate the 2nd offspring based on 1st offspring.



Outputs

Outputs are generated on uw1-320-00 computer from my user folder.

Shortest trip:

1-thread execution:

shortest distance = 449.658

itinerary: V1YZHUE025CWSMPQBD3R7LAF9KGXNT48IO6J

4-thread execution:

shortest distance = 449.658

itinerary: V1YZHUE025CWSMPQBD3R7LAF9KGXNT48IO6J

Performance evaluation:

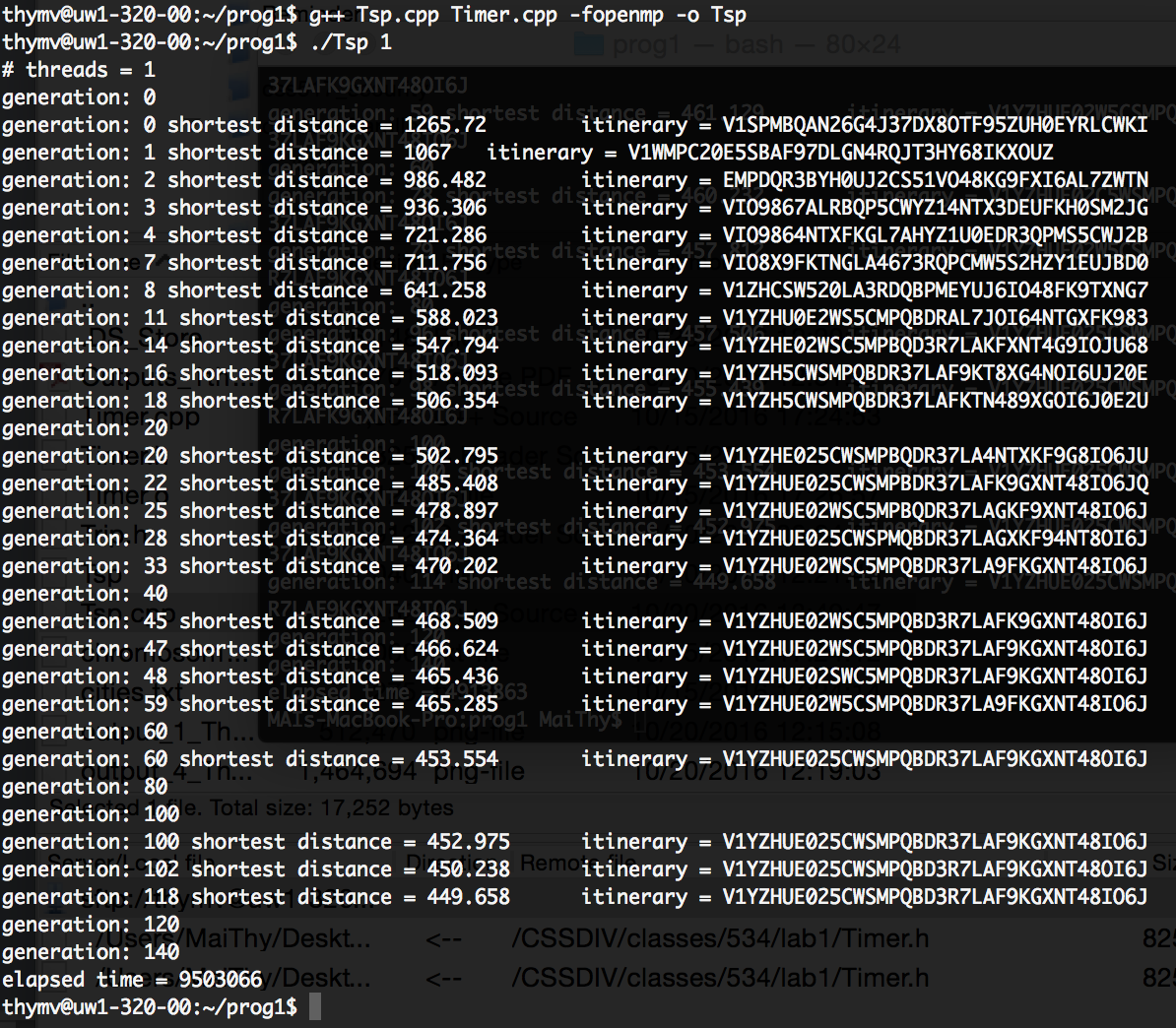
(1-thread-execution time) / (4-thread-execution time)

= 9503066/3580976

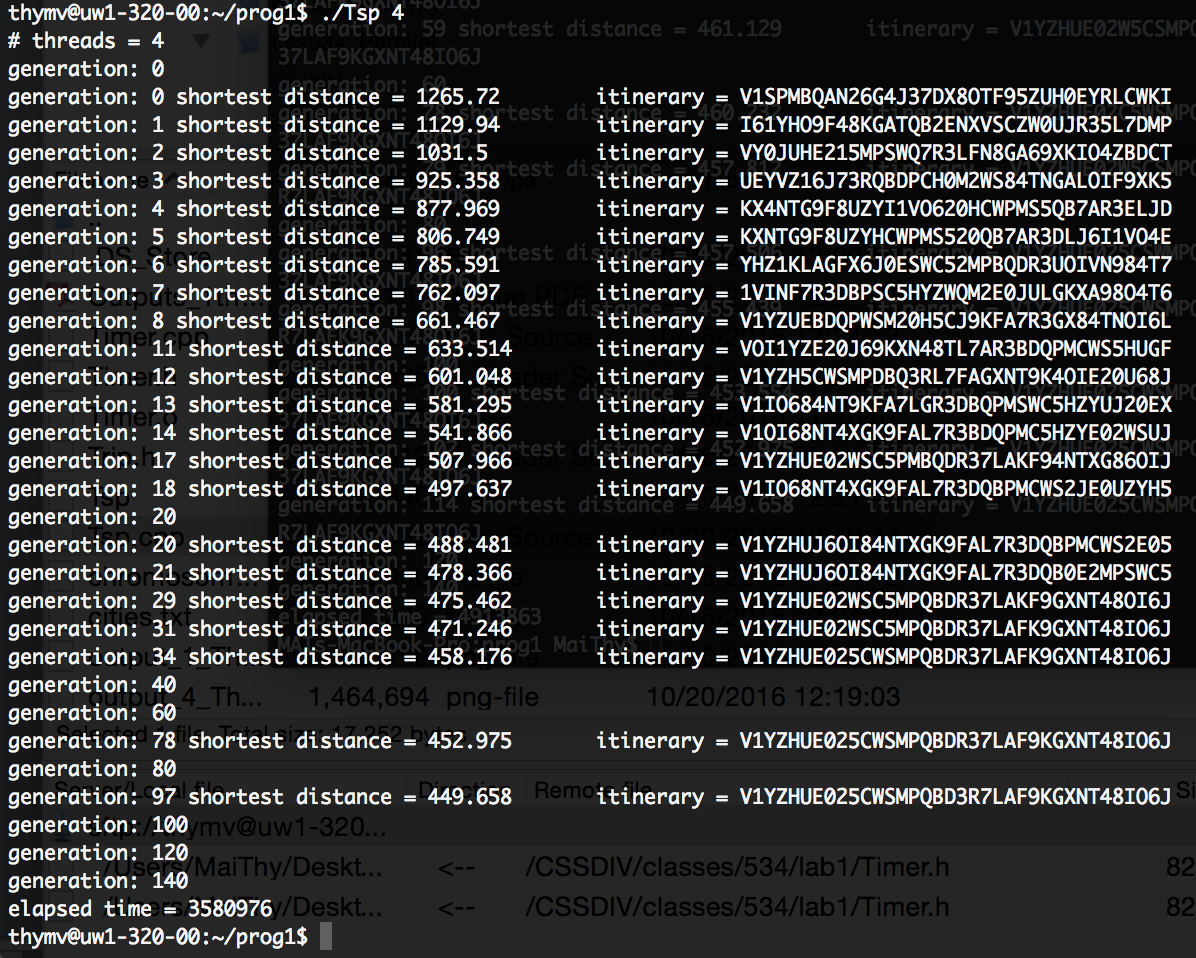
= 2.653

Below are the screenshots of the 2 runs.

Output for 1 thread:



Output for 4 threads:



Discussion

If chromosomes.txt is much more massive, parallelization of initialize() function is worth considering. Trip trip[CHROMOSOMES] can be populated in parallel. Depending on how massive it is (and number of processors present in the computing system), optimal number of threads may differ. For example, thread creation and termination overheads from having too many threads for a too-small chromosomes.txt may take away the time saved with parallelization. If chromosomes.txt is massively long, a possible 2-thread parallelization can populate trip[] from [0] and from middle index of the array at the same time.

The program as it is written right now is good for small number of threads. Further testing and possibly modifications would be needed if the user want to run a larger number of threads (on a computer with greater number of processors). There are also no mechanism from warning or stopping the user from putting inefficient or invalid arguments (e.g. running for 1000 threads on computer with 4 processors).

With enough modification, this program can be changed to accommodate distributed system. (Non-sorting part of) evaluate() function does not have to be limited to having number of threads being equal to the number of processors on one single computer. Its non-sorting part can be done by multiple computers and then sent back the result to one computer for sorting. The same can be said about crossover().

The current program cannot parallelize mutate() because it makes rand() calls every iteration. However, the function mutate() can be a great candidate for making a case for distributed computing. If we have multiple computers to work on mutate(), one thread per computer, then we can execute mutate() in parallel efficiently. The results will also be sent back to one computer, which is an overhead, but if number of offsprings[] are great enough, this method might be worth considering.