CSS 534
Program 1: Parallelizing Traveling Salesman Problem with OpenMP

Professor: Munehiro Fukuda
Due date: see the syllabus

1. Purpose
In this programming assignment, we will code a traveling salesman problem (TSP) based on the concept of genetic algorithms (GA) and parallelize it with OpenMP.

2. GA-based TSP
TSP is known as an NP-hard program that causes a computational explosion. For instance, finding the shortest route through 36 cities needs to examine 36! (= 36 × 35 × … × 1) combinations. GA is quite effective to reduce TSP's computation time while reaching a semi-optimal trip (but no the shortest path). Consider a travel through 36 cities, each named with one of the 36 characters such as A~Z and 0~9. `ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789` is one possible trip. In GA, this string and each city in it can be considered as a chromosome and a gene respectively. We will first generate 50,000 different trips or chromosomes, and then repeat 150 iterations or so-called generations, each including:

(1) **evaluate( ):** evaluates the distance of each trip and sorts out all the trips in the shortest-first order. Memorize the current shortest trip as a tentative answer if it is shorter the previous.
(2) **select( ):** selects the shortest 25,000 trips as parents.
(3) **crossover( ):** generates 25,000 off-springs from the parents. More specifically, we spawn a pair of child[i] and [i+1] from parent[i] and [i+1].
(4) **mutate( ):** randomly chooses two distinct cities (or genes) in each trip (or chromosome) with a given probability, and swaps them.
(5) **populate( ):** populate the next generation by replace the bottom 25,000 trips with the newly generated 25,000 off-springs.

3. Crossover Algorithm
The key to GA-based TSP is to design a suitable crossover algorithm. A typical crossover generates child[i] by combining the first half of parent[i]'s genes and the last half of parent[i+1]'s genes, whereas gives child[i+1] the last half of parent[i]'s genes and the first half of parent[i+1]'s genes. However, this crossover does not work in TSP. For example in a TSP program for visiting only eight cities, consider two parents:

parent[i] = `ABCDEFGH` parent[i+1] = `HGABFECD`

Their children will be::

child[i] = `ABCDFECD` child[i+1]=`HGABEFGH`

Child[i] and [i+1] will end up with revisiting CD and GH respectively. To address this problem, we will use a greedy crossover algorithm:

We select the first city of parent[i], compares the cities leaving that city in parent[i] and [i+1], and chooses the closer one to extend child[i]'s trip. If one city has already appeared in the trip, we choose the other city. If both cities have already appeared, we randomly select a non-selected city. Thereafter, we generate child[i+1]'s trip as a complement of child[i].

In the same example with eight cities: ABCDEFGH, each city's complete is:

City	Complement	City	Completement
A	H	E	D
B	G	F	C
C	F	G	B
D	E	H	A

If child[i] includes ABHEGDFC, child[i+1] should be HGADBECF.

4. Parallelization

The most computation-intensive portions are eveluate( ) and crossover( ), both including large nested for-loops. You can parallelize them using "#pragma omp parallel for". More ambitious is to parallelize an entire computation in each generation from evaluate( ) to populate( ) with multithreads, where we can divide 50,000 trips by the number of threads (say N threads), each independently working on the same generation of 50,000/N trips that generates new 25,000/N children. In this method, you need to exchange all trips among all the N threads at the end of each generation, (i.e., populate( )) but not necessarily every iteration. Furthermore, an implementation of trip exchanges is up to you. As far as your program finds a correct and reasonably short trip, you can try any parallelization techniques.

5. Program Structure

Your work will start with modifying the template that the professor got prepared for. Please login uw1-320-lab.uwb.edu and go to the ~css534/prog1/ directory. You can find the following files:

Program Name	Description
chromosome.txt	Includes 50,000 different trips or chromosomes. Copy it in your working directory and use this data without modifying the contents.
cities.txt	Includes the names and (x, y) coordinates of 36 cities to visit. Copy it in your working directory and use this data without modifying the contents.
compile.sh	Is a shell script to compile all the professor's programs. Generally, all you have to do for compilation is: `g++ *.cpp –fopenmp –o Tsp`
initialize.cpp	Is a source program that generates chromosome.txt and cities.txt. You don't have to use it.
initialize	Is executable code that generates chromosome.txt and cities.txt. You don't have to use it.
Timer.h, Timer.cpp, Time.o	Is a program used in Tsp.cpp to measure the execution time. Copy them in your working directory.
Trip.h	Defines all parameters necessary. Copy it in your working directory and use this header file without changing all constants except `MUTATE_RATE`. In other words, your final performance evaluation must use 50,000 chromosomes, visit 36 cities, generates 25,000 children in each generation, and repeats 150 generations.
Tsp.cpp	Is the main program that executes this GA-based TSP. It has already implemented initialize( ), select( ), and populate( ). You need to implement evaluate( ), crossover( ), and mutate( ). Additionally, for better parallelization, you can modify any portion of Tsp.cpp. However, make sure that your Tsp.cpp uses chromosome.txt and cities.txt. Copy it in your working directory and modify it. Or you can redesign Tsp.cpp from sratch.
Tsp	Is executable code that runs the professor's GA-based TSP.

EvalXOverMutate.cpp	Is a key answer and read/write-protected. It implemented evaluate( ), crossover( ) and mutate( ) as well as parallelize them with OpenMP.
EvalXOverMutate.o	Is an object module that can be linked to Tsp.cpp upon a compilation.

6. Statement of Work

Follow through the steps described below:

Step 1: Implement evaluate( ), crossover( ), and mutate( ) to complete this GA-based TSP program.
Step 2: Parallelize the program with OpenMP and tune up its execution performance as much as you like.
Step 3: Conduct performance evaluation and write up your report.

You can run the professor's program as follows for the purpose of comparing yours with it:

```
css534@uw1-320-15:~/prog1$ Tsp 1
# threads = 1
generation: 0
generation: 0 shortest distance = 1265.72      itinerary = V1SPMBQAN26G4J37DX8OTF95ZUH0EYRLCWKI
generation: 1 shortest distance = 1083.52      itinerary = VG4XAK3R78TZMBW5H0EYU12DIN96OJPCSQLF
generation: 2 shortest distance = 1009.03      itinerary = V120EYUJTZMPCSBW5HQLFG6OXAK3R7DIN489
...
generation: 113 shortest distance = 450.238    itinerary = V1YZHUE025CWSMPQBD3R7LAF9KGXNT48OI6J
generation: 118 shortest distance = 449.658    itinerary = V1YZHUE025CWSMPQBD3R7LAF9KGXNT48IO6J
generation: 120
generation: 140
elapsed time = 26234614


css534@uw1-320-15:~/prog1$ Tsp 4
# threads = 4
generation: 0
generation: 0 shortest distance = 1265.72      itinerary = V1SPMBQAN26G4J37DX8OTF95ZUH0EYRLCWKI
generation: 1 shortest distance = 1083.47      itinerary = I61YHO9F48KGATL7UJR3BQ2CZWS05MDENXVP
...
generation: 79 shortest distance = 450.238     itinerary = V1YZHUE025CWSMPQBD3R7LAF9KGXNT48OI6J
generation: 80
generation: 91 shortest distance = 449.658     itinerary = V1YZHUE025CWSMPQBD3R7LAF9KGXNT48IO6J
generation: 100
generation: 120
generation: 140
elapsed time = 12452981
css534@uw1-320-15:~/prog1$
```

Your minimum requirements to complete this assignment include:

(1) The shortest trip in your program should be equal to or less than 449.658.
(2) The performance improvement with four threads in your program should be equal to or larger than 26234614 / 12452981 = 2.1 times.

7. What to Turn in

This programming assignment is due at the beginning of class on the due date. Please turn in the following materials in a softcopy that should include:

(1) Your report in PDF or MS Word
(2) Source code (either within your report or separate .h and .cpp files)
(3) Execution outputs (either within your report or separate .jpg, .pdf, .tif, or .txt files)

The professor's preference is all in one report.

Criteria	Grade
Documentation of your parallelization strategies including explanations and illustration in <u>one or two pages</u>.	20pts
Source code that adheres good modularization, coding style, and an appropriate amount of commends.	25pts

• 25pts: well-organized and correct code receives • 23pts: messy yet working code or code with minor errors receives • 20pts: code with major bugs or incomplete code receives	
Execution output that verifies the correctness of your implementation and demonstrates any improvement of your program's execution performance. • 25pts: Correct execution and better results than the two requirements (the shortest trip with less than 449.658 AND performance improvement with more than 2.1 times) • 23pts: Correct execution and better results than one of the two requirements (the shortest trip with less than 449.658 OR performance improvement with more than 2.1 times) • 20pts: Correct execution and the two requirements just satisfied (the shortest trip with 449 ~ 500 and performance improvement with 1.8 ~ 2.1). • 18pts: Correct execution and better performance improvement but only one the two requirements satisfied. • 15pts: Correct execution and better performance improvement but none of the two requirements satisfied. • 13pts: Correct execution but little performance improvement • 10pts: Wrong execution	25pts
Discussions about the parallelization, the limitation, and possible performance improvement of your program in one page.	25pts
Lab Sessions 1 Please turn in your lab 1 by the due date of program 1. Your source code, execution outputs, and brief comments are required.	5pts
Total Note that program 1 takes 15% of your final grade.	100pts