Multicore Processors: Architecture & Programming
Lab 2

In this lab you will implement a method for solving a *modified* version of the traveling salesman problem in OpenMP. The challenge here is to find the best algorithm that can be parallelized, and try to get a speedup even if there is a overhead for I/O like reading from a file.

What is the problem definition?

You have a group of n cities (from 0 to n-1). The traveling salesman must start from city number 0, visit each city *once*, and does **not** have to come back to the initial city. You will be given the distance between each city (cities are fully connected to each other). You must pick the path with shortest distance. Note that there may be several paths that satisfy the above conditions, you just need to find one of them. Also, the shortest path is in terms of distance, not in terms of number of cities.

What is the input?

The input to your program is a text file that contains an NxN matrix, one row per line, representing the distance between any two cities. Your program will be called ptsm (for parallel tsm). The command line then will be:

./ptsm x t filename.txt

Where :
x is the number of cities
t is the number of threads
filename.txt is the file that contains the distance matrix

What is output?

You program must output, to screen, the best path you found and the total distance.
Here is an example output:

```
Best path: 0 1 3 4 5 2
Distance: 36
```

This means the best path your program found is 0->1->3->4->5->2 with total distance of 36.

How will you solve this?

This problem is a combinatorial optimization problem (n! possible solutions for n cities). This means an exact optimal solution will take very long time.

We will not test your submission with more than 12 cities and we will not test with more threads than cities.
For example, if we have 10 cities and we have two threads. We always start with city 0. Thread 0 will be responsible for exploring paths that start from 0 and go to either 1, 2, 3, 4, or 5; while thread 1 will explore threads going from 0 to 6, 7, 8, or 9. Each thread can use recursion to check the paths. At the end, each thread will pick the shortest path from the paths it has explored. The final answer is the shortest path found among all threads.

The above is just a very high-level description. But, in this lab, you have the freedom to modify the algorithm and pick one that gives you the best speedup possible. You are not allowed to use any third-party libraries except the math's library, if you want.

To help you:

We are providing you with two executables:

- **./tsm** : This program generates test files (usage: ./tsm numcities). The output of this program is cities*x*,txt that contains the matrix of the x city distances. It will also print a (sub) optimal solution on the screen. The file citiesx.txt is a text file that you can open and take a look.
- **./tsmoptimal** : This programs prints the optimal solution on the screen (usage: ./tsmoptimal x cities*x*.txt) for the x cities problem. This is a sequential program. So, don't try to use it with a large number of cities or it will take forever. Do not compare the time taken by your program with that of tsmoptimal. But compare your program with one thread with your program with more than one thread.

Note: The above two files are provided in binary format. You cannot execute them on your laptop, even if you have Linux as they have been compiled on crunchy machines with the libraries there. In order to execute them do the following:

1. Move them to one of the crunchy machines.
2. Execute this command: *chmod 711 ./tsm* (or ./tsmoptimal)
3. To execute any of the programs, type: ./tsm (or ./tsmoptimal) and it will print how to use it on the screen.

The report

You need to provide a report that includes the following:

- **Experiment 1**: A graph that shows speedup over single-thread version for a problem of size 4 cities, another graph of problem of size 8 cities, and a third graph for 12 cities. For each graph, the x-axis is the number of threads. For the graphs of four cities, the number of threads is 1 and 2. For eight and twelve cities, the number of threads is 1, 2, and 4.
- **Experiment 2**: Fix the number of threads to four. The y-axis is the speedup relative to one thread. The x-axis is the number of cities. In an increment of 2, starting with 6, show the speedup for 6, 8, 10, … x. Where x is the smallest

number of cities where four threads show a speedup > 2. If x turns out to be 6 or 8, for example, then stop at that number.
- Make a list of all observations you found from the above two experiments. And, for each observation, write 1-2 sentences explaining why we have this observation.

Note: You will measure the timing of the program using the *time* command. You need the *real* part of it. Feel free to use omp_get_wtime() while optimizing your code. But this is for your own knowledge. Do not report that number.

Regarding compilation

- Name your source file: ptsm.c and add as many comments as you can. This will help us give partial credit in case your program does not compile.
- Do the compilation and execution on crunchyx (x = 1, 3, 5, or 6) machines.
- Use a version of gcc that is 8 or higher on crunchy by typing, for example: module load gcc-9.2

What to submit?

Add the source code ptsm.c as well as the pdf file that contains your graphs to a zip file named: lastname.firstname.zip
Where lastname is your last name, and firstname is your first name.

Have Fun!