Lab 1 Report

1 Introduction

This lab solves a modified traveling salesman problem using parallelized algorithm with OpenMP. The program is written in C++. The files include a serial version of the solution, `stsm.cpp`, and a parallelized version of the solution, `ptsm.cpp`. The shared header file is `tsm.h`.

To compile and run the serial version, simply do the following:

```
g++ -std=c++11 -g -Wall -O2 stsm.cpp -o stsm
./stsm <num_cities> <filename>
```

To compile and run the parallelized version, simply do the following:

```
g++ -std=c++11 -g -Wall -O2 -fopenmp ptsm.cpp -o ptsm
./ptsm <num_cities> <num_threads> <filename>
```

To use the latest version of `gcc` on CIMS machines, the following command is needed prior to compiling the program:

```
module load gcc-6.4.0
```

Where `<num_cities>` is the number of cities in the file, `<num_threads>` is the number of threads, and `<filename>` is the file that contains the distance information.

2 Algorithm Design

The serial version was simply finished by recursive DFS. The parallelized solution, however, is modified in the following aspects for better performance:

- Only two variables are shared: the minimum distance and the array of route that leads to the minimum distance.
- Instead of distributing the for loop directly, the first loop is manually expanded, to make the distribution of work more even. For example, for 10 cities, since the first city must be city 0, the 9 for loops will be distributed among several threads. This may lead to uneven work of threads, for instance, if 8 threads are assigned, then one thread must handle two loops while the others only need to handle one. This is uneven distribution of work. Therefore, after manually expanding the first loop, for 10 cities, the loops that are distributed among threads are $9 \times 8 = 72$ loops. This higher granularity will make the total work distributed more evenly.
- Static scheduling is used to lower the overhead of schedule directive.
- To reduce the size of critical section, the initial comparison of minimum distance is conducted via atomic read. The minimum section only contains necessary code to update the minimum distance and best route.

3 Speed-up Analysis

The following speed-up analysis was conducted on `crunchy1` server of CIMS. The serial program used is `stsm.cpp` provided and the parallel program used is `ptsm.cpp`.

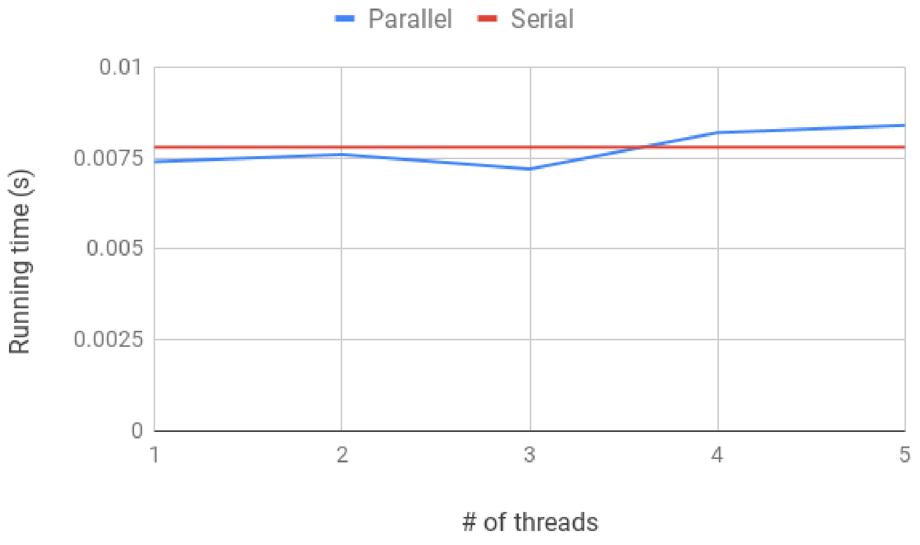

Fig. 1 | Running time analysis for 5 cities

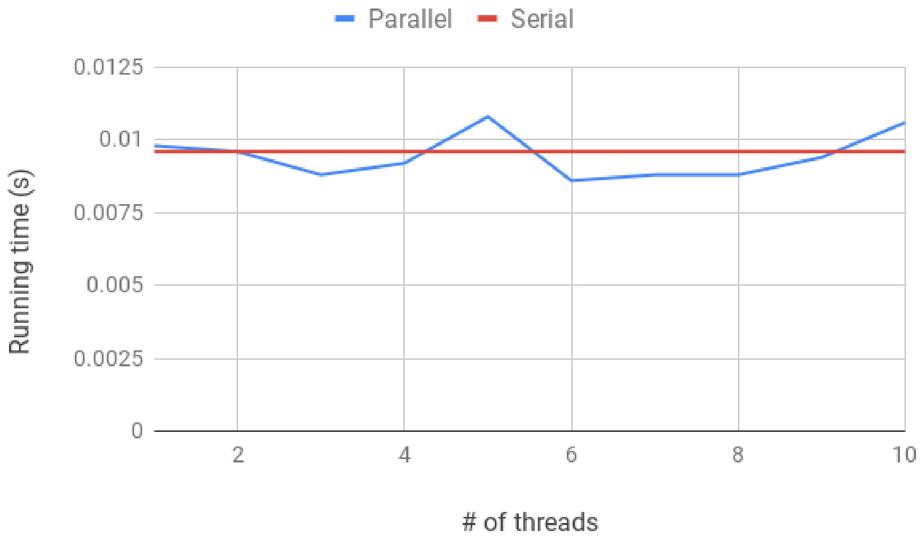

Fig. 2 | Running time analysis for 10 cities

As we can see from the graphs, there is not much performance gain from the parallelized version compared to the serial version. This is because the problem size is relatively small and the serial version is already very fast, consuming very little CPU time. The `crunchy1` server has 64 cores with 2.1 GHz CPU, which is fast enough to solve the problem with very little time. In the outputs of `time` command, the user time is much smaller compared to the system call time: for 5 cities, the user time is usually from 0.001s to 0.002s, while the system call time is usually from 0.003s to 0.006s; for 10 cities, the user time is usually from 0.003s to 0.006s, while the system call time is usually from 0.004s to 0.008s. The system call time will become even larger when the number of threads becomes larger. Therefore, the overhead of parallelized program (thread creation and termination, critical section and communications) makes the performance enhancement meaningless. By enlarging the problem size, the performance gain of parallelized program would be more obvious. To prove this, the following experiments were carried out.

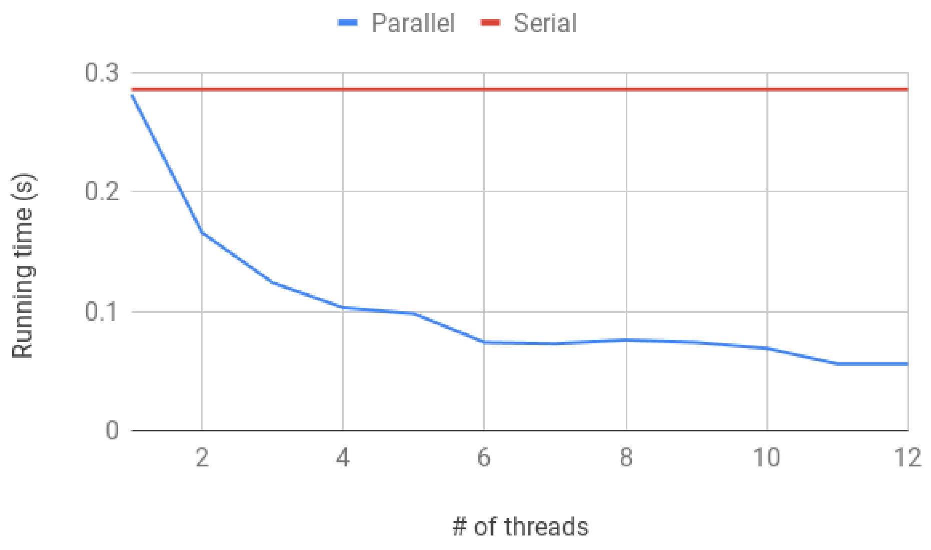

Fig. 3 | Running time analysis for 12 cities

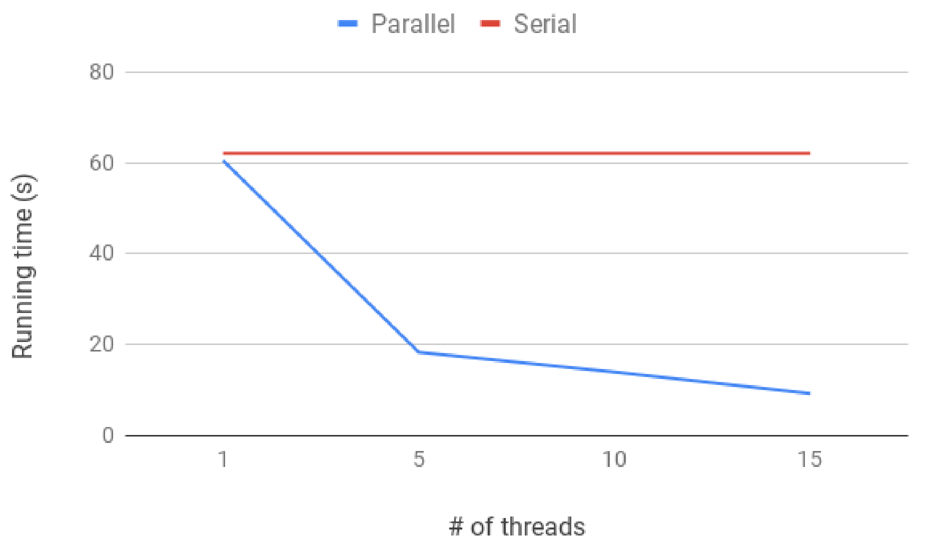

Fig. 4 | Running time analysis for 15 cities

As we can see from the above graphs, when increasing the problem size to 12 cities and 15 cities, the advantage of parallelism takes in charge. The overhead of parallelism becomes insignificant compared to the performance gain. This is because when the number of cities increases, the size of problem increases exponentially. If the number of cities is n, then the problem size could be $n!$ at most. The problem becomes dramatically larger, where the serial solution cannot be finished in relatively short time and the CPU time (user time) takes the most part of the program running time. In this way the parallelized program becomes much more favorable compared to the serial one. The maximum performance gain here is 5 times faster for 12 cities and 6.5 times faster for 15 cities.

4 Specifications

For the CIMS servers, the server used in this lab is `crunchy1`. Checking the file `/proc/cpuinfo`, the important specifications are listed below:

- CPU: Four AMD Opteron 6272 (2.1 GHz) (64 cores)
- Memory: 256 GB
- OS: CentOS 7

5 Conclusion

The serial and parallel programs are both analyzed, and the algorithm used in parallelized program will be able to get huge performance gain given the problem size is relatively large. For 5 and 10 cities the performance gain is insignificant, but for 12 and 15 cities the parallelized version will become much faster. For detailed analysis, see the Section 3, *Speed-up Analysis*.