# INF3201- Parallel Programming

Assignment 2 - Shared Memory

18 September - 8 October 2020

## 1 Traveling Salesman problem (TSP)

The Travelling Salesman Problem (TSP) is the challenge of finding the shortest route for a person to take given a list of specific destinations. It is a well-known algorithmic problem in the fields of computer science and operations research [2]. Figure 1 depicts visualization of solved TSP for 7 cities.

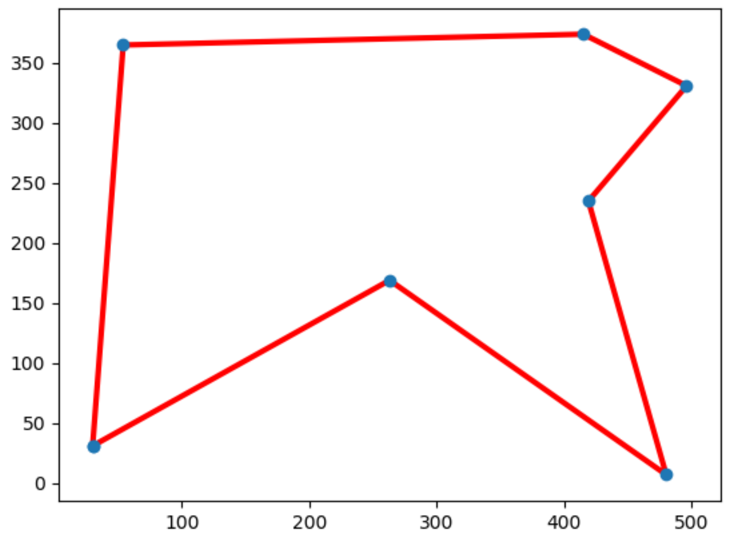

Figure 1: TSP visualization

## 2 Task

Implement a parallel branch-and-bound algorithm [5], [6] to solve the Traveling Salesman Problem [4]. You can base your algorithm on the sequential version given to you. Your code should use OpenMP [3]. The code should be able to run on any number of CPU cores, and the results must always be the same regardless of the number of cores used. If you want to implement another algorithm than the branch-and-bound algorithm to solve the Traveling Salesman Problem, you should contact us first.

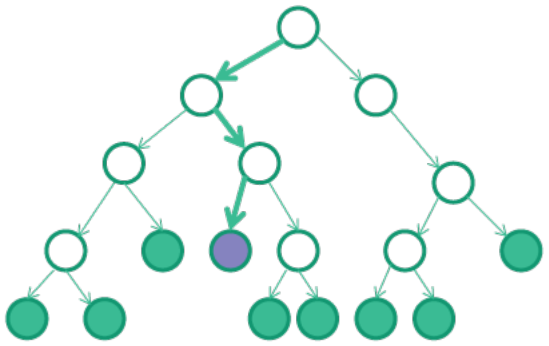

Figure 2: Branch and bound example [5]

Any optimization may be applied, but the result should be correct (ie. give an reasonable answer) and independent of the number of CPU cores that are used to achieve the solution. Any optimization that is applied must be thoroughly described in the report .

## 3 Precode

In the repository you can find a directory called `Code`. Inside that directory there is another one called `FrostTrap` where you will find a few files.

- `RaceTrap.c` - contains precode for the assignment. When you compile the given file you should be able to run it. The executable takes one optional parameter as an input: `./RaceTrap [dump]`
  `[dump]` determines whether the executable should save data that can be used to visualize TSP - check file `plot_data.py` for more details.
- `plot_data.py` - python3 code used to visualize TSP.
- `route.dat` - contains city coordinates. First line in the file is used to specify how many locations is used by the sequential code.
  **Default value: 10**.

## 4 uvcluster

At the Computer Science department we have a small cluster of nodes that can be used to run the parallelized solutions. The cluster has a distributed file system which means that you will need to copy your files only to the frontend server (uvcluster.cs.uit.no) in order to access them from all the other nodes in the cluster.

In order to login to the cluster please use the following command (in linux): `ssh abc123@uvcluster.cs.uit.no` where abc123 is your UiT login.

Please remember to run a command `scl enable devtoolset-8 bash` in order to enable gcc 8 for your current session, otherwise you will be using older version of gcc (4.8.5) and OpenMP.

Please make sure that you are familiar with welcome message from the cluster - see Listing 1. For the given assignment you should list the hardware list(2), pick up a node or two (ex. compute-0-10), check the load of the selected nodes(3), ssh to the selected node (ex. abc123@compute-0-10) and run your code there. Please remember that you can only ssh to the cluster nodes when you are already logged to the frontend server.

```
1 List nodes available right now: /share/apps/ifi/available-nodes.sh
2 List the cluster hardware: /share/apps/ifi/list-cluster-static.sh
3 List nodes by load: /share/apps/ifi/list-nodes-by-load.sh
4 List all your processes: /share/apps/ifi/list-cluster-my-processes.
    sh
5 Clean out all your processes: /share/apps/ifi/cleanup.sh
```

Listing 1: uvcluster welcome message

## 5 Requirements

- Implement parallel version of the given TSP code:
  - write code using C/C++ and OpenMP,
  - write code in understandable way and make sure the code is well-commented,
  - your solution can not use any shortcuts that reduce functionality of the sequential code,
  - your parallelized code is supposed to produce the same `route length` as the sequential code for given number of destinations.
- Analyze your solution:
  - Profile the sequential code and calculate maximum theoretical speedup using Amhdal's law. You might use Gprof[1] in order to complete this requirement.
  - Profile and trace your parallel solution. How well does it scale with regard to number of CPU cores and the problem size (number of cities to travel)?
- Write report:

- Required sections for the report are as follow (section names might be slightly different):
  - `Introduction` - describe your task.
  - `Sequential solution analysis` - profiling, speedup, expectations, assumptions.
  - `Design` - description of your parallelized techniques - your approach to parallelization (Have you adequately explained your solution?).
  - `Experiments` - describe your experiments with different problem size (see description of route.dat file).
  - `Discussion` - discuss positive and negative sides of your solution. Compare theoretical maximum speedup with your results in details. (Have you critically evaluated your solution? Have you made your assumptions clear and separate from your measurements?).
  - Summary.
- Did you find any unexpected results or behaviour of the system? Try to investigate and find a possible explanation. This might require tweaking the source code or experiment parameters to test ideas and compare results.
- Make your figures clear and understandable. Remember about references, captions and axes descriptions.
- Remember to take into consideration the hardware you are using.

## 6 Hand-in

You will work in groups of 2 for the assignment. GitHub classroom is used as a hand-in platform for the course. You should commit and push your changes as often as possible (and reasonable).

**Please remember to push all your changes before 8 October 2020 23:59:59**. Any changes pushed to your repository after that deadline will not be evaluated.

## 7 Route length

To make sure that your parallel code works correctly for given number of destinations and to save you some time please use the attached Table 1 for comparison. The execution time might be used to evaluate the efficiency of your parallel code.

| Number of cities | Route length | Execution time |
|---|---|---|
| 17 | 2100.362805 | 44ms |
| 18 | 2207.508442 | 48ms |
| 19 | 2294.401450 | 137ms |
| 20 | 2302.638499 | 273ms |
| 21 | 2318.572285 | 918ms |
| 22 | 2319.472071 | 1s 740ms |
| 23 | 2322.381384 | 4s 336ms |
| 24 | 2342.199254 | 40s 775ms |
| 25 | 2381.382155 | 2m 40s 714ms |
| 26 | 2406.833154 | 7m 42s 61ms |

Table 1: Results for the efficient implementation of TSP using branch-and-bound algorithm and OpenMP. CPU used: Intel Core i7-8550U.

## References

[1] Himanshu Arora. Gprof tutorial – how to use linux gnu gcc profiling tool. `https://www.thegeekstuff.com/2012/08/gprof-tutorial/`, 2012.

[2] Suzanne Ma. Understanding the travelling salesman problem (tsp). `https://blog.routific.com/travelling-salesman-problem`, 2020.

[3] Peter S. Pacheco. Chapter 5 - shared-memory programming with openmp. In Peter S. Pacheco, editor, *An Introduction to Parallel Programming*, pages 209 – 270. Morgan Kaufmann, Boston, 2011.

[4] Peter S. Pacheco. Chapter 6.2 - parallel program development - tree search. In Peter S. Pacheco, editor, *An Introduction to Parallel Programming*, pages 299 – 308, 316 – 319, 337 –338. Morgan Kaufmann, Boston, 2011.

[5] NTNU Trondheim. Milp algorithms: branch-and-bound and branch-and-cut. `https://www.itk.ntnu.no/_media/fag/fordypning/tk16/milpalgorithms.pdf`.

[6] Stanford University. Branch and bound methods. `https://web.stanford.edu/class/ee364b/lectures/bb_slides.pdf`.