Schedule

Start	18.10.2018
Parallelization using standard APIs	03.12.2018
Parallelization using MPI	14.01.2019

Comparison of parallelization APIs

Travelling salesman problem

Lukasz Marcul
<dev.marcul@gmail.com>

Warsaw, 7th January 2019

CONTENTS

1. Introduction

The cult idea of travelling salesman problem (TSP) which describes **seeking the most optimal connection between all goals of travelling salesman such a way that to visit each of them, one must take e.g. the distance/price/time of travel into consideration**, has been fascinating many amateurs and professionals.

Looking from the perspective of pragmatics who only need to learn some thread APIs and the nature of parallelization, we will get away from detailed analysis and tough processor's time optimization. That's why we are going to take greedy algorithm, which is the most efficient (sic!) from widely known solutions when it comes to utilizing working hours. Out main task is to compare popular thread APIs as well as languages used. We will focus on *OpenMP* (for both *C and C++*), *MPI and std::threads* (*C++*), *Pthreads (C)*.

But wait. What we really want to implement?

1.1. The essence

Quoting precisely:

> *There are n cities, which should be visited by travelling salesman and the distance between every pair of them. Find the shortest path which comes from the start city and goes through all others exactly one time and goes back to start. In other words, minimize the criterion:*

$$\sum_{j=1}^{n-1} c_{\pi(j)\pi(j+1)} + c_{\pi(n)\pi(1)}$$

where: $n \in \mathbb{Z}^+ - number\ of\ cities$

$c_{ij} \in \mathbb{R}^+ - the\ distance\ between\ neighbours\ i, j\ \in \{1, \dots, n\}, i \neq j, assuming\ c_{ij} = c_{ji}.$

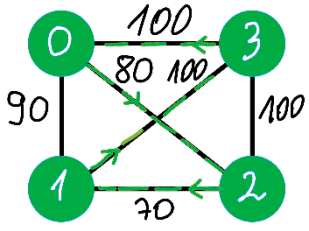

Fig. 1. The optimal solution for example with 4 cities.

From the content of tasks, we need to consider symmetric travelling salesman problem (the distance between neighbors is the same in both directions).

Let's say we need to find the best path from starting at city 0. For the greedy method we assume that all permutations of set of n elements should be considered, that is why we have $n!$ possibilities. But there is easy way to reduce the task (we will not implement that) to consider $\frac{(n-1)!}{2}$ ways only because $c_{ij} = c_{ji}$ and the best path is the same starting from each city. For example (Fig. 1): we only need to consider following sequences: $(0,1,2,3,0)$, $(0,1,3,2,0)$, $(0,2,1,3,0)$, which results in $(0,2,1,3,0)$ with len of 350, as the best path.

1.2. Algorithm

There were two approaches implemented: sequential and parallel.

The simplicity of the first one overcomes the relative speedup of the second. Let's note that for the parallel one, the sequential problem was separated using *divide and conquer* rule. Each of threads had an individual set of permutations to check (provided by the allocation method). After completion of task the thread received another permutations to check and so on. When all permutations were checked, the sequential calculation of $n-1$ partial best paths resulted in extracted best path. Consider the following diagrams.

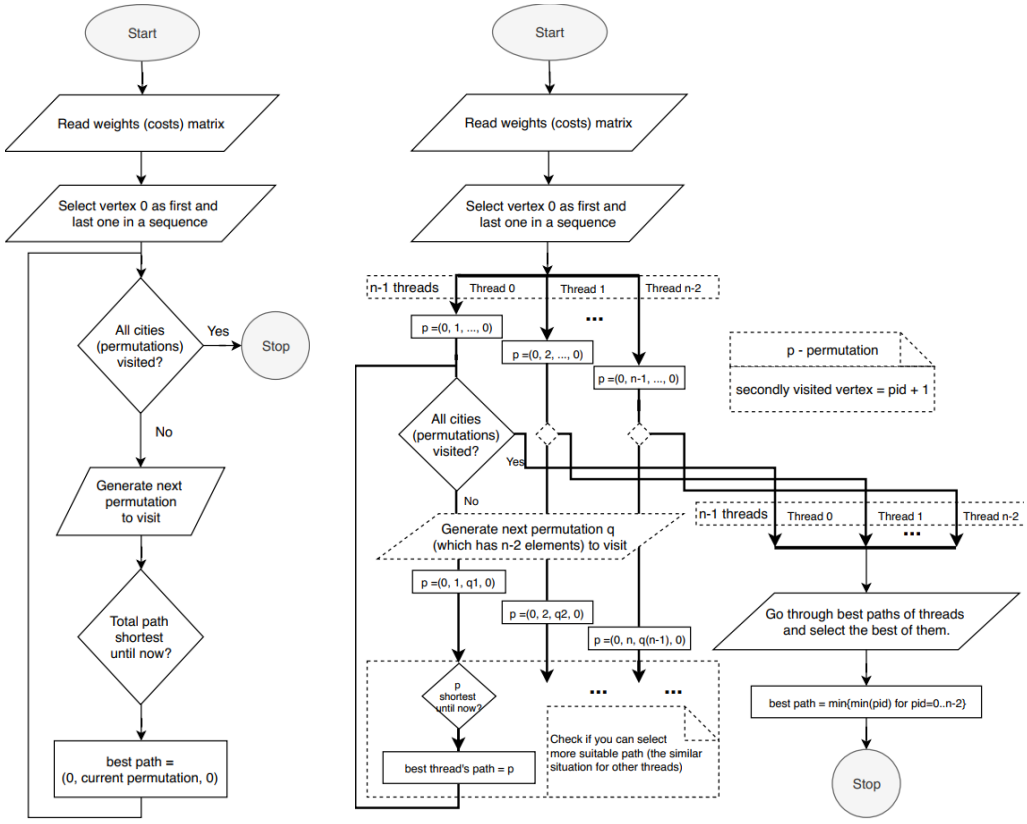

Fig. 2. Block diagrams of greedy algorithm. Sequential (on the left) and parallel (on the right).

In all approaches the similar schema of problem separation for parallelization was used.

1.3. Technologies

Build and run

compiler	GCC 7.3.0
build config	*CMake 3.9*
processor	*Intel Core i7-3610QM 2.30 GHz , 4-core*
memory	8 GB RAM DDR3, SSD (500/450 MB/s)
environment	*Python 3.6.7*, CLion 2018.3.1, Ubuntu 18.04.1 64-bit

Implementation

Sequential	C++17, C11
OpenMP	C++17, C11
std::thread	C++17
OpenMPI	C++17
Pthreads	C11

2. Tests

Were executed in the normal working conditions of typical PC. In particular, the batch mode was not applied.

2.1. Assumptions

We count time using *MPI* for *MPI* solution and *OpenMP* for others. It only concerns the processing of loaded input data (the appropriate graph structure with matrix of costs) for all approaches.

Approach

Let's define an approach as:

- ***referential*** – greedy sequential algorithm for solving TSP,
- ***optimized*** – greedy parallel algorithm for solving TSP, based on sequential but created as a result of separation independent parts for calculations for the given number of threads.

Number of slots

Let's define **number of slots** as the *number of threads for OpenMP, Pthreads and std::threads as well as number of processes for MPI*.

Speedup

Let's define **speedup** as the *speedup of optimized or referential (depending on context) approach in comparison to other referential one:*

$$speedup = \frac{duration\ of\ referential\ approach}{duration\ of\ optimized\ approach}.$$

In the following part, we will consider the dependence between speedup, threads number and the number of cities visited.

2.2. Randomness of graph costs

Let's see the average duration of operation of *referential C* solver, comparing the graph loaded from file with the generated one.

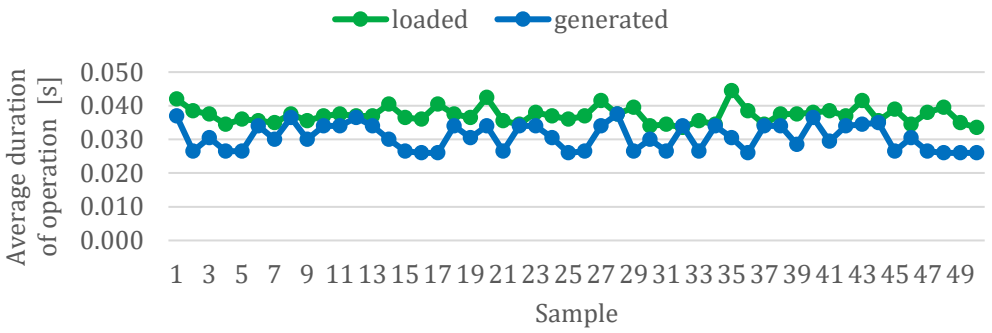

Fig. 3. Dependence of randomness on average duration of *referential* operation for *loaded* and *generated* approaches ($n = 10$).

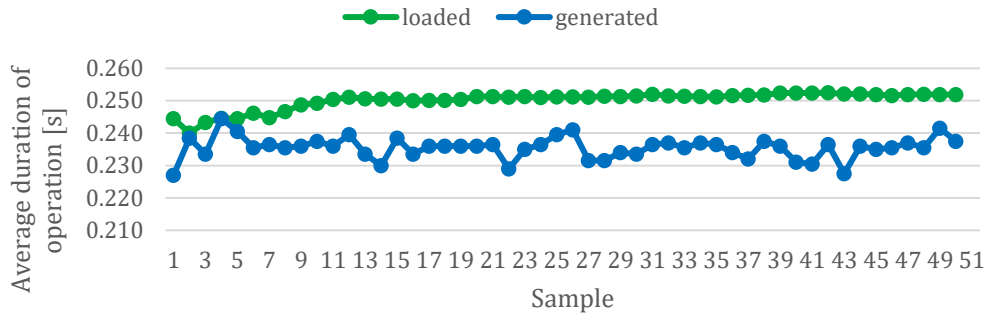

Fig. 4. Dependence of randomness on average duration of *referential* operation for *loaded* and *generated* approaches ($n = 11$).

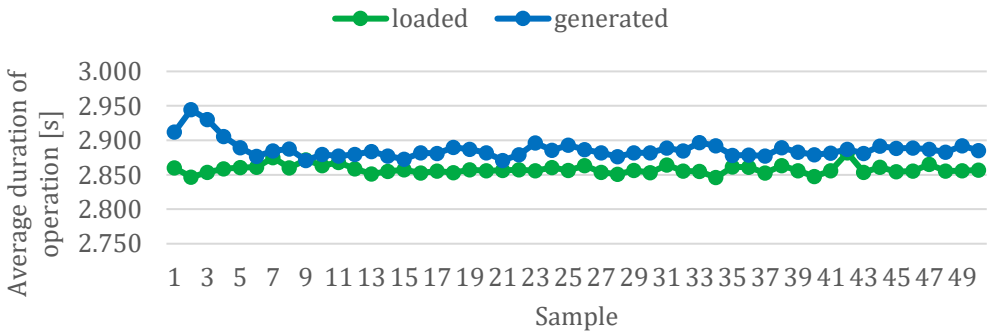

Fig. 5. Dependence of randomness on average duration of *referential* operation for *loaded* and *generated* approaches ($n = 12$).

Loading graph is a little bit more predictable than generating it, however both solutions are reasonable. For testing purposes, let's select the second approach which is simpler to manage.

2.3. Referential duration vs. number of cities

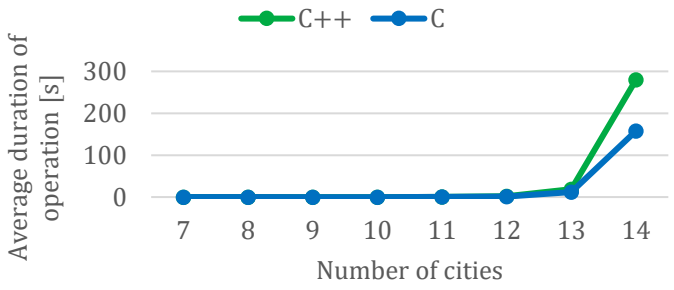

Fig. 6. Average time of operation vs. number of cities for *referential* approach ($average$: 10 $samples$).

Since the number of paths depends on cities like $n!$, according to intuition at some point ($n = 13$) the operation time increases dramatically (from a dozen to hundreds of seconds).

2.4. Speedup vs. slots

Let's say that we have 13 cities. It should be complex enough to have the influence of number of slots on speedup. Which is the most effective config to maximize the speedup?

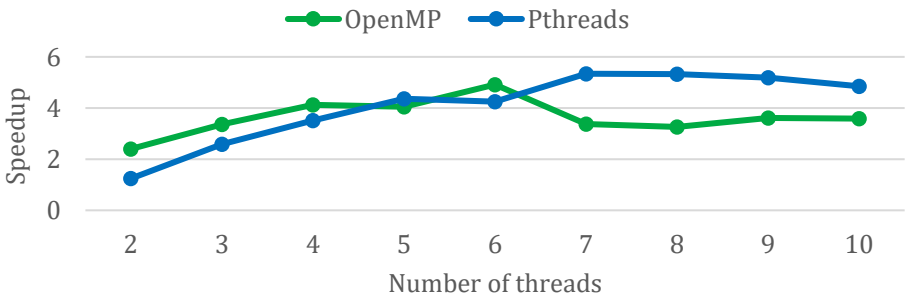

Fig. 7. Speedup vs. number of slots for *optimized* C approaches: OpenMP and Pthreads ($cities$: 13, avg: 10 $samples$).

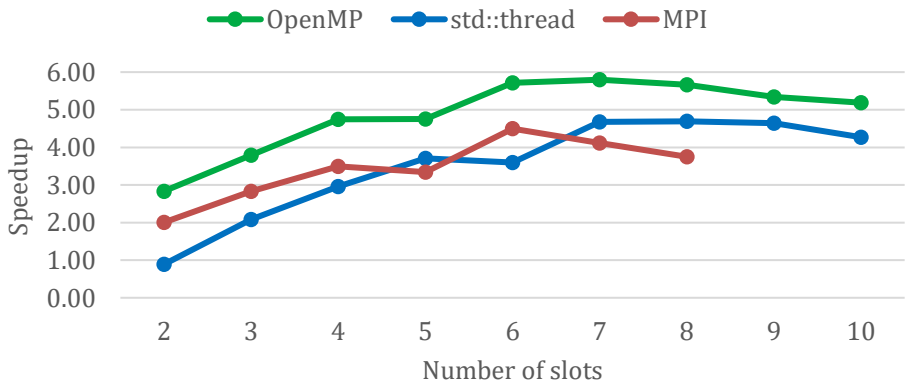

Fig. 7. Speedup vs. number of slots for *optimized* C approaches: OpenMP and Pthreads ($cities$: 13, avg: 10 $samples$).

MPI has been run with 8 virtual processors – that is why 2-8 slots were available for it only. The most best configuration for later purpose is that with 6 slots because of biggest speedups. It is interesting that on 4 physical cores we prefer 6 threads to 4.

2.5. Speedup vs. cities

The comparison reveals interesting results.

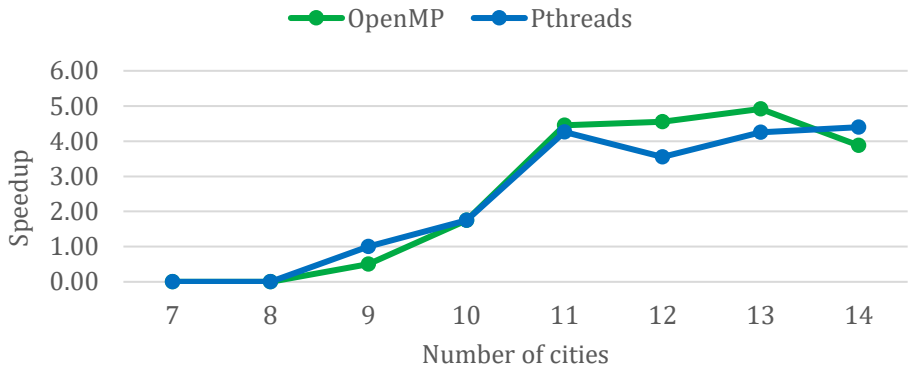

Fig. 8. Parallelization speedup vs. number of cities for *optimized* C approaches: OpenMP and Pthreads ($slots: 6, avg: 10\ samples$).

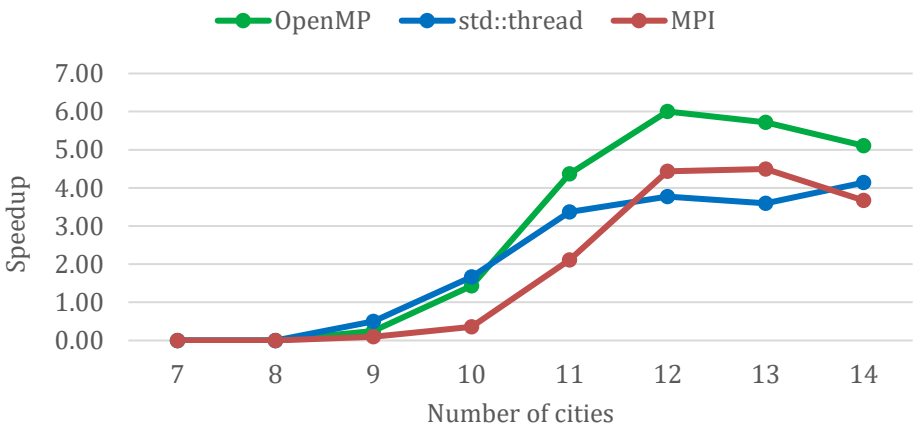

Fig. 9. Parallelization speedup vs. number of cities for *optimized* C++ approaches: OpenMP, std::threads and MPI ($slots: 6, avg: 10\ samples$).

Note that ***optimized* solutions are more efficient than *referential* only for number of cities** ≥ 10. That is because operating system puts a lot of effort to provide context switching and scheduling.

We can see that OpenMP approach seems to be the most optimized. However, MPI provides an easy way to distribute the solution in case of new computational cluster (let's say another computer) addition. Unfortunately, such a configuration was not tested.

2.6. Efficiency C vs. C++

It is said that a writing in C is more efficient than in C++. Is it really true? Let's find out.

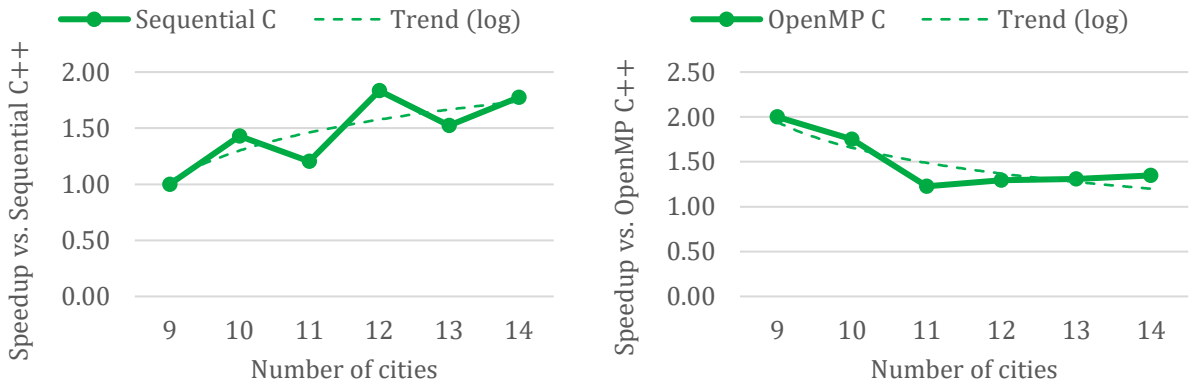

Fig. 10. Speedup comparison C vs. C++ (in units of sequential C++ average duration): Sequential and OpenMP approaches ($slots: 6, avg: 10\ samples$).

We could say that the C++ is slower than corresponding C solution, but not as much as it might seem. Modern C++ are optimized for speed well. We could say, combining above charts for 9-14 cities and 6 slots, **solution written in C is 1.47x faster than in C++ in average**.

3. Results

Randomness of graph cost matrix does not impact on processing time.

The **efficient way for using TSP is to use solvers run on 6 threads** (or processes).

At first glance, it should have seen that maximum speedup should theoretically oscillate around 4, because that was the number of processor cores allowed to process data. We noticed, that for a lot of situations it is true. But sometimes it was bigger – why?

We need to remember that in a real world we not only need to consider a time complexity but also space complexity (which we have not done). Apart from that all implementations of approaches are slightly different (e.g. because of deriving from different libraries). That's are main reasons why sometimes speedup was bigger. Even though real yield is smaller, it is **reasonable in a lot of situations where we can divide a problem into several smaller, independent pieces**. But talking about where it is possible is a big topic for other considerations.