INF3201 - Parallel Programming
Parallelizing the Traveling Salesman Problem with OpenMP

Narae Park and Jonas Fladset Hoel

October 8, 2020

1 Introduction

Travelling Salesman Problem (TSP) is a well-known NP-hard (Non-deterministic Polynomial-time hard) problem in computational complexity theory [3]. It is a problem of finding the shortest route that goes around all cities by exploring all possible routes according to the number of destinations given. As the number of destinations increases, the number of possible routes increases with factorial, so it takes a lot of time to explore. This paper explores whether the speed improvement and performance improvement can be achieved by applying parallelization using OpenMP in solving the TSP problem, and presents a design using such an approach. The solution will use a branch-and-bound approach.

2 Sequential Solution Analysis

Before an improved parallel solution can be presented and evaluated, a sequential (or serial) solution must be constructed. The sequential solution evaluates a tree of nodes (in TSP context, this would be cities) in a recursive fashion (meaning: the function which calculates the shortest route calls itself in the code). Pseudocode for the sequential solution is provided in listing 1. The tree mentioned earlier is not a literal data structure wherein each node/city has leaf nodes which are connected to them, but rather an emergent data structure due to how the solution works. The Solution recursively explores routes from the current node to all nodes that can be connected. The emergent route is stored for later reference, and the solution moves to evaluate the distance from the new neighbour node (now the current node) and all other nodes (except the previous one). Once the solution has found the shortest possible route that passes through all nodes, it returns up the function tree and an optimal route is produced.

```
Route* shortestRoute(Route* route){
    if (route->num_placed == num_nodes){
        route->length += #distance to return to the root
        return route;
    }
    i = route->num_placed;
    while (i < num_nodes)
        new_length = route->length + #distance to node #i
        # if new_length is worse than the global best, skip this loop iteration and continue
        new_route = swapped(route);
        new_route = shortestRoute(new_route); # call self, making this recursive
        if new_route->length < best_route->length {
            best_route = new_route;
            if best_route->length < global_best{
                global_best = best_route;
            }
        }
        else{
            delete(new_route);
        }
        i++
```

```
22      }
23      return best_route;
24  }
```

Listing 1: Sequential implementation pseudocode

In order to provide a baseline for any improvements by parallelizing TSP, the analysis of the sequential implementation which only uses a single processor will have to be done first. The results as presented by Valgrind can be seen in figures 1 and 2.

2.1 Sequential profiling

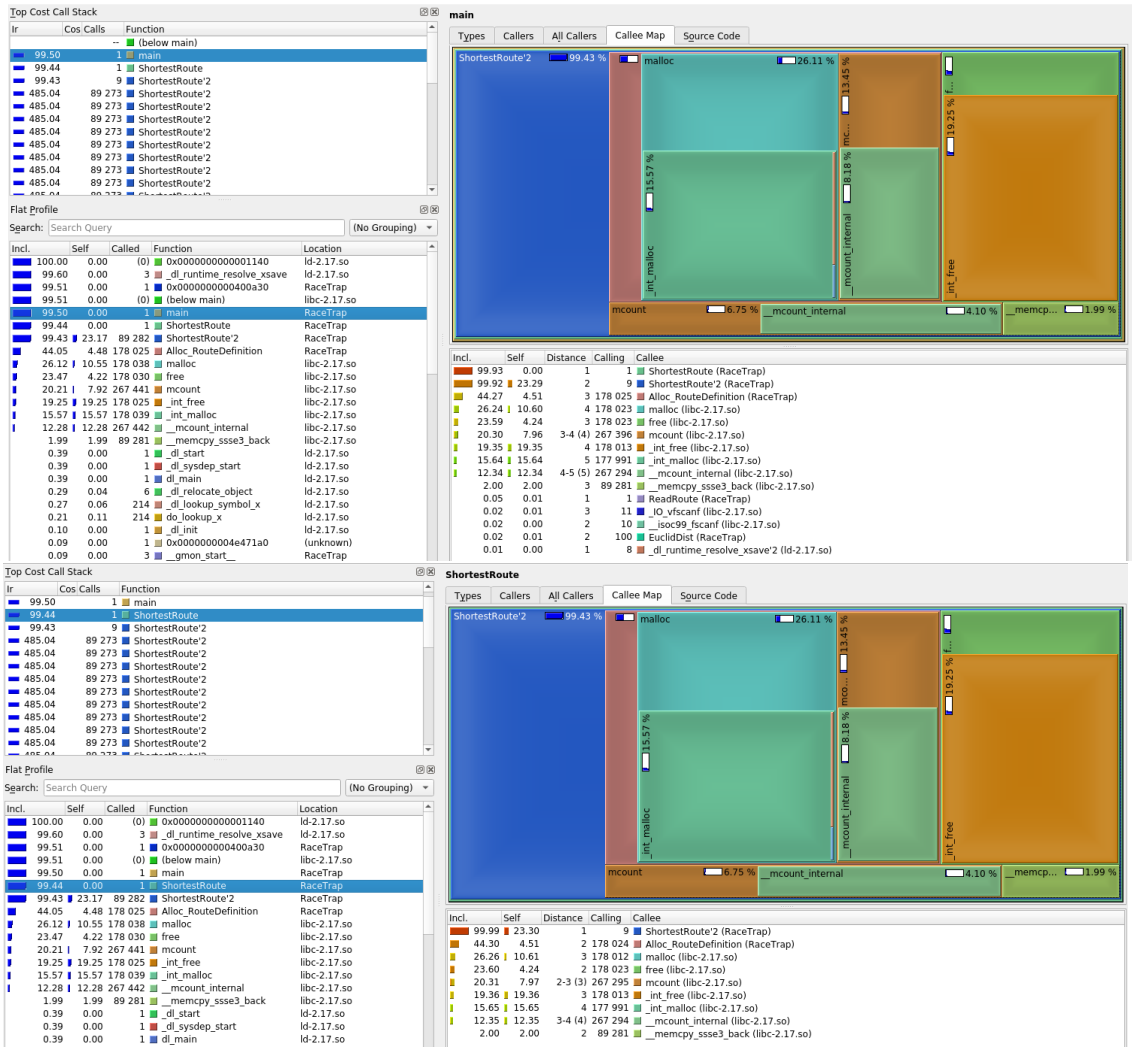

Figure 1: Profiling sequential solution pt. 1

Profiling the sequential version (see figure 1 and 2) shows that the ShortestRoute() function is being called recursively. Therefore, if this calls are executed in parallel through multiple threads, it would improve performance. Also, dynamic allocation of memory takes a lot of time(44.05%) as shown in the figure. If this can be controlled in the future, it would contribute to improve performance.

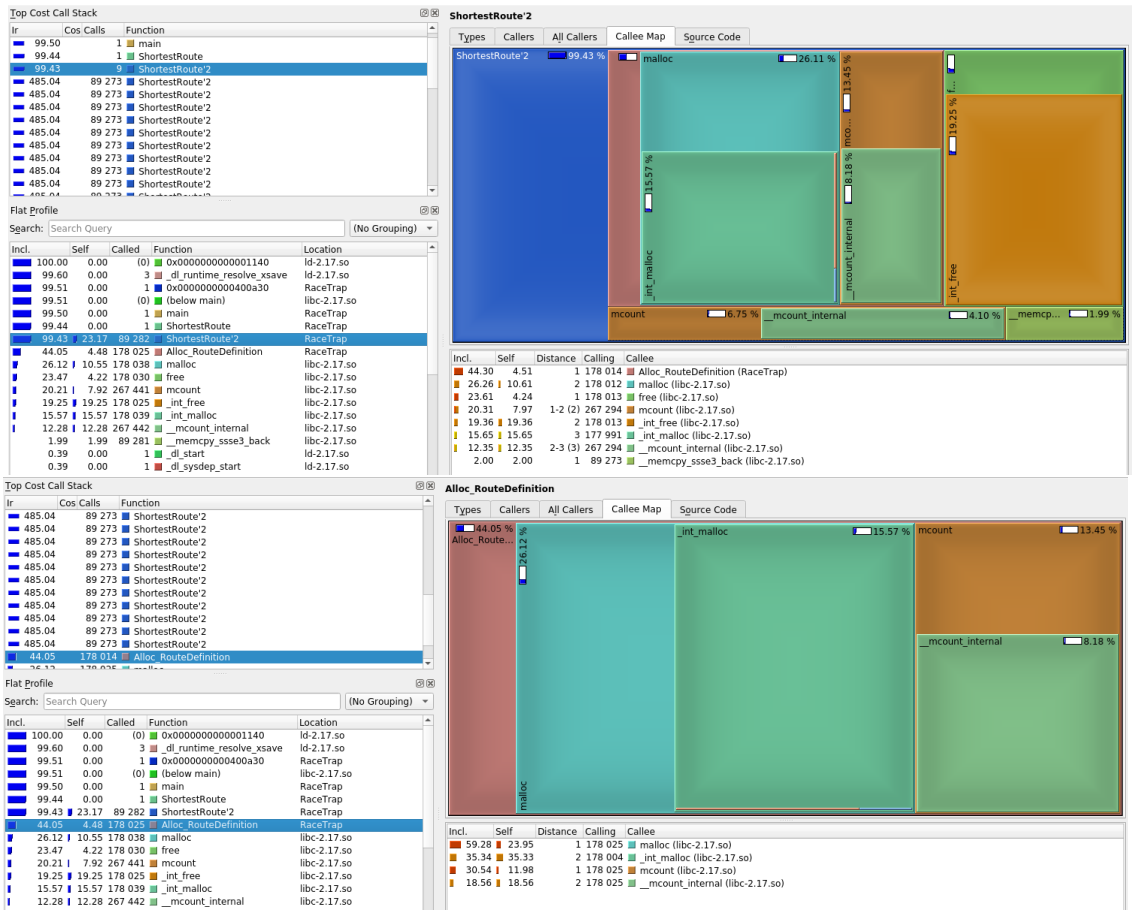

Figure 2: Profiling sequential solution pt. 2

2.2 Assumptions & Expectations

In order to construct an improved solution to the sequential solution, we must first assert a list of items that are assumed to be valid once the solution is parallellized, as well as expected outcome by parallellizing.

2.2.1 Assumptions

1. There will be a performance gain by parallellizing the serial solution.
2. There will be performance overhead due to the required passing of messages between threads as well as necessary critical sections to protect shared variables, such as the values for the best route and distance which can be changed by any thread at any point.

2.2.2 Expectations

1. Parallelization will shorten the execution time and improve performance.
2. If there are too many threads, there will be a performance penalty due to the overhead of thread handling.

3 Design

The design splits the sequential task of finding the shortest route from the current city to several threads using OpenMP, whenever the branch was divided. See figure 3

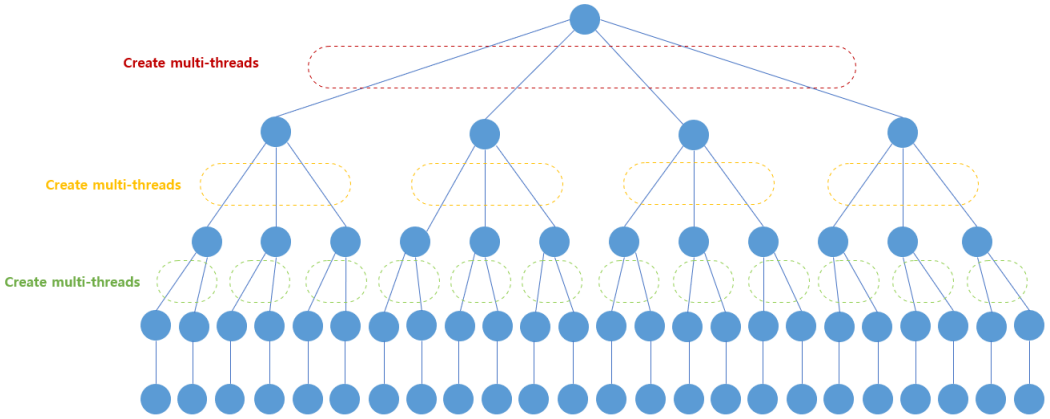

Figure 3: Design using parallel methods

3.1 Solution mixing serial and parallel methods

However, the method using a fully parallel approach consumed more time than a serial solution.(see table 1). As a result, the design was further refined to parallelize only up to the first few levels(see figure 4), the speed improved significantly.(see table 1)

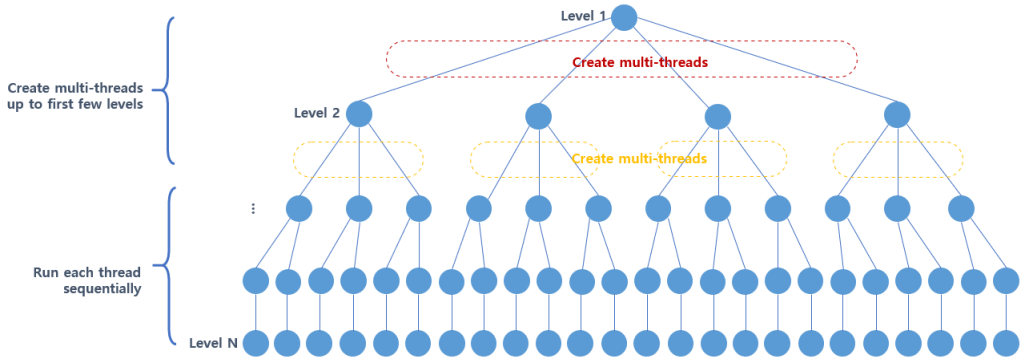

Figure 4: Design mixing sequential and parallel methods

Table 1: Execution time with serial, parallel and hybrid (first parallel and then serial) solutions

number of cities	Route length	sequential	parallel	parallel(upto level 1)	parallel(upto level 2)	parallel(upto level 3)
10	1762.289587	2ms	66ms	2ms	2ms	2ms
11	1778.446225	10ms	188ms	9ms	8ms	8ms
12	1821.468222	57ms	1s 18ms	15ms	27ms	17ms
13	1838.085899	204ms	3s 442ms	45ms	47ms	51ms
14	1929.435237	1s 309ms	22s 684ms	317ms	314ms	299ms
15	2029.921688	6s 541ms	2m 2s 906	1s 625ms	1s 623ms	1s 602ms

3.2 Solution applying lower bound

A different approach was tried[2][1] for greater performance improvement. It calculates the lower bound and reflected it in the pruning. The method is as follows.

1. Ideally, the shortest tour would include two edges with the smallest distance among the edges connected at one vertex. The distance sum of the two smallest edges in each vertex is assumed as the lower bound of the current state. (see figure 5 as an example)
2. Once one path is confirmed, a new lower bound is calculated by removing the randomly assumed edge and adding the distance of the determined path.
3. In this way, a tour length is calculated to the last leaf node and set it as the global upper bound (global best).
4. Based on this upper bound, if the lower bound in one state is greater than the upper bound, it is excluded from subsequent navigation.
5. If the length of the newly found route is less than the current global best, the global best is updated to a shorter one.

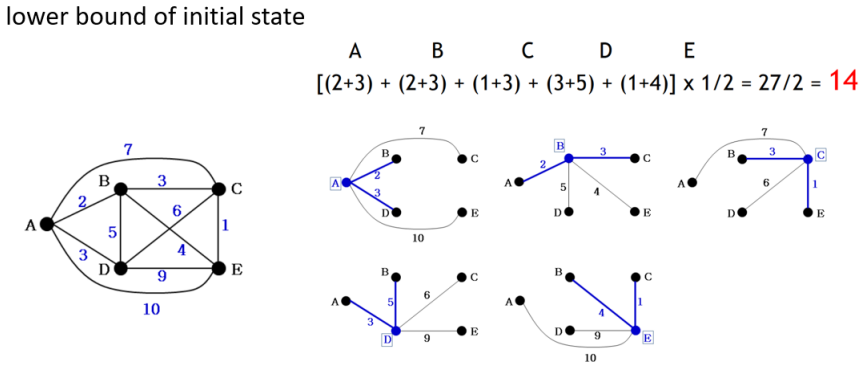

Figure 5: An example of calculating a lower bound in Branch and Bound algorithm [1]

This enables an update of lower the upper bound (which is the global best) when a tour path is completed. In other words, it allows pruning to occur earlier, reducing the number of possible routes that need to be explored. Applying a better algorithm enabled the solution to make better use of the advantages of the branch and bound method, which is reducing the number of routes through pruning, and to make faster speed and performance improvements.

4 Experiments

All experimental data provided are, unless otherwise stated, produced by (an) implementation(s) following the design(s) as described, compiled with GCC with optimisation level 2 on a Linux host (Ubuntu 18.04, WSL1). The host CPU was an AMD Ryzen 7 4800H 2.9GHz 8-core
As the number of cities increases, the results of measuring the execution time of serial, parallel 1, and parallel 2 are as follows.(see table 2[1] and figure 6)

We also checked the number of pruning and global best updates for the solutions (see table 3). It shows that the number of pruning is the lowest in the parallel solution using lower bounds. This means that pruning was done at earlier levels, reducing the number of routes to be explored.

[1]In the table, data that could not be measured since the execution time was too long were left blank.

Table 2: Execution time of searching TSP according to serial, parallel(hybrid), parallel(lower bound) solutions

number of cities	Route length	sequential	parallel(up to level 2)	parallel(lower bound)
10	1762.289587	2ms	2ms	7ms
11	1778.446225	10ms	8ms	10ms
12	1821.468222	57ms	27ms	13ms
13	1838.085899	204ms	47ms	28ms
14	1929.435237	1s 309ms	314ms	43ms
15	2029.921688	6s 541ms	1s 623ms	157ms
16	2032.317177	37s 927ms	7s 402ms	373ms
17	2100.362805	2m 11s 25ms	28s 328ms	397ms
18	2207.508442	9m 23s 29ms	2m 18s 292ms	569ms
19	2294.40145		18m 59s 173ms	1s 771ms
20	2302.638499			2s 816ms
21	2318.572285			6s 617ms
22	2319.472071			9s 939ms
23	2322.381384			29s 252ms
24	2342.199254			3m 42s 461ms
25	2381.382155			11m 24s 288ms
26	2406.833154			22m 40s 946ms

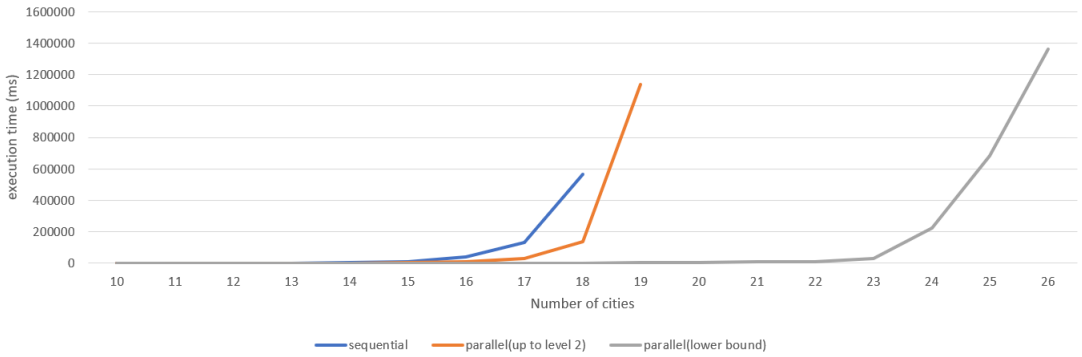

Figure 6: Execution time of searching TSP according to serial, parallel(hybrid), parallel(lower bound) solutions

Table 3: Number of prunings and global best updates of searching TSP according to serial, parallel(hybrid), parallel(lower bound) solutions

	sequential	parallel(up to level 2)	parallel(lower bound)
number of prunings	601604610	197554576	664136.6
number of global best updates	69	65.2	43.8

This is also evident in the following profiling results (see fig 7). Looking at the number of function calls in the three solutions, the number is significantly reduced in the order of sequential, parallel (hybrid), and parallel (lower bound). It shows that the amount of work is reduced through pruning by rapidly updating the upper bound through multi-threads running according to paralleling. Especially, when using the lower bound, it enables to update upper bound of a lower number, which maximizes the advantages of the branch and bound algorithm.

Looking at the profiling according to the three solutions, the callees list and map of the serial solution and the parallel (hybrid) solution is not very different. (see fig 8, 9) It might be because the parallel (hybrid) solution follows the serial method after the first few levels. In the parallel (lower bound) solution, it can be

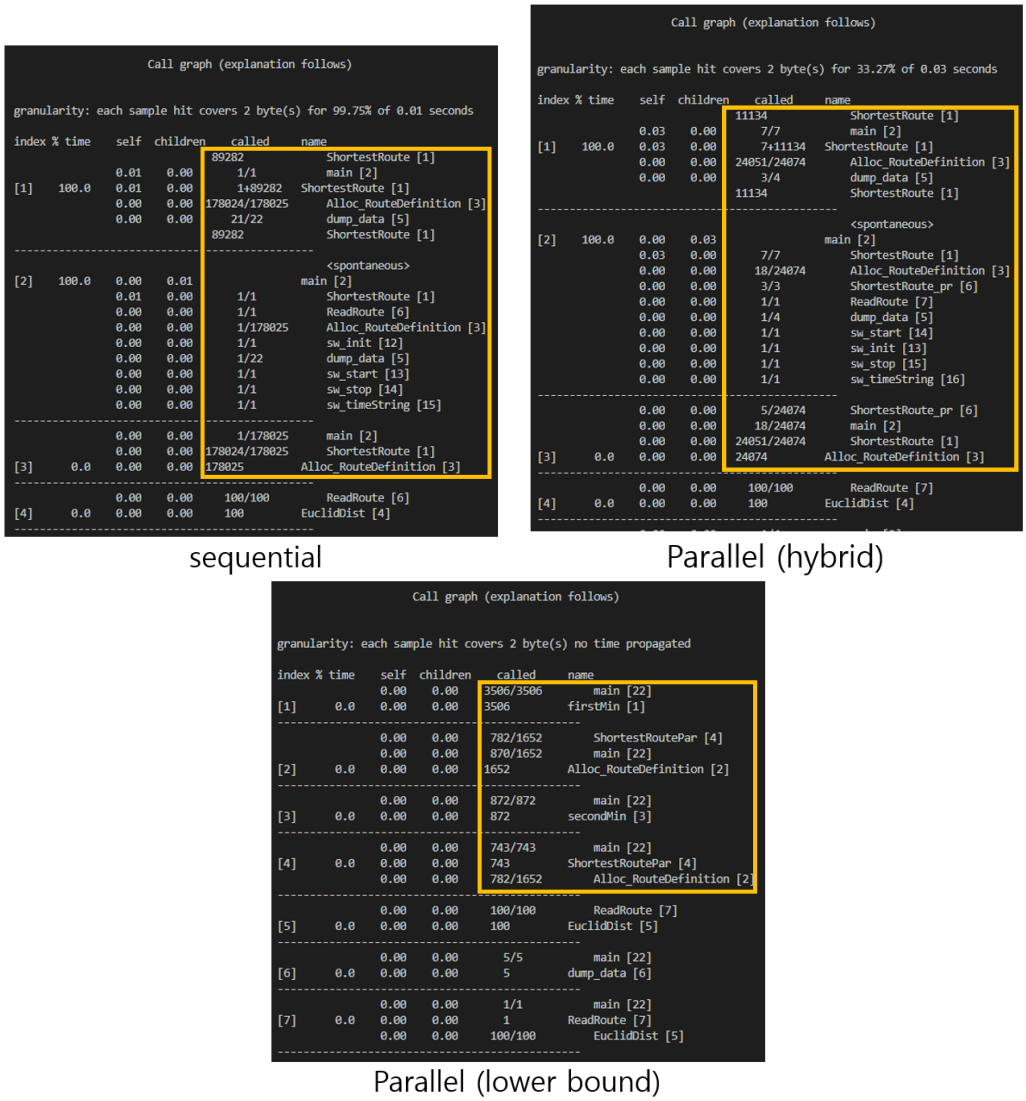

Figure 7: Profiling of serial, parallel(hybrid), parallel(lower bound) solutions (N=10, Gprof)

seen that the task or barrier function calls account for a greater proportion as the OpenMP(omp) is more actively used (see fig 10).

5 Discussion

While developing a parallel variant of the sequential solution, two parallel variants emerged with interesting properties. While the first iteration had less of an improvement in performance compared to the second, more improved variant, the results are nonetheless interesting enough to discuss and reflect on. As mentioned briefly in the Design section, the first solution explicitly uses parallel workload division, meaning even if the really is only one task to complete we incur the overhead of having to invoke a parallel environment with OpenMP. This could be the cause of the discrepancy seen in the experimental results of this first solution. The second method takes a different approach, also presented in the Design section. At a certain depth of recursively calling the calculating function, it swaps over to using a sequential approach as to not incur unneccesary overhead, race conditions with global variables and waiting around at critical sections.

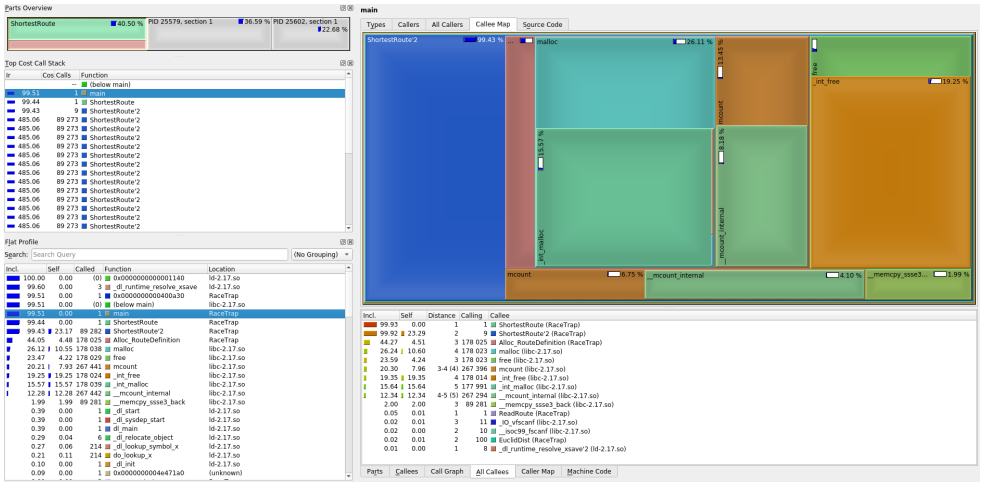

Figure 8: Profiling sequential solution (N=10, Valgrind)

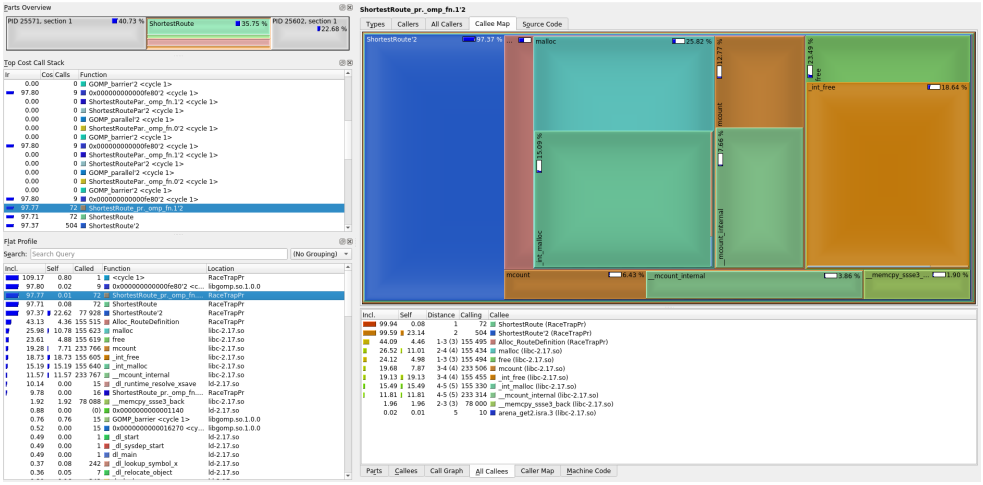

Figure 9: Profiling parallel(hybrid) solution (N=10, Valgrind)

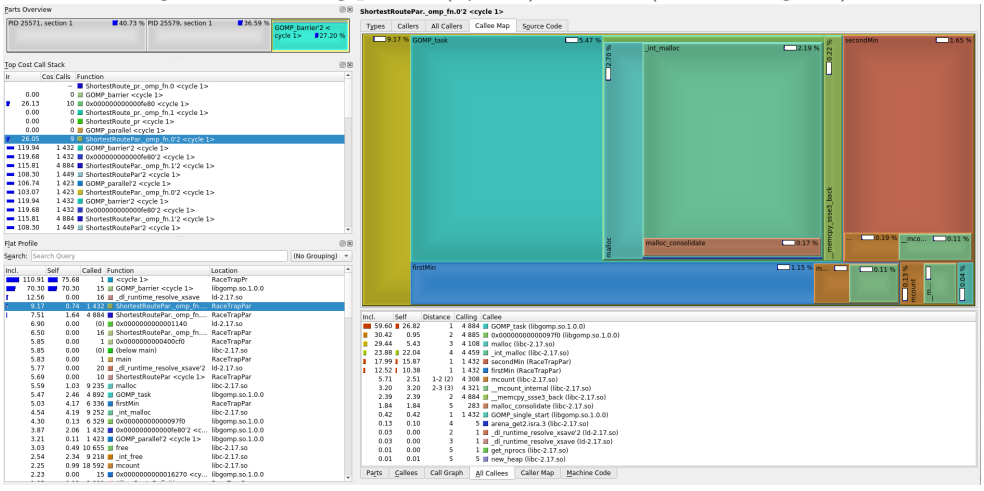

Figure 10: Profiling parallel(lower bound) solution (N=10, Valgrind)

In the process of implementing the TSP parallel solution, we would like to mention that it took more time to optimize parallelization than parallelization itself. Parallelization does not guarantee necessarily better performance itself. As we saw in the first parallel solution, if the cost of managing parallelized threads is greater than the benefits of parallelization, it results in lower performance.
We also found that the overhead of managing parallelized threads was greater than we thought. In the first parallel solution, it consumed more resources than the calculation work itself. This indicates that finding an optimal parallel solution requires a number of experiments and profilings to ensure that it is implemented in a way that achieves the best performance.

The branch and bound algorithm reduces the number of unnecessary searches through pruning. In order for pruning to take place at earlier levels, the lower upper bound should be updated more quickly. In the current solution, the upper bound can be updated based on the length only after a tour has been made. In other words, a thread must perform calculations up to the leaf node to update the global best criteria. If this process can be shortened, performance can be further improved.

Several optimizations have been applied to improve performance even slightly. Functions related to calculations were inlined, and the function calls were executed outside the for loop and assigned to variables unless it is necessary. Although we didn't do this this time, but we know that dynamic allocation of memory took a lot of time, as we saw in the sequential solution analysis. if we can optimize it in the future, it seems that we can achieve a greater performance improvement.

6 Summary

Parallelization has led to significant performance improvements when appropriate optimisation is involved. Too much thread execution has resulted in performance degradation that is worse than the serial method due to overhead. There was a greater performance improvement than when the solution was designed (with lower bound) in a way that maximized the effectiveness of the branch and bound algorithm applied to solve the TSP problem. Therefore, when designing a solution, it is necessary to consider the algorithms to be applied to solve the problem.

References

[1] J. Jang. Introduction to algorithms (9. search algorithm) (http://scsctrack.sogang.ac.kr/front/cmsboardview.do?currentpage=1searchfield=allsearchvalue=searchlowitem=allbbsconfigfk=2650siteid=scsctrackpkid=819253).

[2] A. Rai. Traveling salesman problem using branch and bound (https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/).

[3] wikipedia. Travelling salesman problem (https://en.wikipedia.org/wiki/travelling$_s alesman_p roblem$).