# Parallelization of the Vehicle Routing Problem Using OpenMP

Epshita Sarkar
CSE
Indian Institute of Information Technology Sricity,
Chittor, India
epshita.s21@iiits.in

Kavya Lolla
CSE
Indian Institute of Information Technology Sricity,
Chittor, India
kavya.l21@iiits.in

Sai Deepthi Padala
CSE
Indian Institute of Information Technology Sricity,
Chittor, India
saideepthi.p21@iiits.in

***Abstract***

The Capacitated Vehicle Routing Problem (**CVRP**) is a well-known and **NP-hard** combinatorial optimisation problem. Here we present our approach to solving CVRP by utilizing Open-MP parallelization techniques. The capacitated Vehicle Routing issue (CVRP) is a difficult optimization issue that requires determining the best routes for a fleet of vehicles to serve a specific set of clients while minimizing overall transportation costs.

As CVRP is NP-hard, finding a solution (ideal) is unfeasible, hence heuristic methods are used to find near-optimal results. However, heuristic techniques might be computationally expensive, especially for large VRP instances.

The paper we took as a reference utilizes local search and randomization to get better results. To further optimize execution speed, we use **OpenMP** to parallelize the method. Our approach allows us to explore a huge number of possible solutions while also providing a viable answer for large problem instances.

**Keywords:** Capacitated Vehicle Routing Problem , OpenMP, Parallelize , Ideal Solution

## I. Introduction

Vehicle Routing Problem (VRP) - VRP is a problem of computing the best (optimal) routes for a given set of vehicles to fulfill delivery for a group or specified group of customers based on some demand. Optimal routes serve the purpose of minimizing the overall transportation cost while minimizing the number of vehicles, minimum distance traveled, minimum travel time, or other objectives. Some of the applications of VRP include Supply Chain Management, Bus and Railway Route Optimization, Vehicle Optimization, etc.

One of the variants of VRP is Capacitated VRP (CVRP). CVRP states that m capacitated vehicles initially at depot locations are required to fulfill demands on customer nodes. The objective is to design the set of least-distant paths with known customer demands..

The Reference paper ParMDS employs a (MST) and a (DFS)-based technique.
The **Minimum Spanning Tree** (MST) is the collection of edges that is required to connect all the vertices in a graph, basically an undirected graph, with the minimum total edge weight.

**Depth-first search** (DFS), is a traversing algorithm , for tree and graphs, it returns the traversal order.
Basically , a MST-based solution provides a 2-approximate solution to the Travelling salesman problem. ParMDS uses a two-pronged approach: it exploits **"local search" and "randomization"** to improve the solution quality, and uses OpenMP parallelization to reduce execution time.

The Reference paper includes:

1. A MST and DFS combinatorial heuristic having both iterative local search and randomization to find an optimal solution to capacitated vehicle routing problem. The randomization is over the different traversal orders of a MST, each of which provides a valid candidate solution.

+

2.Present efficient shared-memory parallel of our proposed technique & significant increase in speed.

## II. Background and related work

### A. Problem Description

Given a fleet of vehicles from a depot, CVRP finds the minimum-cost round-trip delivery routes for the vehicles to a set of scattered customers while considering the constraints on the maximum load fits on each vehicle.

We are given a depot and $(n - 1)$ customers with coordinates $(xi, yi)$ for all $i = 0, \ldots, (n - 1)$. The depot is at node 0, there is a fleet of identical vehicles, each having a capacity $Q > 0$, which are initially located at the depot.

A customer $i$ has demand $di$ such that $0 < di \leq Q$; $i = 1, \ldots, (n - 1)$. The depot has zero demand. The cost for traveling between any pair of customers, or between customers and the depot, is taken as the Euclidean distance between them.

The aim of CVRP is to determine a set of routes having the optimal(minimum) total distance to serve all the customers such that each route starts and ends at the depot, each customer is served exactly once, and total customer demands on any route does not exceed the vehicle capacity $Q$.

**CVRP instance as a graph**.:

A CVRP instance can be modeled as a graph . Let, in an input instance, there be a depot and $(n - 1)$ customers denoted by nodes having ids $\{0, \cdots, (n - 1)\}$. The depot is represented by a node with id 0, whose demand equals 0. We define a complete graph $G = (V, E)$ such that $V = \{0, \ldots, (n - 1)\}$ is the vertex set. $V$ represents the depot and the customers. The edge-weight associated with an edge $(i, j) \in E$ is the Euclidean distance between vertices $i$ and $j$; the edge-weight is symmetric.

### B. Related Work

We have worked on paper **Effective Parallelization of the Vehicle Routing Problem**

They proposed a novel approach, ParMDS, for calculating an approximate yet reasonable solution to CVRP quickly, using parallelization. They also present an optimized sequential version of ParMDS, which they referenced as SeqMDS.

**ParMDS Overview :**

The algorithm takes input of graph representation, $G$, of a CVRP [adjacency matrix] , the customer demands, $D$[integer value] , and the vehicle capacity, $Q$[integer value], and outputs a valid CVRP solution[optimal and cost efficient route]. ParMDS proceeds in four steps. At starting itself it creates a minimum spanning tree (MST), $T$ , of this edge weighted complete graph ; this is also the first step in the well-known 2-approximation algorithm for Traveling Salesman Problem. In the second step, ParMDS performs depth-first traversal on the MST, starting from the depot, to define an ordering on the nodes . This step is preceded by randomly shuffling the adjacency lists of the MST which enables generating a different[unique] DFS order of the MST vertices in every iteration of the superloop . The third step partitions the ordered list of nodes ($\pi i$ ) into separate routes to comply with the capacity, $Q$. Repeat step step two and three for a fixed number, $\varrho$, of times to explore the space of different DFS traversals in order to get better routes. Finally, the fourth step refines the ordering of the nodes within each route of the solution using the well-known 2-Opt = and the nearest-neighbor heuristics for TSP ]. At the end, we obtain a collection of routes which is a valid CVRP solution, along with its cost.

**Our contribution :**

Now that we have ParMDS with openMp we future optimized it for prims algo , calculating the cost , and then the refining of routes. After utilizing the OpenMp primitives the algorithm is performing in a much better way with accurate results.

## III. Objective of the Work

This study aims to provide a new method for effectively addressing the difficult combinatorial optimization problem known as the capacitated vehicle routing problem (CVRP). This method, which aims to achieve shorter solution times while keeping reasonable accuracy, even on huge issue cases with hundreds of clients, combines local search and randomization to explore a vast space of potential solutions. Experimental comparisons between the proposed approach and state-of-the-art GPU implementations show a significant speedup and improved solution quality. The ultimate goal is to provide a functional and efficient CVRP system that can be used in real-world delivery, transportation and logistics situations.

## IV. Profiling

We used Gprof to profile the proposed method. The following are the results of profiling the sequential code (SeqMDS), parallelized code (ParMDS) and our improved parallel code(Opt_ParMDS).

Each sample counts as 0.01 seconds.

| % time | cumulative seconds | self seconds | calls | self ms/call | total ms/call | name |
|---|---|---|---|---|---|---|
| 48.48 | 15.79 | 15.79 | 100000 | 0.16 | 0.16 | ShortCircutTour(std::vector |
| 18.30 | 21.75 | 5.96 | 100000 | 0.06 | 0.06 | calCost(VRP const&, std::ve |
| 16.70 | 27.19 | 5.44 | 600100000 | 0.00 | 0.00 | void std::shuffle<__gnu_cx |
| 5.59 | 29.01 | 1.82 | 100000 | 0.02 | 0.02 | convertToVrpRoutes(VRP cons |
| 3.44 | 30.13 | 1.12 | | | | main |
| 3.19 | 31.17 | 1.04 | 131300000 | 0.00 | 0.00 | std::ctype<char>::do_widen |
| 1.32 | 31.60 | 0.43 | 1 | 430.00 | 666.37 | VRP::cal_graph_dist() |
| 0.83 | 31.87 | 0.27 | 95968 | 0.00 | 0.00 | void std::vector<Edge, std: |
| 0.80 | 32.13 | 0.26 | 517445 | 0.00 | 0.00 | std::pair<std::_Rb_tree_ite |
| 0.64 | 32.34 | 0.21 | | | | _init |
| 0.46 | 32.49 | 0.15 | 1 | 150.00 | 443.63 | PrimsAlgo(VRP const&, std:: |
| 0.09 | 32.52 | 0.03 | 605888 | 0.00 | 0.00 | void std::vector<int, std:: |
| 0.09 | 32.55 | 0.03 | 696 | 0.00 | 0.04 | tsp_2opt(VRP const&, std::v |
| 0.06 | 32.57 | 0.02 | 1000040 | 0.00 | 0.00 | void std::vector<std::vecto |
| 0.00 | 32.57 | 0.00 | 4530 | 0.00 | 0.00 | frame_dummy |
| 0.00 | 32.57 | 0.00 | 348 | 0.00 | 0.00 | tsp_approx(VRP const&, std: |
| 0.00 | 32.57 | 0.00 | 3 | 0.00 | 0.00 | std::vector<std::vector<int |
| 0.00 | 32.57 | 0.00 | 2 | 0.00 | 15.10 | postprocess_2OPT(VRP const& |
| 0.00 | 32.57 | 0.00 | 2 | 0.00 | 0.00 | std::vector<std::vector<Edg |
| 0.00 | 32.57 | 0.00 | 1 | 0.00 | 0.00 | verify_sol(VRP const&, std: |
| 0.00 | 32.57 | 0.00 | 1 | 0.00 | 0.00 | printOutput(VRP const&, std |
| 0.00 | 32.57 | 0.00 | 1 | 0.00 | 30.29 | postProcessIt(VRP const&, s |
| 0.00 | 32.57 | 0.00 | 1 | 0.00 | 0.10 | postprocess_tsp_approx(VRP |
| 0.00 | 32.57 | 0.00 | 1 | 0.00 | 0.00 | VRP::read(std::__cxx11::bas |
| 0.00 | 32.57 | 0.00 | 1 | 0.00 | 0.00 | VRP::~VRP() |
| 0.00 | 32.57 | 0.00 | 1 | 0.00 | 0.00 | std::_Bvector_base<std::all |
| 0.00 | 32.57 | 0.00 | 1 | 0.00 | 0.00 | std::vector<Point, std::all |
| 0.00 | 32.57 | 0.00 | 1 | 0.00 | 0.00 | std::vector<std::vector<int |
| 0.00 | 32.57 | 0.00 | 1 | 0.00 | 0.00 | std::vector<double, std::al |

**SeqMDS**

Each sample counts as 0.01 seconds.

| % time | cumulative seconds | self seconds | calls | self ms/call | total ms/call | name |
|---|---|---|---|---|---|---|
| 22.99 | 0.57 | 0.57 | 2011 | 283.55 | 286.95 | ShortCircutTour(std::vector<std::vector<Edge, std::allocat |
| 15.73 | 0.96 | 0.39 | | | | VRP::cal_graph_dist() |
| 13.31 | 1.29 | 0.33 | 11887309 | 0.03 | 0.04 | void std::shuffle<__gnu_cxx::__normal_iterator<Edge*, std: |
| 11.70 | 1.58 | 0.29 | | | | frame_dummy |
| 8.07 | 1.78 | 0.20 | | 0.01 | 4.14 | PrimsAlgo(VRP const&, std::vector<std::vector<Edge, std::a |
| 8.07 | 1.98 | 0.20 | 9284533 | 0.02 | 0.02 | unsigned long std::uniform_int_distribution<unsigned long> |
| 8.07 | 2.18 | 0.20 | 95968 | 2.08 | 2.08 | void std::vector<Edge, std::allocator<Edge> >::_M_realloc_ |
| 7.66 | 2.37 | 0.19 | 517445 | 0.37 | 0.37 | std::pair<std::_Rb_tree_iterator<std::pair<double, int> >, |
| 3.63 | 2.46 | 0.09 | 1809 | 49.77 | 57.03 | convertToVrpRoutes(VRP const&, std::vector<int, std::alloc |
| 0.40 | 2.47 | 0.01 | 33942 | 0.29 | 0.29 | void std::vector<int, std::allocator<int> >::_M_realloc_in |
| 0.40 | 2.48 | 0.01 | 16229 | 0.62 | 0.62 | void std::vector<std::vector<int, std::allocator<int> >, s |
| 0.00 | 2.48 | 0.00 | 1932 | 0.00 | 0.00 | calCost(VRP const&, std::vector<std::vector<int, std::allo |
| 0.00 | 2.48 | 0.00 | 578 | 0.00 | 0.00 | std::vector<int, std::allocator<int> >::operator=(std::vec |
| 0.00 | 2.48 | 0.00 | 12 | 0.00 | 0.00 | tsp_2opt(VRP const&, std::vector<int, std::allocator<int> |
| 0.00 | 2.48 | 0.00 | 7 | 0.00 | 0.00 | tsp_approx(VRP const&, std::vector<int, std::allocator<int |
| 0.00 | 2.48 | 0.00 | 4 | 0.00 | 0.00 | std::vector<std::vector<int, std::allocator<int> >, std::a |
| 0.00 | 2.48 | 0.00 | 2 | 0.00 | 12.78 | postProcessIt(VRP const&, std::vector<std::vector<int, std |
| 0.00 | 2.48 | 0.00 | 1 | 0.00 | 0.00 | _GLOBAL__sub_I_DEBUGCODE |
| 0.00 | 2.48 | 0.00 | 1 | 0.00 | 13.78 | postprocess_tsp_approx(VRP const&, std::vector<std::vector |
| 0.00 | 2.48 | 0.00 | 1 | 0.00 | 5.67 | get_total_cost_of_routes(VRP const&, std::vector<std::vect |
| 0.00 | 2.48 | 0.00 | 1 | 0.00 | 0.00 | std::vector<Point, std::allocator<Point> >::_M_default_app |
| 0.00 | 2.48 | 0.00 | 1 | 0.00 | 0.00 | std::vector<double, std::allocator<double> >::_M_default_a |

**ParMDS**

Each sample counts as 0.01 seconds.

| time | seconds | seconds | calls | ms/call | ms/call | name |
|---|---|---|---|---|---|---|
| 23.17 | 0.22 | 0.22 | 2334 | 94.29 | 94.29 | ShortCircutTour(std::vector<std::vector<Edge, std::allocator<Edge> >, std::allocator<st |
| 18.95 | 0.40 | 0.18 | 6982957 | 0.03 | 0.04 | void std::shuffle<__gnu_cxx::__normal_iterator<Edge*, std::vector<Edge, std::allocator< |
| 12.64 | 0.52 | 0.12 | | | | VRP::cal_graph_dist() |
| 11.58 | 0.63 | 0.11 | 5414328 | 0.02 | 0.02 | unsigned long std::uniform_int_distribution<unsigned long>::operator()(std::linear_cong |
| 9.48 | 0.72 | 0.09 | 240860 | 0.37 | 0.37 | std::pair<std::_Rb_tree_iterator<std::pair<double, int> >, bool> std::_Rb_tree<std::pai |
| 9.48 | 0.81 | 0.09 | | | | frame_dummy |
| 7.37 | 0.88 | 0.07 | | 0.01 | 2.26 | PrimsAlgo(VRP const&, std::vector<std::vector<Edge, std::allocator<Edge> >, std::alloca |
| 5.27 | 0.93 | 0.05 | 2169 | 23.06 | 23.06 | convertToVrpRoutes(VRP const&, std::vector<int, std::allocator<int> > const&) |
| 2.11 | 0.95 | 0.02 | 44994 | 0.44 | 0.44 | void std::vector<Edge, std::allocator<Edge> >::_M_realloc_insert<Edge>(__gnu_cxx::__nor |
| 0.00 | 0.95 | 0.00 | 38969 | 0.00 | 0.00 | void std::vector<int, std::allocator<int> >::_M_realloc_insert<int const&>(__gnu_cxx::_ |
| 0.00 | 0.95 | 0.00 | 18139 | 0.00 | 0.00 | void std::vector<std::vector<int, std::allocator<int> >, std::allocator<std::vector<int |
| 0.00 | 0.95 | 0.00 | 2216 | 0.00 | 0.00 | calCost(VRP const&, std::vector<std::vector<int, std::allocator<int> >, std::allocator< |
| 0.00 | 0.95 | 0.00 | 1133 | 0.00 | 0.00 | std::vector<int, std::allocator<int> >::operator=(std::vector<int, std::allocator<int> |
| 0.00 | 0.95 | 0.00 | 10 | 0.00 | 0.00 | std::vector<std::vector<int, std::allocator<int> >, std::allocator<std::vector<int, std |
| 0.00 | 0.95 | 0.00 | 2 | 0.00 | 8.78 | postprocessIt(VRP const&, std::vector<std::vector<int, std::allocator<int> >, std::allo |
| 0.00 | 0.95 | 0.00 | 2 | 0.00 | 0.00 | tsp_2opt(VRP const&, std::vector<int, std::allocator<int> >&, std::vector<int, std::all |
| 0.00 | 0.95 | 0.00 | 1 | 0.00 | 0.00 | _GLOBAL__sub_I_DEBUGCODE |
| 0.00 | 0.95 | 0.00 | 1 | 0.00 | 0.00 | tsp_approx(VRP const&, std::vector<int, std::allocator<int> >&, std::vector<int, std::a |
| 0.00 | 0.95 | 0.00 | 1 | 0.00 | 0.00 | postprocess_tsp_approx(VRP const&, std::vector<std::vector<int, std::allocator<int> >, |
| 0.00 | 0.95 | 0.00 | 1 | 0.00 | 1.05 | get_total_cost_of_routes(VRP const&, std::vector<std::vector<int, std::allocator<int> > |
| 0.00 | 0.95 | 0.00 | 1 | 0.00 | 0.00 | std::vector<Point, std::allocator<Point> >::_M_default_append(unsigned long) |
| 0.00 | 0.95 | 0.00 | 1 | 0.00 | 0.00 | std::vector<double, std::allocator<double> >::_M_default_append(unsigned long) |

**Opt_ParMDS**

## V. Algorithmic Details

**Algorithm 1**: Improved ParMDS

**Input**: G(V,E), demands D ($d_i$ for ith node ), Capacity Q

**Output**: R, collection of routes, $C_R$ the cost of R

1 T <- PRIMS_MST(G) //parallel

2 $C_R$ <- inf

3 for i <- 1 to p do //parallel

4 $T_i$ <- RANDOMIZE(T)

5 $\pi_i$ <- DFS($T_i$, Depot)

6 $R_i$ <- CONVERT_TO_ROUTES($\pi_i$,D,Q)

7 $C_{Ri}$ <- CALC_COST($R_i$) //parallel

8 if $C_{Ri}$<$C_R$ then

9 $C_R$ <- $C_{Ri}$

10 R' <- $R_i$

11 end

12 end

13 R<- REFINE_ROUTES(R')

14 return R,$C_R$

**Algorithm 2:** PRIMS_MST

**Input**: Graph instance (G)

**Output**: Adj list of MST (T)

1 T <- Φ

2 u <- depot (0,0)

3 T.push(u)

```
4 while !T.empty()        //parallelize
5                 top=T.top()
6                 T.pop()
7                 if(vis[top]) continue
8                 vis[top]=1
9                 for( it: adj[node])
10                        if(!vis[it])
11                                T.push(it)
```

**Algorithm 3:PostProcessIt**

**Input**: Routes

**Output**: Improved routes

```
1R1<-NEAREST_NEIGHBOUR_HEURISTIC(Route
s)
2 R2 <- 2-OPT_HEURISTIC(R1)
3 R3 <- 2-OPT_HEURISTIC(Routes)
4 ImprovedRoutes <- Φ
5 for i <- 1 to Routes.size() do //parallel
6                 Cost(R2[i]) //parallelize for loop
7                 Cost(R3[i]) //parallel for loop
8                 Min-ith-Route <- MIN(R2[i],R3[i])
9                 ImprovedRoutes.push(Min-ith-Route)
10 end
11 return ImprovedRoutes
```

**Algorithm 4**: GET_TOTAL_COST_OF_ROUTES – calculate the cost of all capacitated subroutes

**Input**: Array of capacitated routes (final_routes)

**Output**: total_cost

```
1 total_cost =0.0
2 for i <- 0 to final_routes.size() //parallel
3           curr_routes_cost= 0.0
4           for j <- 1 to final_routes[i].size()
5                       curr_route_cost+=calc_dist(
                        final_routes[i][j-1] ,final_routes[i][j])
6           curr_route_cost+=          cal_dist
(DEPOT,final_routes[i][final_routes[i].size()-1])
7           total_cost +=curr_route_cost
8 return total_cost
```

**STEPS OF PARMDS ALGORITHM**

1) **Construction of MST**: using Prim's algorithm and input of graph instance G, we construct the MST of G. The Prim's algorithm works on the principle of trying to form a tree of the input graph by iteratively searching for edges of unvisited with minimum cost and adding them to the MST list.

2) **DFS**: The adjacency list of the MST is shuffled to get a unique DFS permutation in every run of the ParMDS algorithm. The DFS is then used to find the capacitated routes.

3) **Generating the routes**: The routes are then generated by slicing the DFS permutation based on the capacity constraint Q. Individual routes are added to a set of routes.

4) **Refining the routes**: The generated routes are refined using two heuristic methods: nearest neighbors and 2-opt.

   a. **Nearest neighbors** involves initially assuming that all nodes in a route are unvisited, then after visiting a node, greedily continuing to pick the next nearest node, respecting the capacity constraint Q.

   b. **2 opt** is another intra-route refinement technique that **swaps the edges** between two pairs of nodes and adds the update if the change has led to an improvement in cost.

   c. **2 opt** is essentially used to remove any intersections in the routes.

## VI. Results

In various aspects, we compared and improved upon the following areas:

*1) Execution Time:*

ParMDS exhibited exceptional performance regarding execution time. Completing all instances in just 187 seconds, it demonstrated its efficiency across diverse CVRP instances.

In contrast, SeqMDS, Base1, and Base2 required significantly more time to complete the same instances. Base1 finished in 107 minutes, Base2 astonishingly took 2.5 days, while SeqMDS required only 28 minutes.

This substantial difference in execution time underscores the efficiency enhancements achieved by ParMDS, rendering it a compelling choice for real-world scenarios where timing is critical.

*2) Comparison with Baselines:*

Notable speedups were observed when comparing ParMDS with Base1 and Base2. On average, ParMDS outpaced SeqMDS, Base1, and Base2 by 9×, 36×, and 1189×, respectively.

These enhancements underscore ParMDS's heightened efficiency, crucial for surmounting the computational hurdles associated with solving CVRP on large-scale problem instances.

With the ability to deliver solutions in a fraction of the time required by cutting-edge methods, ParMDS emerges as a robust option for real-world applications where rapid decision-making is imperative.

*3) Solution Quality:*

In terms of solution quality, ParMDS consistently generated solutions at lower costs compared to the baselines. This was evident from the Gap metric, measuring the disparity between the algorithm's solutions and CVRPLIB's Best Known Solutions (BKS).

ParMDS achieved an average disparity of 11.85%, indicating its capacity to produce high-quality solutions closely approaching the optimal solutions available in the literature.

The enhancement in ParMDS solution quality stems from the synergistic interplay of randomization and parallelization, enabling efficient exploration of a broader solution space.

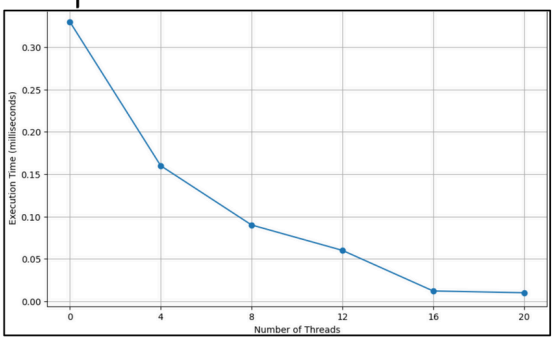
*4) Scalability:*

We also investigated the scalability of ParMDS using different thread sizes to assess its performance across diverse hardware configurations.

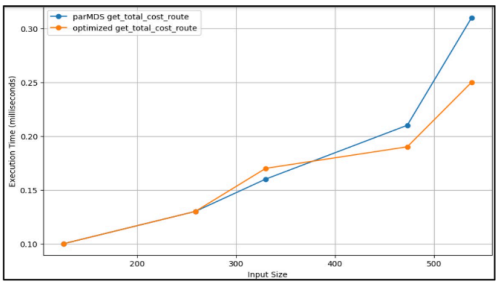
Overall, ParMDS demonstrated commendable scalability, with consistent speedups observed as the number of threads increased.

In instances where customers were densely packed and the capacity-to-customer demand ratio was high, marginal acceleration was noted due to the expanded solution space, underscoring the complexity of certain problem instances.

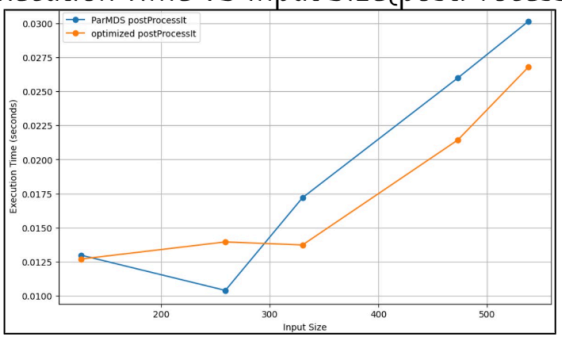
Number of threads VS Execution Time



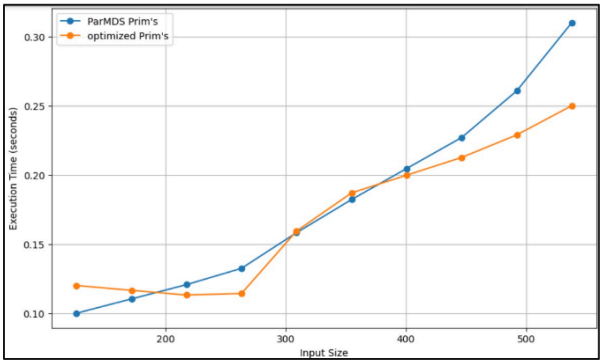

Execution Time VS Input Size(get_total_cost_route)




Execution Time VS Input Size(postProcessIt)




Execution Time vs Input Size(Prim's Alg)



## VII. Conclusion

We presented ParMDS, an effective shared-memory parallel technique, to solve the capacitated vehicle routing issue more quickly and with a good approximation. In addition to randomization, ParMDS uses a revolutionary MST- and DFS-based approach that allows it to swiftly explore a vast universe of potential solutions, even for big input instances. We evaluated ParMDS against the most advanced techniques utilizing GPU-parallel evolutionary algorithms.

Tests conducted on various input instances from CVRPLIB revealed that ParMDS achieves a better solution quality and is, on average, 36-1189× faster than the baselines; the average gap between ParMDS and the most well-known solution is 11.85%.

Regarding the algorithmic front, we intend to improve the ParMDS solution quality by using inter-route refining techniques and incorporating direction awareness into the existing scheme. Moreover, our empirical observations reveal that randomization has a crucial role in enhancing the quality of the solution in ParMDS. A theoretical investigation on how randomization affects the quality of the solution would be fascinating.

## VIII. REFERENCES


1. Rajesh Pandian M, Somesh Singh, Rupesh Nasre, and N.S. Narayanaswamy. 2023. Effective Parallelization of the Vehicle Routing Problem. In Genetic and Evolutionary Computation Conference (GECCO ’23), July 15–19, 2023, Lisbon, Portugal. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/ 3583131.3590458