Ashith Shetty
21BCS019

Smruti Patil
21BCS082

Pratik Pakhale
21BCS085

Shreyansh Tiwari
21BCS114

Parallel Algorithms for Expediting Vehicle Routes

Dr. Pramod Yelmewad (Project Supervisor)

Index

Problem Statement

The Problem:

Deliver goods from a depot to multiple customers using a fleet of limited-capacity vehicles.

The Goal:

Minimize total distance, serve every customer once, and never exceed vehicle capacity.

Why It's Challenging:

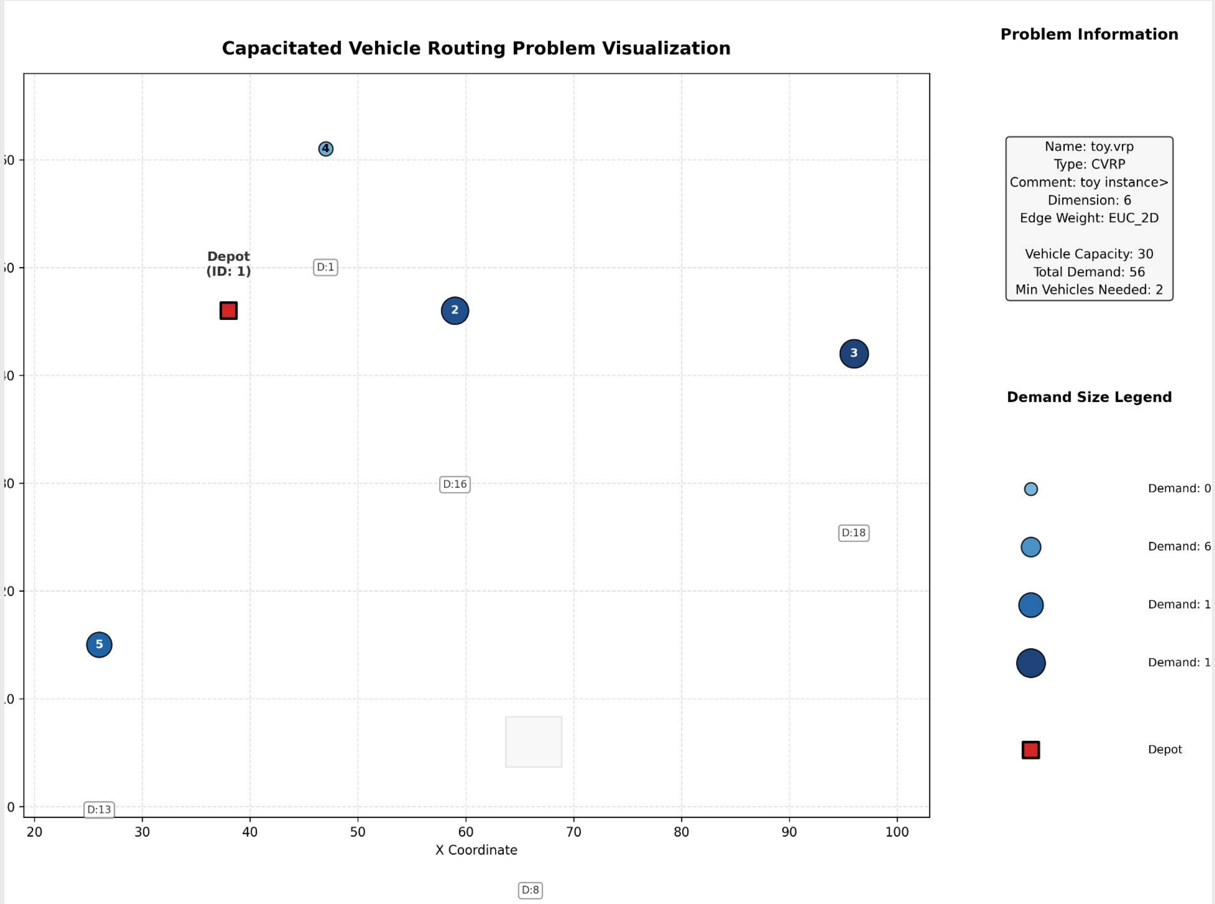
- CVRP is **NP-Hard**
- Problem size grows **exponentially**
- Needs **smart heuristics** for large-scale cases

Real-World Relevance

- Logistics (Amazon, DHL, Flipkart)
- Ride-sharing, Waste collection, Supply chains

Visualization

Visualizing a `toy.vrp` problem to check how the data is distributed.



Related Work

parMDS combines minimum spanning trees with randomized depth-first search traversals, achieving remarkable speed—solving 30,000-customer instances in seconds while maintaining competitive solution quality. This approach demonstrates that effectively parallelized simple heuristics can outperform complex metaheuristics on large problems.

GPU-Accelerated Hybrid GA leverages thousands of GPU cores for parallel fitness evaluation and neighborhood exploration in genetic algorithms. While effective for medium instances, its memory requirements limit scalability to very large problems.

FHCSolver introduces adaptive algorithm selection, dynamically choosing between HGS, FILO-HGS, and I-FILO based on instance characteristics—securing first place in the DIMACS Challenge by recognizing that different problems respond better to different solution approaches.

HLNS enhances the genetic search framework with ruin-and-recreate operations to escape local optima unreachable through standard local search. Its sophisticated fitness calculation considers both solution cost and population diversity.

POP-HGS addresses scalability by decomposing large CVRP instances into overlapping subproblems containing routes close to particular customers, effectively leveraging HGS's power for larger problems.

Several common themes emerge across these algorithms: hybridization of techniques (HLNS), problem decomposition (POP-HGS), parallel implementations (parMDS, Parallel LSH), and specialized memory structures (GPU implementations). Most algorithms face tradeoffs between solution quality and speed, with different approaches favoring different problem sizes—HGS variants excel on small instances, POP-HGS on medium ones, and parMDS dominates on very large instances with superior scaling characteristics.

ParMDS

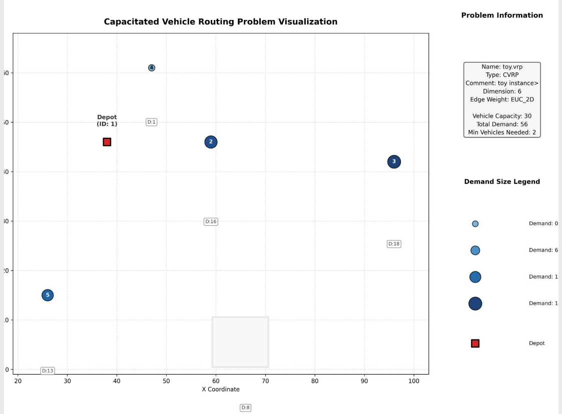
Algorithm 1: ParMDS: The proposed method

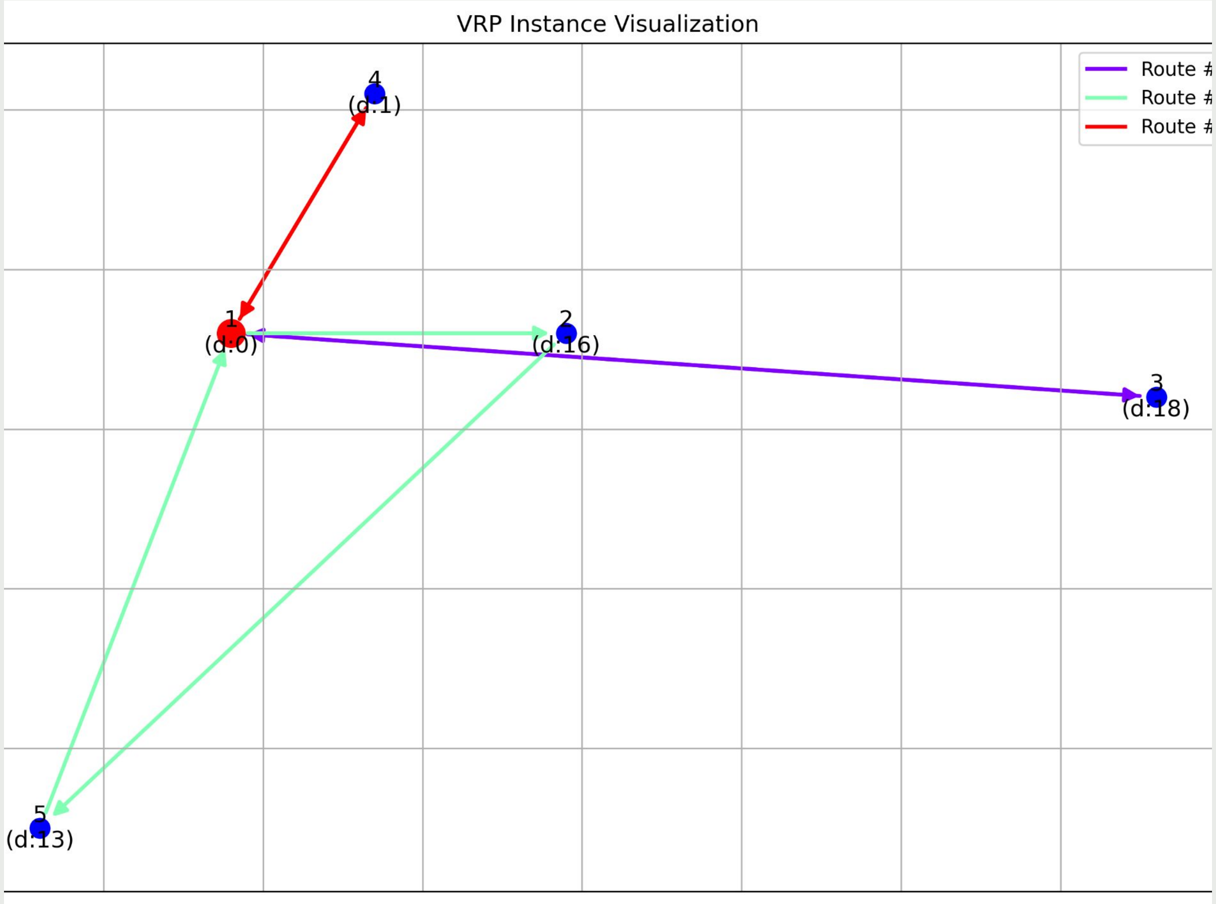
Input: $G = (V, E)$, Demands $D := \bigcup_{i=1}^{n} d_i$, Capacity Q

Output: R, a collection of routes as a valid CVRP solution
C_R, the cost of R

$T \leftarrow$ PRIMS_MST (G) /* Step 1 */
$C_R \leftarrow \infty$
for $i \leftarrow 1$ *to* ρ **do** /* Superloop */ /* Parallel */
  $T_i \leftarrow$ RANDOMIZE (T) /* Shuffle Adjacency List */
  $\pi_i \leftarrow$ DFS_VISIT (T_i, Depot) /* Step 2 */
  $R_i \leftarrow$ CONVERT_TO_ROUTES (π_i, Q, D) /* Step 3 */
  $C_{R_i} \leftarrow$ CALCULATE_COST (R_i) /* Parallel */
  if $C_{R_i} < C_R$ **then**
    $C_R \leftarrow C_{R_i}$ /* Current Min Cost */
    $R' \leftarrow R_i$ /* Current Min Cost Route */
  end
end
$R \leftarrow$ REFINE_ROUTES (R') /* Step 4 */
return R, C_R

ParMDS is a shared-memory parallel algorithm designed to solve large-scale instances of the Capacitated Vehicle Routing Problem (CVRP) efficiently while maintaining good solution quality. The method builds a Minimum Spanning Tree using Prim's algorithm and performs randomized DFS traversals to generate customer sequences, which are split into capacity-feasible routes. It repeats this for ρ iterations, selecting the best solution and applying intra-route optimization using Nearest Neighbor and 2-opt heuristics.





Visualization

Route #1: 1 -> 3 -> 1, Total Demand: 18
Route #2: 1 -> 2 -> 5 -> 1, Total Demand: 29
Route #3: 1 -> 4 -> 1, Total Demand: 1

Extending ParMDS

Algorithm 1: ParMDS: The proposed method

Input: $G = (V, E)$, Demands $D := \bigcup_{i=1}^{n} d_i$, Capacity Q
Output: R, a collection of routes as a valid CVRP solution
C_R, the cost of R

$T \leftarrow$ PRIMS_MST (G) /* Step 1 */
$C_R \leftarrow \infty$
for $i \leftarrow 1$ *to* ρ **do** /* Superloop */ /* Parallel */
  $T_i \leftarrow$ RANDOMIZE (T) /* Shuffle Adjacency List */
  $\pi_i \leftarrow$ DFS_VISIT $(T_i$, Depot$)$ /* Step 2 */
  $R_i \leftarrow$ CONVERT_TO_ROUTES (π_i, Q, D) /* Step 3 */
  $C_{R_i} \leftarrow$ CALCULATE_COST (R_i) /* Parallel */
  if $C_{R_i} < C_R$ **then**
    $C_R \leftarrow C_{R_i}$ /* Current Min Cost */
    $R' \leftarrow R_i$ /* Current Min Cost Route */
  end
end
$R \leftarrow$ REFINE_ROUTES (R') /* Step 4 */
return R, C_R

Outstanding Parallelization Potential: parMDS features a parallel superloop where each iteration is independent, making it ideal for GPU acceleration and parallel experimentation.

Exceptional Speed-Quality Balance: It offers a strong trade-off between speed and quality, solving large instances in seconds with only an 11.85% average gap from best-known solutions.

Elegant Algorithmic Simplicity: Its straightforward structure—MST construction followed by randomized DFS and local optimization—made it easy to extend without breaking its parallel design.

Clear Extension Opportunities: Its limited intra-route optimization and simple route-splitting left room for enhancements like inter-route reassignment and smarter partitioning methods.

Inter-Route Refinement

Algorithm 9 relocationMove()

Require: Collection of routes R, Distance matrix D

Ensure: Refined routes with reduced cost

```
for each pair of routes (R_1, R_2) in R do                          ▷ Parallel execution
    for each customer i ∈ R_1 do                                     ▷ Parallel execution
        cost_remove ← cost saving from removing i from R_1
        residual_capacity_R1 ← R_1.demand - demand(i)
        if residual_capacity_R1 ≥ 0 then
            for each position p in R_2 do
                residual_capacity_R2 ← R_2.demand + demand(i)
                if residual_capacity_R2 ≤ Capacity then
                    cost_insert ← cost of inserting i at position p in R_2
                    Δ cost ← cost_remove - cost_insert
                    if Δ cost < −1e − 6 then                        ▷ Move is beneficial
                        Move customer i from R_1 to position p in R_2
                        Update R_1.demand and R_2.demand
                        break                     ▷ Optional: accept first improving move
                    end if
                end if
            end for
        end if
    end for
end for
return R
```

Relocation moves involved systematically evaluating the cost impact of removing a customer from its current route and inserting it into a different route. For each customer originally in route R1, we calculated the cost saved by removing it, and for every feasible insertion into route R2, we computed the new cost and the net benefit. If the resulting total cost decreased beyond a minimal threshold (-1e-6), the move was executed.

Inter-Route Refinement

Algorithm 10 swapMove()

```
Require: Collection of routes R, Distance matrix D
Ensure: Refined routes with reduced cost
for each pair of routes (R_1, R_2) in R do                ▷ Parallel execution
    for each customer i ∈ R_1 do                          ▷ Parallel execution
        for each customer j ∈ R_2 do
            new_demand_R1 ← R_1.demand - demand(i) + demand(j)
            new_demand_R2 ← R_2.demand - demand(j) + demand(i)
            if new_demand_R1 ≤ Capacity and new_demand_R2 ≤ Capacity then
                cost_remove1 ← cost saving from removing i from R_1
                cost_insert1 ← cost of inserting j at i's position in R_1
                cost_remove2 ← cost saving from removing j from R_2
                cost_insert2 ← cost of inserting i at j's position in R_2
                Δ cost ← (cost_remove1 - cost_insert1) + (cost_remove2 - cost_insert2)
                if Δ cost < −1e − 6 then                  ▷ Swap is beneficial
                    Swap customers i and j between routes R_1 and R_2
                    Update R_1.demand and R_2.demand
                    break                                 ▷ Optional: accept first improving move
                end if
            end if
        end for
    end for
end for
return R
```

Swap moves considered exchanging customers between two distinct routes. For each candidate pair of customers i ∈ R1 and j ∈ R2, we checked feasibility and computed the combined cost effect of swapping them. If a net cost reduction was achieved, the swap was performed.
We parallelized the evaluation of these operations to improve efficiency. Multiple threads concurrently evaluated relocation and swap options, though the approach was brute-force in nature and lacked sophisticated filtering mechanisms.

This approach is not scalable.

Optimised Route Splitting

Algorithm 2: CONVERT_TO_ROUTES (π, Q, D)

Input: A permutation π, Capacity Q, Demands D
Output: Routes, a set of routes

```
OneRoute ← ϕ; Routes ← ϕ /* Initialize to empty */
ResidueCap ← Q /* Residual capacity */
for v ∈ π do
    if v = Depot then continue /* Skip Depot */
    if ResidueCap - D[v] ≥ 0 then /* Same Route */
        OneRoute.add(v)
        ResidueCap ← ResidueCap - D[v]
    else /* New Route */
        /* Add previous route to Routes set */
        Routes ← Routes ∪ OneRoute
        OneRoute ← ϕ
        OneRoute.add(v)
        ResidueCap ← Q - D[v]
    end
end
/* Add the last route to Routes set */
Routes = Routes ∪ OneRoute
return Routes
```

Greedy Route Splitting (Current Approach)

Used in the existing **ParMDS implementation** for solving CVRP
Performs **route splitting using a greedy strategy**
During DFS traversal:

- Customers are **sequentially added** to a route
- A new route is started when the next customer **exceeds capacity**

Efficient and simple, but **short-sighted**
Makes decisions based only on **local feasibility**, not global cost
Leads to **suboptimal route distributions** and inefficient clustering
Affects **overall travel cost**, especially in later segments

Proposed Solution – DP-Based Route Splitting in ParMDS

Introduced a **Dynamic Programming (DP)** strategy to improve route splitting
Finds the **optimal way to divide** the DFS traversal into feasible routes

Approach:

- Let `n` be the number of customers
- Define:
  - `C[i:j]`: Cost of serving customers *i* to *j* in one route
  - `dp[j]`: Minimum cost to serve customers from 1 to *j*
  - `split[j]`: Index where the last optimal route starts

$$dp[j] = \min_{i<j}(dp[i] + C[i:j])$$

⚙️ **Optimization & Complexity:**

- Precompute `C[i:j]` using prefix sums in **O(n²)**
- Fill `dp` table in **O(n²)**; parts can be **parallelized**
- Efficient for **mid-sized datasets (~1000 nodes)**

Execution Time & Min Cost – Greedy vs DP

Dataset Size	Method	Execution Time (sec)	Min Cost
n = 6	Greedy	0.004	277
	DP	0.0001	277
n = 101	Greedy	0.075	30,068
	DP	0.268	29,737
n = 1001	Greedy	0.477	80,748
	DP	22.174	80,356

ParMDS Greedy vs DP (n=101, Execution Time: 0.075 vs 0.268, Min Cost: 30068 vs 29737)

```
aasheeth@MathCo-PW0FSDDL:/mnt/c/Users/AshithDineshShetty/Code/parMDS$ ./parMDS-with-dp.exe ./inputs/X-n101-k25.vrp  -nthreads 20 -round 1 -dp 0
./inputs/X-n101-k25.vrp Cost 33170 30909 30068 Time(seconds) 0.00136531 0.0749959 0.075133 parLimit 20 VALID
Route #1: 32 95 31 24
Route #2: 73 53 33
Route #3: 93 75
Route #4: 30 85 11 79
Route #5: 50 19 23 21
Route #6: 100 61 97
Route #7: 91 52
Route #8: 83 71 51
Route #9: 89 98 99 62 81
Route #10: 27 8
Route #11: 17 34 64 3 40
Route #12: 44 58 12
Route #13: 18 4 66
Route #14: 84 9 92
Route #15: 68 54 86
Route #16: 1 70 76 90
Route #17: 55 16 69
Route #18: 74 13 77 88
Route #19: 39 25 28 14 63
Route #20: 10 65 42
Route #21: 78 6 49 2 7
Route #22: 37 36 29 43 45
Route #23: 59 67
Route #24: 60 57 72 87 48 26
Route #25: 38 47 82 96
Route #26: 56 94 80
Route #27: 46 20 35
Route #28: 41 22 15 5
Cost 30068
```

```
aasheeth@MathCo-PW0FSDDL:/mnt/c/Users/AshithDineshShetty/Code/parMDS$ ./parMDS-with-dp.exe ./inputs/X-n101-k25.vrp  -nthreads 20 -round 1 -dp 1
./inputs/X-n101-k25.vrp Cost 31746 30644 29737 Time(seconds) 0.000832341 0.26814 0.268223 parLimit 20 VALID
Route #1: 32 31 24
Route #2: 95 73 53 33
Route #3: 93 75
Route #4: 30 85 11 79
Route #5: 50 19 23
Route #6: 21 100 61 97
Route #7: 91 52
Route #8: 83 51
Route #9: 71 62 99 98 89
Route #10: 81 27 8
Route #11: 17 80
Route #12: 34 64 3 96 94
Route #13: 56 88 77 40
Route #14: 39 25 28 14 63
Route #15: 10 65 42
Route #16: 78 6 49 2 7
Route #17: 37 36 29 43 45
Route #18: 59 67
Route #19: 60 57 72 82
Route #20: 38 47 26 48 87 44
Route #21: 58 12
Route #22: 18 4
Route #23: 66 84 90
Route #24: 76 55 16 69
Route #25: 13 74 92
Route #26: 68 54 86
Route #27: 1 70 9
Route #28: 46 20 35
Route #29: 41 22 15 5
Cost 29737
```

Empirical results indicate that the DP-based method often yields **better initial solutions**, giving it a **head-start in metaheuristic search spaces**. However, the final solution quality post-optimization may converge with greedy-based results, with more **computational overhead**. This highlights a **trade-off** between early-stage solution quality and runtime efficiency.

Direction Aware Tour Construction

Original ParMDS Approach

What it does:

- Solves **CVRP** using fast, parallel heuristics.
- Core idea: generate *many randomized DFS tours* from an MST and pick the best.

Key Steps:

1. Build MST over all customers.
2. Randomize neighbor orders.
3. DFS traversal → one long tour.
4. Greedy split tour into valid routes (respecting capacity).
5. Keep the best tour across iterations.
6. Apply **TSP-style refinement** (2-Opt, Nearest Neighbor).

Strength:
Cheap randomization + parallelism = fast, decent solutions!

Optimized Approach

Fix 1: Parallel Correctness

- Original code had a **data race** — threads modified shared MST.
- I fixed this by giving **each thread its own copy** of the MST.

Fix 2: Direction-Aware Search

- Instead of randomizing neighbors, I tried a **geometry-based DFS**:
  - Sort neighbors by **angle** (e.g., clockwise from depot or current direction).
  - Attempt to build "smooth" tours with fewer turns.

Goal:

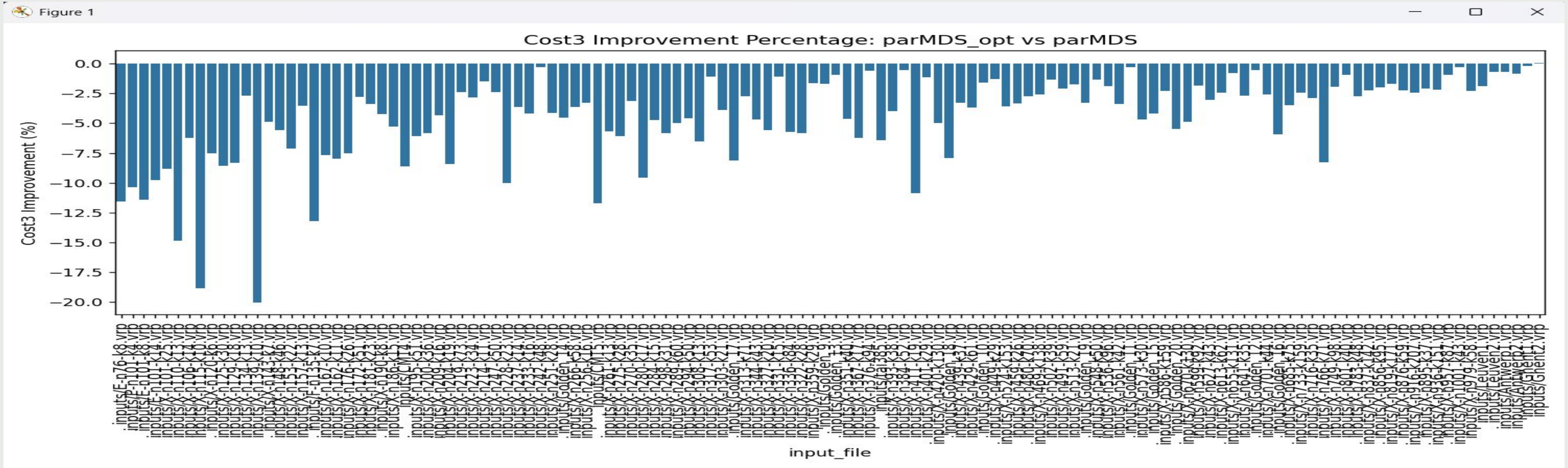
- Use spatial intuition to guide DFS.
- Reduce randomness → speed up convergence and improve route quality.

Direction Aware Tour Construction

What Actually Happened:

❌ **Performance Got Worse**

- **Costs increased by ~3.5%** on average.
- **Execution time increased by ~179%** (almost 3× slower!).



Direction Aware Tour Construction

Why did it fail?

Reason 1: It Was Slower

- Direction-aware DFS required:
  - Angle calculation (`atan2`) for every edge
  - Sorting neighbors (expensive!)
- **Random shuffling is cheaper** and faster in large iterations.

Reason 2: It Didn't Help CVRP

- CVRP ≠ TSP. It's not just about nice-looking paths.
- A "smooth" tour might split poorly due to **capacity constraints**.
- Random tours sometimes align better with splitting logic.

Reason 3: Less Exploration

- Direction-guided DFS restricts the search space.
- Misses "lucky" random tours that split beautifully.
- Refinement (like 2-Opt) works better when there are flaws to fix.

Randomized DFS and Improved Post Processing

Algorithm 8 MST Enhancements

```
Input: MST T from graph G = (V, E), vehicle capacity Q, customer demands D, number of iterations rho
Output: Best CVRP solution R* with cost CR
for each iteration i from 1 to rho in parallel do ▷ NEW: Fully parallelized superloop using OpenMP
    T_i ← Shuffle_Adjacency_List(T, seed = i)        ▷ NEW: Randomized DFS via shuffling adjacency lists
    pi_i ← DFS_Traversal(T_i, starting from depot)
    R_i ← Split_Tour_By_Capacity(pi_i, Q, D)
    R_A ← TSP_Approximation(R_i)            ▷ NEW: Heuristic TSP-style postprocessing
    R_B ← TwoOpt_Improvement(R_i)           ▷ OLD: 2-opt already in original
    R_i* ← Select_Better_Routes(R_A, R_B)   ▷ NEW: Layered dual-route comparison
    CR_i ← Evaluate_Cost(R_i*)
    if CR_i ¡ CR then
        CR ← CR_i
        R* ← R_i*
    end if
end for
return R*, CR
```

Randomized DFS on the MST

The original algorithm generated a single tour by performing a depth-first search (DFS) on the minimum spanning tree (MST) built from the complete distance graph. In the updated version, instead of relying on just one DFS ordering, the algorithm now repeatedly shuffles the adjacency lists of the MST using varying random seeds.

Layered and Parallel Post-processing Heuristics

Rather than applying a single TSP improvement method after the tour is obtained, the updated approach incorporates two distinct post processing schemes. One uses a TSP approximation strategy, while the other uses a classic 2-opt local improvement. By running both post processing techniques concurrently and then comparing their respective route costs to select the best result, the algorithm is able to refine candidate solutions more robustly.

Code Optimizations

01

Safety

In the original code, candidate solutions were generated in a loop that updated the current best solution without adequate protection. In the updated version, a critical section (using OpenMP's "**#pragma omp critical**") we introduced so that only one thread at a time can update the shared best cost (minCost) and best route (minRoute).

02

Elimination of Redundant Computations

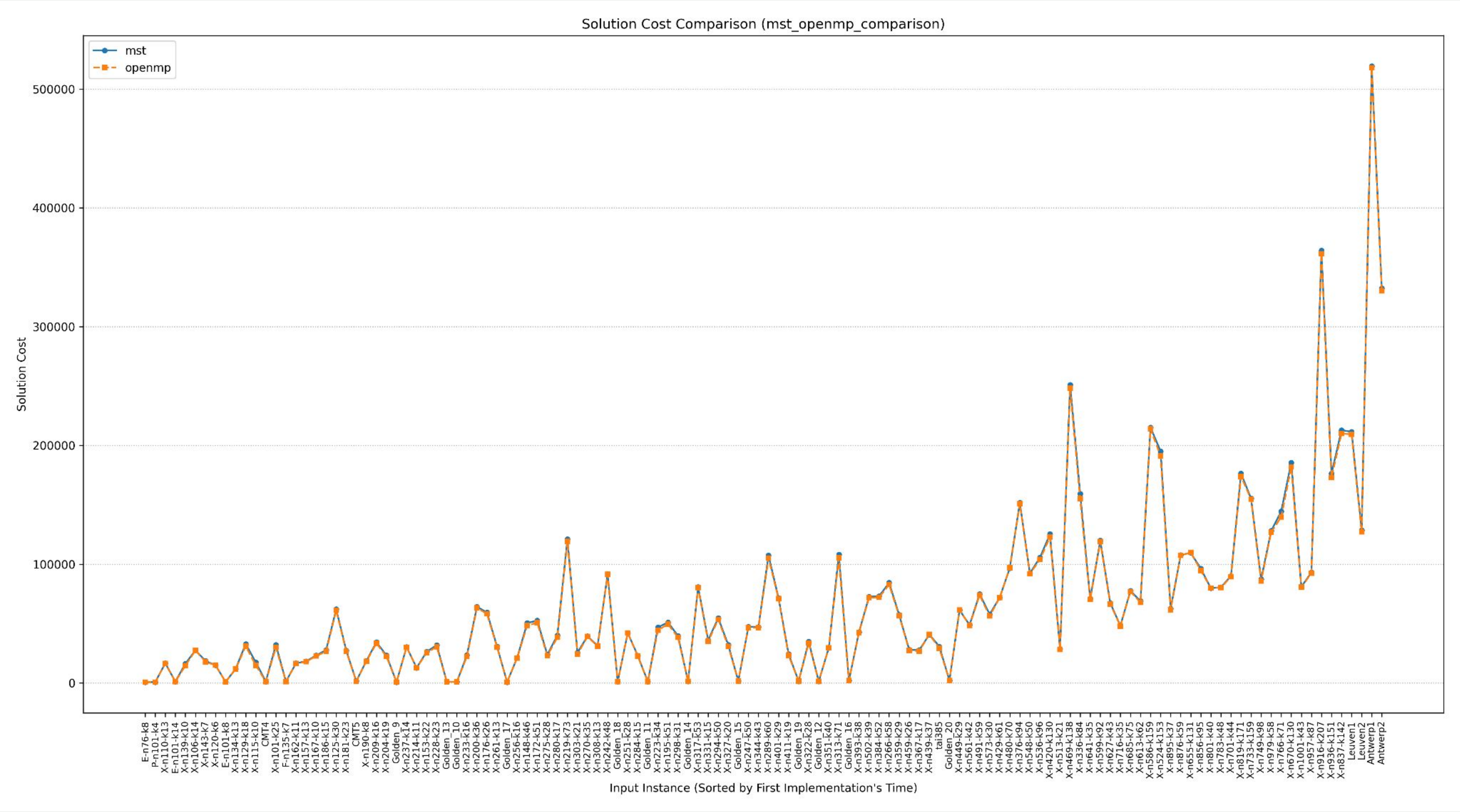
Some of the duplicated or repeated calculations (for example, re-calculating route costs in multiple loops) have been streamlined.
When the cost of each candidate set of routes is computed (the sum of the distances traversed in each route), the updated code uses **OpenMP's reduction clause**

03

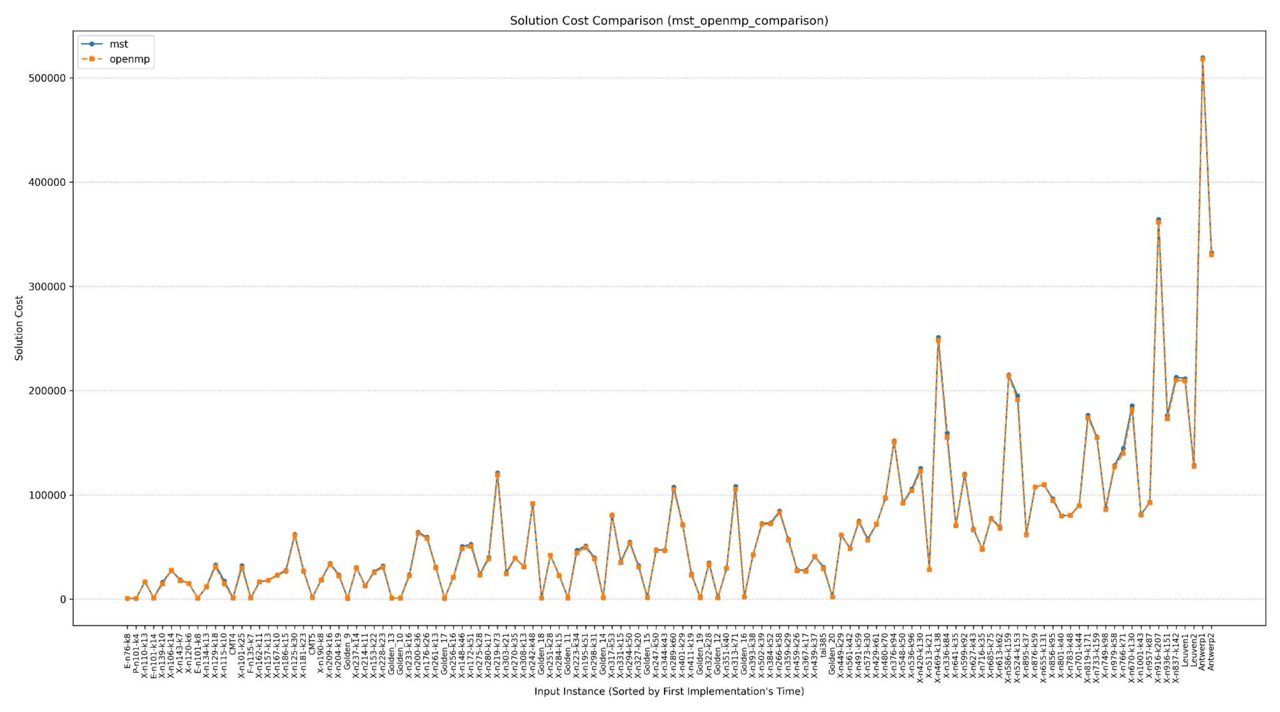
Better Utilization of Multi-Core Systems

By reorganizing the iterative DFS improvements into a parallel loop with **dynamic scheduling**, the updated code now is able to exploit the full potential of available CPU cores. We reduced Synchronization Overhead and did thread-specific randomization to explore the solution space more efficiently.

Results (Cost)

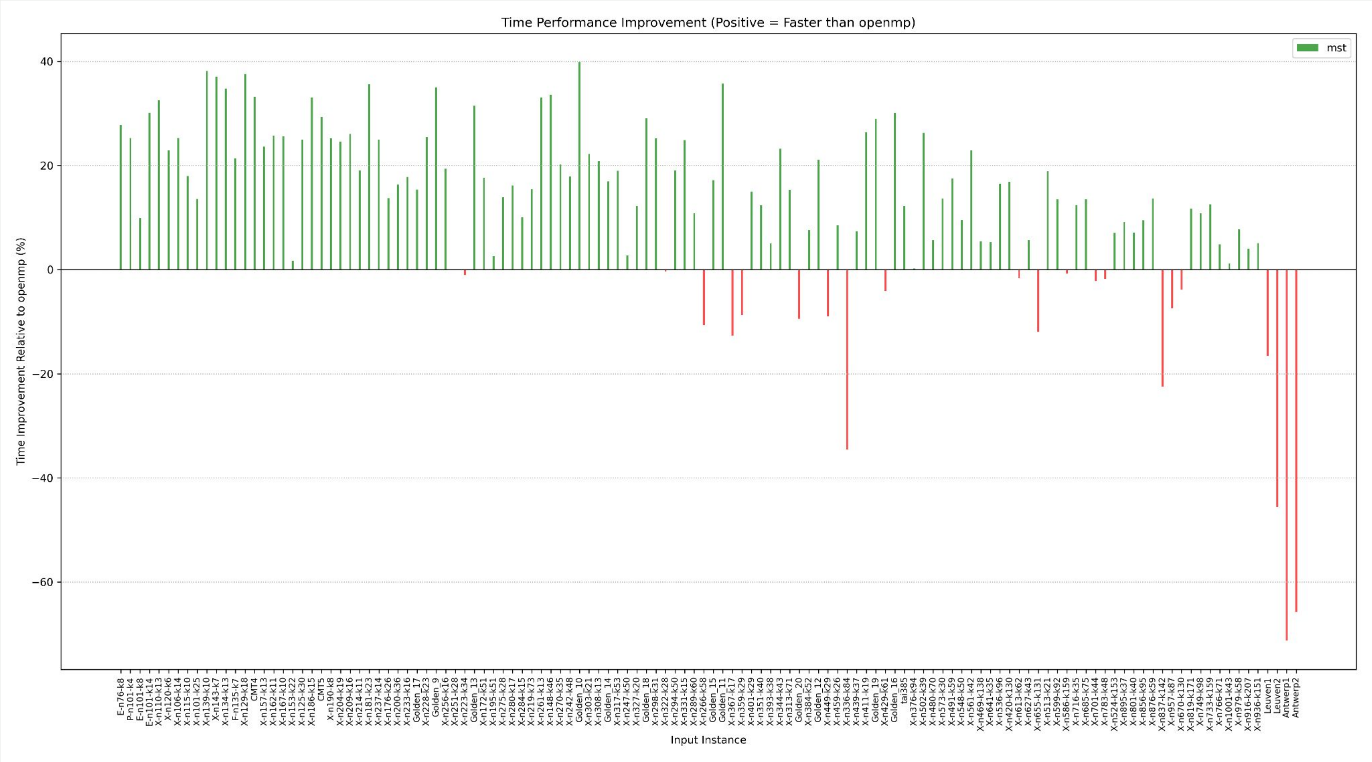


Results (Cost)



Mean loss: 1.91%
Median loss: 1.42%
Instances with worse cost: 112
Instances with better cost: 12

Results (Time)



Mean improvement: 12.55%
Median improvement: 15.12%
Instances faster: 103
Instances slower: 21

Porting to CUDA

Improvements

– Uses a GPU kernel to compute the pairwise distances between nodes in parallel.

– Applies a tiling strategy with shared memory to load coordinate data in 32 × 32 blocks.

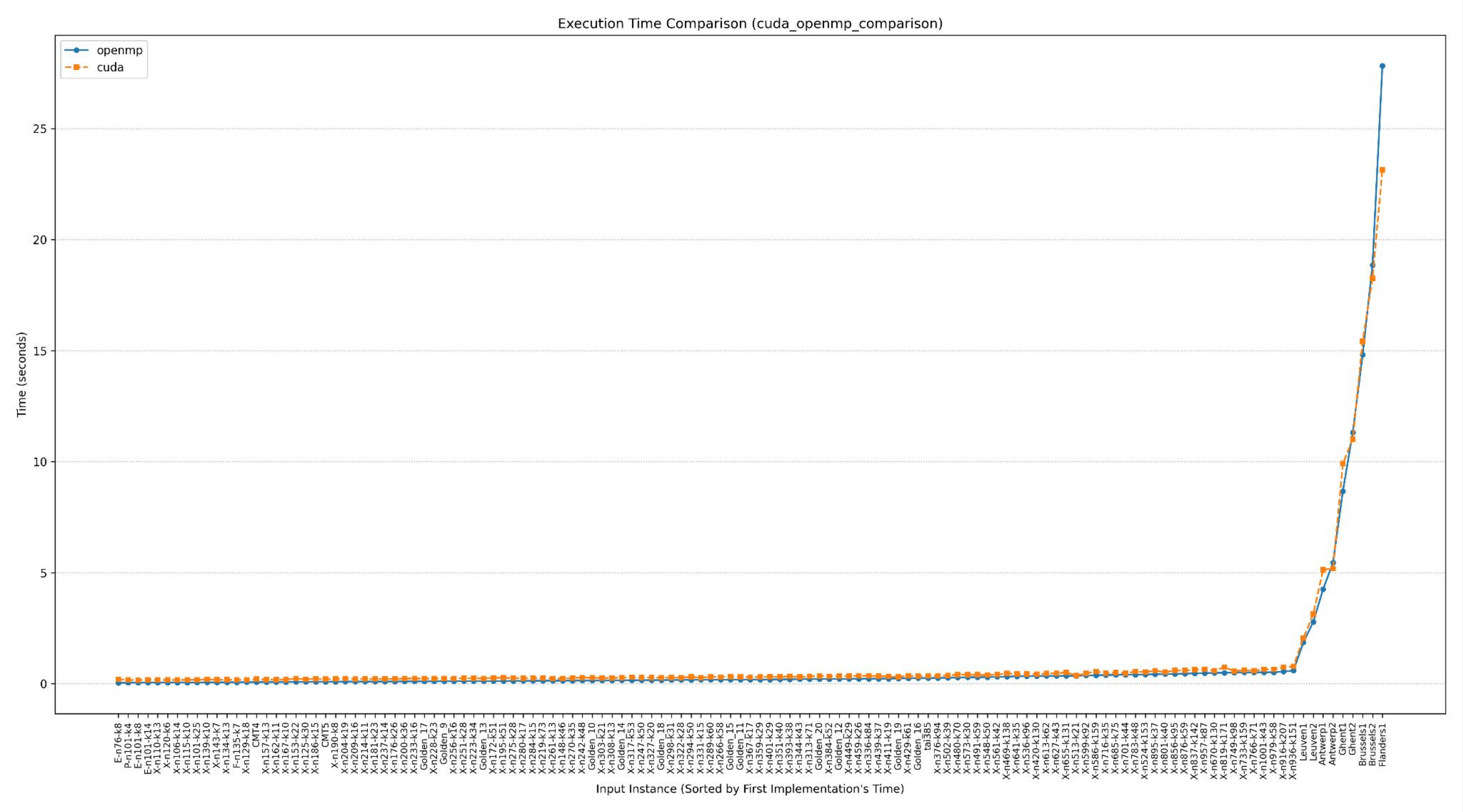
– After the CUDA kernel computes the distance matrix (stored as a flat, triangular array), the distances are transferred back to the host where the complete graph (an adjacency list) is reconstructed.

– The remainder of the algorithm follows a similar structure as in the CPU version: constructing a minimum spanning tree (using Prim's algorithm), generating a tour via depth-first search (DFS), and then post processing that tour (using TSP approximations and two-opt improvements).

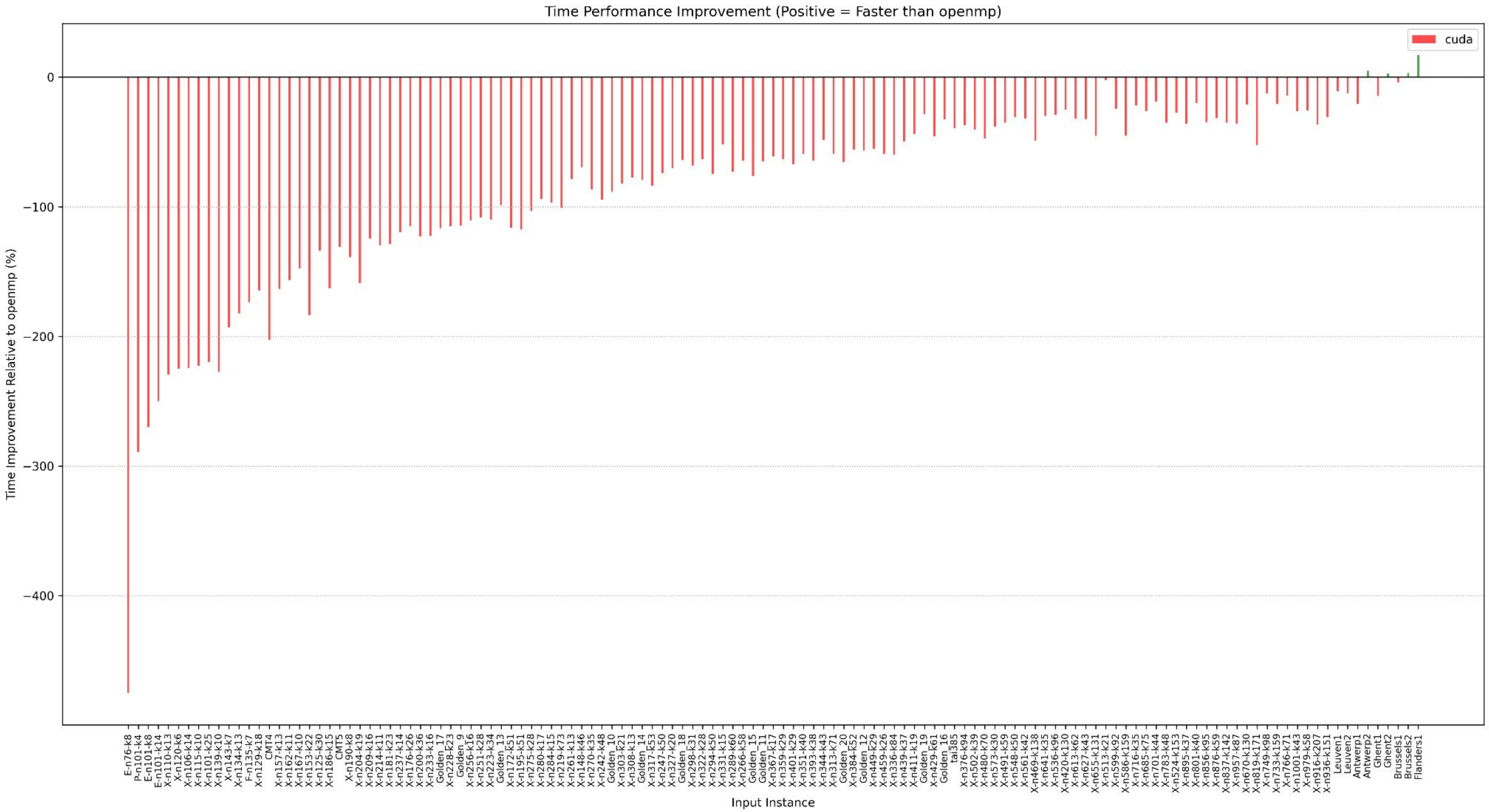
Bottlenecks

The device we currently use to run parMDS is GTX 1650 with 4gb VRAM. GTX is much slower for precision (double) variables hence using better resources we can vastly improve the results and clearly see the tradeoff of using GPUs for larger instances.

Results



Execution Time Comparison (cuda_openmp_comparison)
openmp
cuda
Time (seconds)
0
5
10
15
20
25
Input Instance (Sorted by First Implementation's Time)



Time Performance Improvement (Positive = Faster than openmp)
cuda
0
−100
−200
−300
−400
Time Improvement Relative to openmp (%)
E-n76-k8
P-n101-k4
E-n101-k8
E-n101-k14
X-n110-k13
X-n120-k6
X-n106-k14
X-n115-k10
X-n101-k25
X-n139-k10
X-n143-k7
X-n134-k13
F-n135-k7
X-n129-k18
CMT4
X-n157-k13
X-n162-k11
X-n167-k10
X-n153-k22
X-n125-k30
X-n186-k15
CMT5
X-n190-k8
X-n204-k19
X-n209-k16
X-n214-k11
X-n181-k23
X-n237-k14
X-n176-k26
X-n200-k36
X-n233-k16
Golden_17
X-n228-k23
Golden_9
X-n256-k16
X-n251-k28
X-n223-k34
Golden_13
X-n172-k51
X-n195-k51
X-n275-k28
X-n280-k17
X-n284-k15
X-n219-k73
X-n261-k13
X-n148-k46
X-n270-k35
X-n242-k48
Golden_10
X-n303-k21
X-n308-k13
Golden_14
X-n317-k53
X-n247-k50
X-n327-k20
Golden_18
X-n298-k31
X-n322-k28
X-n294-k50
X-n331-k15
X-n289-k60
X-n266-k58
Golden_15
Golden_11
X-n367-k17
X-n359-k29
X-n401-k29
X-n351-k40
X-n393-k38
X-n344-k43
X-n313-k71
Golden_20
X-n384-k52
Golden_12
X-n449-k29
X-n459-k26
X-n336-k84
X-n439-k37
X-n411-k19
Golden_19
X-n429-k61
Golden_16
tai385
X-n376-k94
X-n502-k39
X-n480-k70
X-n573-k30
X-n491-k59
X-n548-k50
X-n561-k42
X-n469-k138
X-n641-k35
X-n536-k96
X-n420-k130
X-n613-k62
X-n627-k43
X-n655-k131
X-n513-k21
X-n599-k92
X-n586-k159
X-n716-k35
X-n685-k75
X-n701-k44
X-n783-k48
X-n524-k153
X-n895-k37
X-n801-k40
X-n856-k95
X-n876-k59
X-n837-k142
X-n957-k87
X-n670-k130
X-n819-k171
X-n749-k98
X-n733-k159
X-n766-k71
X-n1001-k43
X-n979-k58
X-n916-k207
X-n936-k151
Leuven1
Leuven2
Antwerp1
Antwerp2
Ghent1
Ghent2
Brussels1
Brussels2
Flanders1
Input Instance

Thank You!