DTUMOS Analysis and Optimization for Intelligent Transportation Systems

1st Mir Sulaiman Sultan
Computer Science
Wilfrid Laurier University
Waterloo, Canada
sult2271@mylaurier.ca

2nd Sukhjit Singh Sehra
Computer Science
Wilfrid Laurier University
Waterloo, Canada
ssehra@wlu.ca

***Abstract*—In this paper I outline a comprehensive analysis and optimization strategy for Digital Twin Urban Mobility Operating System (DTUMOS). I conducted a deep structural analysis of the code, completing a static, dynamic, and asymptotic analysis of the system, identifying severe bottlenecks in the sequential dispatching logic. The original framework relied on individual sequential calls to Open Source Routing Machine (OSRM) API, and it suffered from long execution times, commonly exceeding an estimated completion time of over 260 hours for an 18-hour urban scenario. To combat the extremely long execution times for the current system, I implemented a vectorized approach using batching and leveraging matrix distance calculations, coupling that with request chunking and parallel multi-threaded execution. This improved the runtime significantly seeing a 10.8x performance increase when running the system on a 10 thread configuration. Theoretical analysis as proposed later in the paper suggests a near linear scalability as CPU compute scales. The results, supported by a flame graph analysis, demonstrate that optimizing the interface between the simulation engine and the OSRM routing service is critical for enabling scalable urban mobility runtime analysis.**

***Index Terms*—Digital Twin, Urban Mobility, OSRM, Batch Processing, Parallel Computing, Route Optimization, Performance Engineering, Smart Cities, Dispatch Systems**

I. Introduction

The rapid urbanization of cities around the world has led to the necessary development of digital twin systems like DTUMOS. These systems are used to model, analyze, and optimize complex transportation networks. Systems like DTUMOS are also used as important tools for city planners and service providers enabling high-fidelity scenarios from fleet management to real time ride sharing predictions. However, the fidelity is constrained by the necessary computational complexity introduced by modeling interactions between thousands of agents on top of a dynamic road network.

The heart of any urban mobility simulation is the routing engine, used to calculate the distance and travel times between Origin-Destination (OD) pairs. In an agent based model simulation, the dispatching algorithm may need to calculate a cost matrix between N active passengers and M available vehicles in order to achieve optimal assignments. As the simulation scales, the routing requests grow at a scale of N x M, introducing a potential significant computational hurdle.

In the initial implementation of DTUMOS, using data from the Seoul metropolitan area, we encountered severe performance limitations driven by routing overhead. The Seoul implementation involves over 6,000 passengers and 613 vehicles over an 18-hour period, which amounts to millions of routing calls over the length of the simulation. The sequential HTTP requests to OSRM resulted in an estimated completion time of over 260 hours, which renders the system as extremely impractical for any iterative research or real-time analysis.

This paper proposes a high-performance structural optimization for OSRM based simulations to address these scalability issues. I introduce a vectorized dispatching framework that replaces the sequential API calls with a clustered batch approach with matrix distributions of distances and durations. This approach implements intelligent request chunking to respect server-side constraints and leverages parallel execution threads to maximize throughput.

The primary contributions of this work are:

- **Algorithmic Optimization**: A transformation of the dispatching logic from $O(N \times M)$ sequential operations to batched parallel execution.
- **DTUMOS Analysis**: A rigorous profiling of the simulation bottleneck, identifying the trade-offs between network I/O reduction and JSON parsing overhead.
- **Empirical Results**: My implementation demonstrated 10.8x speedup, reducing simulation time from over 260 hours to approximately 24 hours using a10-threadd distribution.

The rest of this paper is organized as follows: Section II reviews related work. Section III formulates the mathematical model of the dispatch problem. Section IV details the proposed vectorized optimization framework. Section V presents the algorithmic complexity analysis. Section VI discusses the experimental results, and Section VII concludes the paper.

II. Related Work

A. Digital Twin Urban Mobility Operating Systems

Digital Twin Urban Mobility Operating System (DTUMOS) has emerged as a critical and immensely useful framework used for simulating and optimizing large-scale transportation networks. Similar platforms like SUMO [14] and MATSim [13] provide a microscopic and agent based simulation capability, but they often require adaptation for use in any

real-time application contexts. Ruch et al. has developed Amodeus [15], a specialized testing platform for on-demand autonomous mobility, however Amodeus struggles to scale into city-wide fleets and reproduce accurate routing. To bridge these gaps, Yeon et al. [3] introduced the foundational structure of DTUMOS, which demonstrated its capability to handle over 20,000 vehicles and 400,000 requests per day. Yeon et al.'s work on DTUMOS highlighted the potential digital twins have for solving real-world problems and assisting with real issues like the Seoul taxi shortage, integrating real world taxi trip data and smart car logs. However, key limitations such as scalability and performance optimizations were identified, particularly when scaling to large-scale metropolitan areas and when requiring real-time responsiveness. Recent studies explored various different heuristics and approaches when dealing with vehicle routing problems, like the hybrid swarm intelligence [10] and the forest vehicle routing optimization [11], along with dynamic high capacity ride sharing algorithms [16]. But these algorithms and heuristics more often focus on algorithmic efficiency rather than underlying retrieval of the data from the current architecture. This paper addresses the latter by working to resolve the computational bottleneck introduced by the current sequential nature of the data retrieval system by optimizing the routing engine interface.

B. ETA Prediction and Traffic Simulation

Accurate Estimated Time of Arrival (ETA) predictions are an essential part of any reliable urban mobility simulation. Lin et al. [4] proposed TransETA, a hybrid deep learning model which integrated Graph Convolutional Networks (GCN) and Transformers to improve ETA accuracy. Using this system TransETA attempts to account for local congestion and dynamic trajectory data. While their work focuses on the accuracy of the predictive model, my research focuses on the throughput of the routing queries. In any large-scale agent-based simulation, the latency introduced from receiving ETAs becomes a critical factor. The vectorized batch approach introduced in this paper is complementary to any state-of-the-art ETA model, providing the necessary optimized infrastructure to query data more efficiently at scale.

III. Mathematical Modelling and Problem Formulation

Following the formalization approach for large-scale combinatorial problems described by Sugiarto [5], we model the dispatch optimization as follows.

A. Notation and Definitions

Let $G = (V, E)$ be the directed graph representing the road network, where V is the set of intersections and E is the set of road segments. Let $R = \{r_1, r_2, \ldots, r_N\}$ denote the set of active trip requests (passengers) at time t. Let $K = \{k_1, k_2, \ldots, k_M\}$ denote the set of available vehicles. The travel cost (time) between any two locations i and j in G is denoted by C_{ij}.

B. Dispatch Optimization Problem

The goal of the dispatch system is to assign each request $r_i \in R$ to a vehicle $k_j \in K$ such that the total system cost is minimized. We define a binary decision variable x_{ij}:

$$x_{ij} = \begin{cases} 1 & \text{if request } r_i \text{ is assigned to vehicle } k_j \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

The objective function is formulated as:

$$\text{Minimize } Z = \sum_{i=1}^{N} \sum_{j=1}^{M} x_{ij} C_{ij} \tag{2}$$

Subject to:

$$\sum_{j=1}^{M} x_{ij} = 1, \quad \forall i \in \{1, \ldots, N\} \tag{3}$$

$$\sum_{i=1}^{N} x_{ij} \leq 1, \quad \forall j \in \{1, \ldots, M\} \tag{4}$$

(assuming one-to-one matching for simplicity in this context).

C. Routing Overhead Formulation

The calculation of the cost matrix $C = \{C_{ij}\}$ is the computational bottleneck. In a sequential approach, the total time T_{seq} to retrieve C is:

$$T_{seq} = \sum_{k=1}^{N \times M} (t_{net} + t_{proc}) \tag{5}$$

where t_{net} is the network latency per request and t_{proc} is the processing time.

In the proposed vectorized approach, we batch requests into chunks of size B. The total time T_{vec} is:

$$T_{vec} = \sum_{l=1}^{\lceil (N \times M)/B \rceil} (t_{net} + t_{batch\_proc}) \tag{6}$$

Since t_{net} is incurred only once per batch, and $\lceil (N \times M)/B \rceil \ll N \times M$, we achieve significant latency reduction.

IV. Proposed Optimization Framework

I propose a solution to overcome the computational bottleneck that comes with sequential routing queries. I developed a vectorized dispatching architecture. This vectorized approach shifts the paradigm between the DTUMOS engine and the OSRM server from a request-response pattern to a batch-processing model.

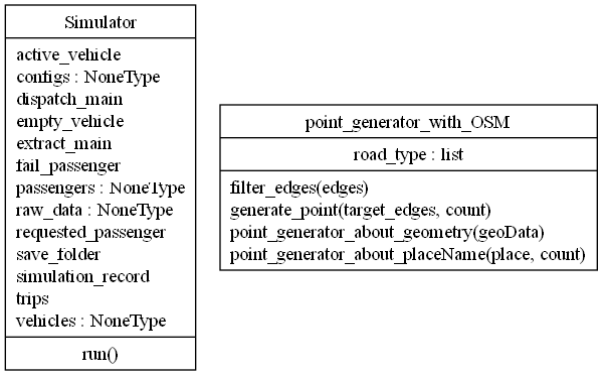

Fig. 1. Class diagram of the DTUMOS core modules. The `OSRM_Routing` class interfaces with the `Dispatch` logic to handle vectorized requests.

A. Vectorized Batch Processing

The standard OSRM `/route` endpoint is designed for single OD pair queries, which then return detailed trajectory data, including geometry data. For dispatching simulation in DTUMOS, only the aggregate metrics distance and duration are needed. We transitioned to the OSRM `/table` endpoint, which computes the duration matrix for a set of source and destination coordinates.

By vectorizing the input, we aggregate K individual dispatch requests into a single HTTP transaction. This reduction in network I/O is extremely important, as the TCP handshake and the HTTP overhead for the millions of individual calls contributed to the vast majority of the time to compute for the simulation.

B. Intelligent Request Chunking

While the Table API supports batching, it is constrained by the server's length limit and internal thread default/capacity. Simply sending a $N \times M$ matrix in a single call can lead to timeouts or server instability. To combat this I implemented an intelligent chunking strategy that breaks down the global OD matrix into manageable sub-matrices.

Based on empirical testing with our Dockerized OSRM instance (configured with a custom `--max-table-size` larger than the default 50), I determined an optimal chunk size of 100 OD pairs per request. While larger chunks are technically possible, they increase the per-request latency and reduce the opportunities to implement parallelization. Smaller chunks maximize the thread utilization by ensuring that more threads remain active, this avoids scenarios where one thread is bottlenecking the simulation progression by processing a chunk at 100 percent thread usage while other threads are inactive.

C. Parallel Execution Framework

To improve the execution time of the computation, I implemented a concurrent execution model using Python's `ThreadPoolExecutor`. The simulation's dispatch logic was refactored to make use of multiple threads to process chunks simultaneously.

I configured the executor with 10 worker threads, and allowed my OSRM instance to use `--threads=10`. This alignment ensures that the client-side parallelism matches the server-side concurrency capabilities, maximizing our hardware utilization without causing context-switching thrashing. However, theoretical speedup S will follow Amdahl's Law, where the parallelizable portion of the workload is significantly sped up, limited only by the serial overhead of result aggregation of JSON parsing.

V. Algorithmic Complexity Analysis

To identify performance potential bottlenecks, I conducted a static time complexity analysis of all the files and functions within the main DTUMOS module. Table I summarizes my findings of the asymptotic complexity of key modules and functions within DTUMOS.

TABLE I
Time Complexity of Key DTUMOS Modules

Module	Key Function	Complexity
`dispatch.py`	`ortools_dispatch`	$O(N^2)$
`level_of_service.py`	`figure_1, figure_2`	$O(N^2)$
`osrm_routing.py`	`get_res`	$O(1)$
`simulator.py`	`run`	$O(N)$
`dispatch_cost.py`	`dispatch_cost_matrix`	$O(N)$

This analysis confirmed that the dispatch function (`ortools_dispatch`) and certain other key functions within DTUMOS exhibit quadratic complexity $O(N^2)$, where N represents the number of agents (passengers/vehicles). Although the asymptotic analysis of the routing (osrm_routing.py) itself revealed an $O(1)$ per call, the cumulative effect of calling it within the quadratic dispatch loop created the primary bottleneck.

Moving to a vectorized approach does not change the theoretical complexity of the matching algorithm itself (which is dependent on the solver), but it drastically reduces the constant factors associated with the data retrieval. The implementation of this optimization transitions from $O(N \times M)$ network calls to $O(\frac{N \times M}{B})$ calls, we shift the bottleneck from network I/O to CPU-bound JSON parsing. Which is easily parallelizable.

VI. Experiments and Results

A. Experimental Setup

The optimization was tested using a full-scale simulation of the Seoul metropolitan area. The simulation parameters included approximately 6,000 passengers and 613 vehicles, operating over an 18-hour period (1,080 time steps). The OSRM backend server was deployed using a Docker container with the Multi-Level Dijkstra (MLD) algorithm enabled and configured to use 10 threads.

B. Performance Improvement

The baseline implementation, using the sequential OSRM routing calls was estimated to take over 260 hours to complete the simulation. This extreme delay was almost completely

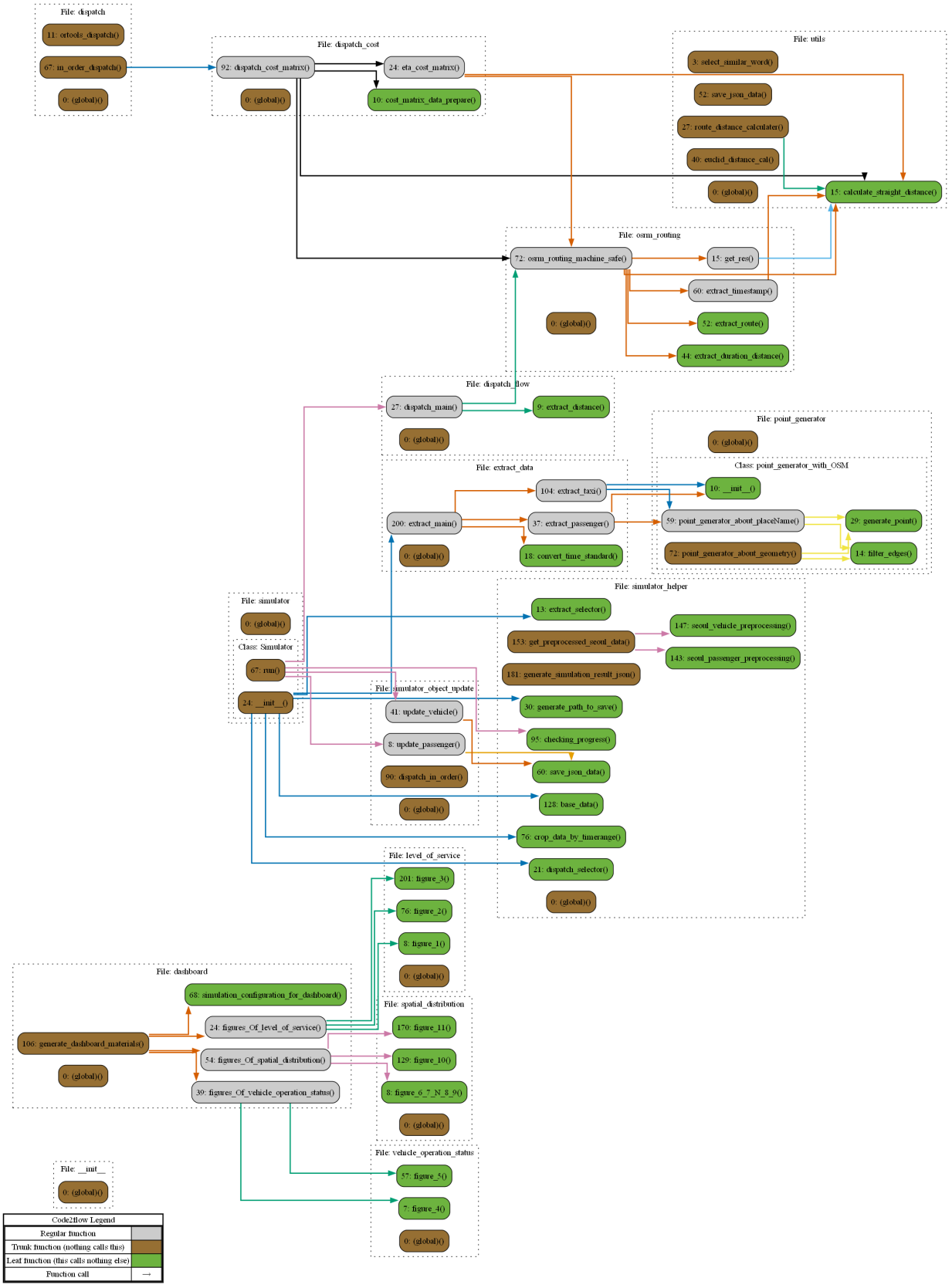

Fig. 2. Call graph illustrating the optimized dispatch flow. The `dispatch_vehicles` function orchestrates the batching and parallel execution of routing requests.

Fig. 3. Dependency graph generated by Pyan, illustrating the high call density within the dispatch module. The dense cluster of edges pointing to routing functions visually corroborates the algorithmic bottleneck.

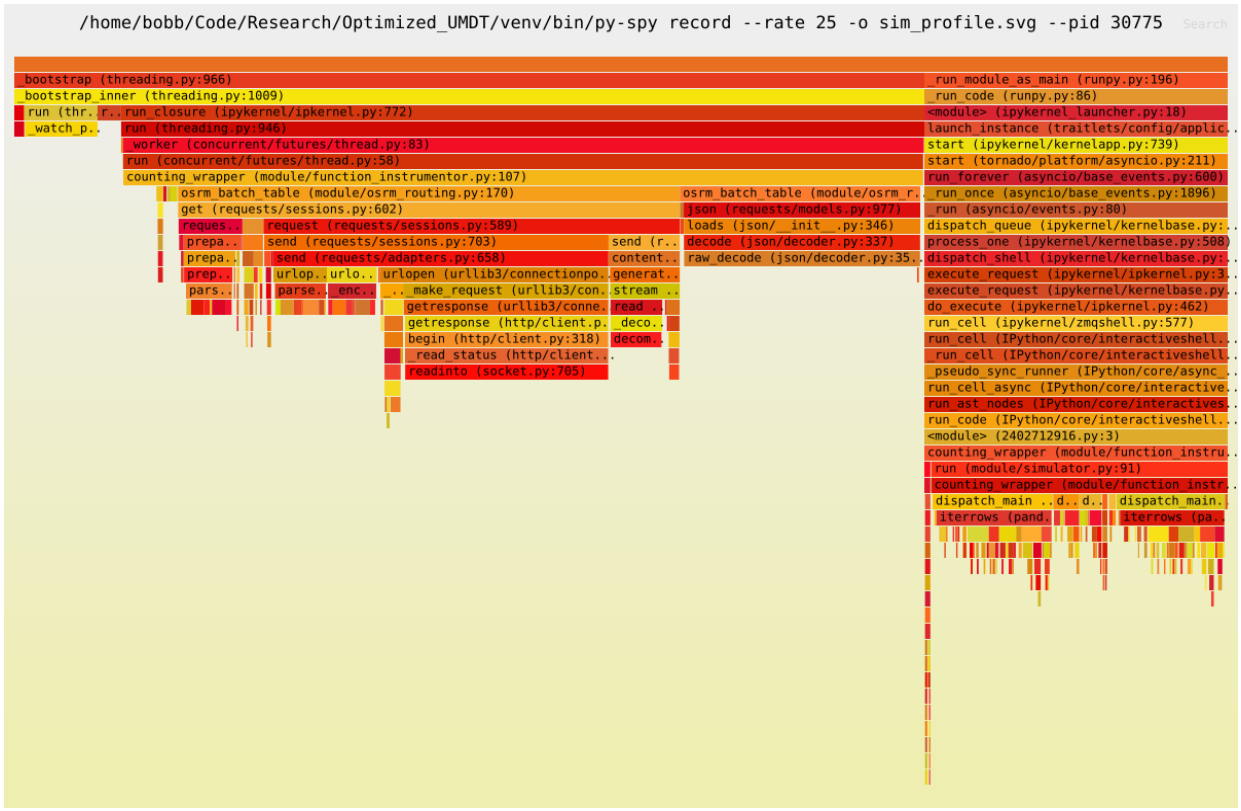

Fig. 4. Performance profiling comparison showing the reduction in routing overhead. The flame graph illustrates the shift in computational density from network I/O wait times to data processing.

caused by the cumulative latency of the hundreds of thousands of individual HTTP requests to the OSRM server.

After implementing the proposed vectorized batching and multi-threaded execution strategy, the total simulation time was reduced from over 260 hours to approximately 24 hours. Representing a speed up factor of $10.8\times$.

C. Profiling and Bottleneck Analysis

To understand the computational load of the simulation at a deeper level, I employed statistical profiling tools such as `py-spy` [9] to create and outline the lower-level implications that the code has on the machine it is leveraging. The profiling data, summarized in Table II, reveals the breakdown of execution time in the optimized vectorized implementations.

The output from this statistical profiling analysis identifies that the `osrm_batch_table` function is the primary consumer, accounting for approximately 41% of the total execution time. More significantly, the bottleneck has shifted from a total network overhead bottleneck to a mix of I/O network and CPU-bound data processing. JSON decoding alone now takes up around 19% of the execution time, which indicates that parsing these large payload outputs from OSRM is now a significant cost contributor. Additionally, URL encoding of the coordinate strings now accounts for ∼8%, suggesting

TABLE II
PROFILING BREAKDOWN OF OPTIMIZED DISPATCH

Operation	Time (%)
JSON Decoding (CPU)	$\sim 19\%$
Socket Read (Network Wait)	$\sim 17\%$
URL Encoding	$\sim 8\%$
HTTP Request Prep	$\sim 5\%$
Response Decompression	$\sim 4\%$

potential opportunities for improvement using preencoding or binary protocols. Despite these overheads, the reduction of the massive TCP handshake overhead stemming from sequential OSRM calls resulted in a $10.8\times$ performance gain.

D. Scalability Analysis

The use of a 10-thread `ThreadPoolExecutor` along with an equal OSRM server thread count resulted in efficient resource utilization. Theoretical analysis suggests that this architecture scales almost linearly, assuming the thread distribution between client and server is appropriate. The $10.8\times$ speedup that was observed is directly correlated with the thread allocation that it was provided, confirming that the system could effectively leverage additional hardware resources. Future work could explore distributing the OSRM backend across multiple nodes to further reduce the 24-hour execution time.

VII. Conclusion and Future Work

This study demonstrates the significant dependence of the overall performance of large-scale urban mobility simulations on the efficiency of the routing engine interface. Through transitioning from a sequential request-response pattern to a vectorized, parallelized batch-processing architecture, we achieved a $10.8\times$ reduction in simulation execution time.

The optimization strategy, which comprises OSRM Table API usage, intelligent request chunking, and thread-pool execution, proved effective at mitigating network latency bottlenecks. However, profiling has revealed that JSON parsing has become the dominant cost in the simulation.

Future work will focus on further optimization of this data ingestion pipeline. To mitigate the decoding overhead, I plan to explore the adoption of high-performance JSON parsing libraries or binary data formats for OSRM responses. I also plan to investigate the potential of offloading matrix calculations to GPU-accelerated solvers. In addition, I aim to implement adaptive chunking algorithms and variable thread allocation strategies that adjust resource usage dynamically in response to real-time server load. To ensure optimal resource utilization, I will also incorporate work-stealing mechanisms that distribute tasks evenly across all assigned threads in future implementations.

Beyond the optimization of the routing interface, future research will also explore algorithmic improvements to reduce the $O(N \times M)$ complexity of the dispatch matching problem. Spatial indexing techniques, such as KD-trees or Ball trees, could enable K-nearest-neighbor queries that limit vehicle candidates to only the K closest to each passenger. This would reduce the cost matrix computation from $O(N \times M)$ to $O(N \times K)$ where $K \ll M$, potentially achieving order-of-magnitude improvements for large fleet sizes. These approaches sacrifice finding the absolute best solution in favor of faster computation, which may be an acceptable compromise for real-time applications.

References

[1] Project OSRM, "Open Source Routing Machine," *GitHub Repository*, https://github.com/Project-OSRM/osrm-backend.

[2] D. Luxen and C. Vetter, "Real-time routing with OpenStreetMap data," in *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2011, pp. 513–516.

[3] H. Yeon, T. Eom, K. Jang, et al., "DTUMOS, digital twin for large-scale urban mobility operating system," *Sci Rep*, vol. 13, p. 5154, 2023.

[4] S. Lin, Y. Xu, S. Zhao, Y. Wang, and J. Xu, "TransETA: transformer networks for estimated time of arrival with local congestion representation," *Applied Intelligence*, vol. 53, pp. 30384–30399, 2023.

[5] Sugiarto, "A Hybrid Optimization Algorithm for Large-Scale Combinatorial Problems in Cloud Computing Environments," *ALCOM: Journal of Algorithm & Computing*, vol. 1, no. 1, pp. 001–012, 2025.

[6] SemanticBeeng, "pyan: Static analysis of python code to determine call dependency graphs," GitHub Repository, https://github.com/SemanticBeeng/pyan.

[7] Pylint Developers, "Pyreverse: UML diagram generator," *Pylint Documentation*, https://pylint.pycqa.org/en/latest/pyreverse.html, 2025.

[8] S. Rogowski, "code2flow: Call graph generator for Python," GitHub Repository, https://github.com/scottrogowski/code2flow, 2021.

[9] B. Fredrickson, "py-spy: Sampling profiler for Python programs," GitHub Repository, https://github.com/benfred/py-spy, 2023.

[10] Y. Shen, M. Liu, J. Yang, Y. Shi, and M. Middendorf, "A Hybrid Swarm Intelligence Algorithm for Vehicle Routing Problem With Time Windows," *IEEE Access*, vol. 8, pp. 99374–99391, 2020.

[11] H. Havaeji, "Optimizing a Transportation System Using Metaheuristics Approaches (EGD/GA/ACO): A Forest Vehicle Routing Case Study," *World Journal of Engineering and Technology*, vol. 12, no. 1, pp. 100–115, 2024.

[12] M. Cheng, J. Li, P. Bogdan, and S. Nazarian, "H2O-Cloud: A Resource and Quality of Service-Aware Task Scheduling Framework for Warehouse-Scale Data Centers," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3676–3687, 2020.

[13] K. W. Axhausen, A. Horni, and K. Nagel, *The Multi-agent Transport Simulation MATSim*. Ubiquity Press, 2016.

[14] M. Behrisch, L. Bieker, J. Erdmann, and D. Krajzewicz, "SUMO–simulation of urban mobility: An overview," in *Proceedings of SIMUL 2011, The Third International Conference on Advances in System Simulation*, 2011.

[15] C. Ruch, S. Hörl, and E. Frazzoli, "Amodeus, a simulation-based testbed for autonomous mobility-on-demand systems," in *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, pp. 3639–3644, IEEE, 2018.

[16] J. Alonso-Mora, S. Samaranayake, A. Wallar, E. Frazzoli, and D. Rus, "On-demand high-capacity ride-sharing via dynamic trip-vehicle assignment," *Proceedings of the National Academy of Sciences*, vol. 114, no. 3, pp. 462–467, 2017.