Real-Time Routing in Traffic Simulations: A Distributed Event Processing Approach

Paul Heinrich
Transport Systems Planning
Technical University of Berlin
Germany
heinrich@vsp.tu-berlin.de

Janek Laudan
Transport Systems Planning
Technical University of Berlin
Germany
laudan@vsp.tu-berlin.de

Kai Nagel
Transport Systems Planning
Technical University of Berlin
Germany
nagel@vsp.tu-berlin.de

***Abstract*—This paper explores the integration of distributed event processing in the context of real-time routing within a parallel transportation simulation framework for the Multi-Agent Transport Simulation (MATSim). A novel simulation prototype, utilizing Rust and MPI, demonstrates significant reductions in computation time by applying domain decomposition of the network and assigning each part to a separate process. However, sharing and processing events within this setup remains challenging. We present a proof of concept for integrating distributed event processing. Our evaluation has shown that real-time routing significantly increases simulation runtime because uneven distribution of routing requests increases load imbalances and makes speedups less efficient. Sharing event data between processes benefits from smaller subdomains. However, global synchronization for sharing data leads to waiting times introduced by load imbalances.**

***Index Terms*—Mulit-Agent Transport Simulation, Distributed Computing, MPI, Real-Time Routing**

I. Introduction

Multi-agent simulations are powerful tools for transportation planning. The persistent demand for larger and more finely detailed models amplifies the computational burden of these simulations. Executing them within a parallel or even distributed environment is the key for expanding models while maintaining a reasonable runtime.

In the context of multi-agent simulations, Rousset *et al.* [1] provide a helpful overview of existing parallel and distributed approaches. The presented frameworks are implemented in different programming languages such as Java [2] or C++ [3], [4], providing abstract functionality for agent modeling and general communication as well as functionality for agents to query the state of neighboring agents, which may be handled by another process. However, the specific implementation of complex simulation logic, such as communicating a general traffic state, is generally left to the implementing user.

The traffic simulator Mobiliti [5] works as a parallel discrete event simulator instead of simulating fixed time steps, and achieves impressive speedups on distributed HPC infrastructure. Building on Mobiliti, Chan *et al.* [6] implement distributed ad-hoc routing similar to the concept presented in this article: each partition has a single router, which is aware of current travel times across the entire network. In contrast to our approach, travel times are only updated once they are higher than a certain factor. Following the actor model, each link takes on the responsibility of transmitting its current travel time to the routing components. Other distributed transport simulators like dSUMO [7] do not provide distributed event processing functionality.

Another well-established simulator is the the Multi-agent transport simulation (MATSim) [8] having a large community and many extension points. MATSim operates as a co-evolutionary algorithm with a three-phase iterative approach serving as an open-source framework for large-scale agent-based transportation simulations. Within MATSim, each synthetic person (*agent*) has a plan that represents its daily schedule. A plan consists of a concatenation of activities and legs. Activities are performed at specific times and locations, while legs represent journeys between activities. Agents optimize their travel behavior over multiple iterations of the same simulated day, leveraging the co-evolutionary algorithmic approach.

Each iteration consists of three phases: mobility simulation, scoring, and replanning, with the mobility simulation consuming most of the computational time. The mobility simulation involves simulating the movement of agents on the traffic network while considering the capacities of links and intersections. Currently, the default implementation of the mobility simulation in MATSim, QSim (Queue-based traffic flow simulation) [9], scales up to 8 processes and relies on native Java concurrency primitives. However, a new prototype in Rust achieves significantly reduced simulation times[1][10]. Utilizing the MPI (Message Passing Interface), it runs on distributed high-performance clusters. The parallelization approach uses domain decomposition dividing the network into subnetworks. Each process executes the mobility simulation for its subnetwork. Vehicles crossing a domain boundary are communicated as messages to neighboring processes.

Event handling (as described in Chapter II) in such a distributed environment is more complex than in monolithic software because data must be explicitly shared via messages instead of passing data pointers between functions within the same process. This complexity affects components responsible for event processing during simulation, such as real-time routers. These routers must be kept up-to-date with the actual

[1]https://github.com/matsim-vsp/parallel_qsim_rust

travel times across the entire network. Since vehicles can cause congestion on network links, travel times may change accordingly. The message passing functionality needs to be expanded, as the current message passing only supports point-to-point communication but communicating travel times of the whole networks relies on collective messaging.

The contribution of this paper is twofold: (i) we implement distributed event processing demonstrated through the example of ad hoc rerouting in the distributed prototype of MATSim and (ii) we assess the influence on the runtime performance.

II. Principles of Distributed Event Processing

The QSim generates events that describe the movement of the agents within the mobility simulation, such as ending an activity, entering a network link with a vehicle or leaving a vehicle. By reading the event stream, consumers gain insight into the state of the QSim. Traffic state information, such as travel times for links, can be extracted from this event stream.

Various components can be plugged into the QSim, with a distinction between online and offline processing. Offline processing can be conducted completely independently from the simulation, typically on the output files after the simulation has finished. In contrast, ad-hoc routing necessitates online event processing, which is performed while the simulation runs, as it influences the behaviour of the simulation.

In general, there are several approaches to integrate such components into the distributed QSim. They can be executed by the same QSim processes, by the same cluster or even remotely. For the ad-hoc router as the first additional component of the distributed QSim, we decided to implement it on the same processes as the QSim itself. A description of the implementation is provided in Chapter III.

The prototype implementation of the distributed QSim demonstrates a hundredfold speedup over the current default implementation. Regardless of the chosen architecture, maintaining the performance gains achieved by the distribution of work is crucial when integrating new features. Performing a routing query takes significantly longer than a simulation step without routing: milliseconds versus microseconds. Introducing ad-hoc routing can disrupt load balancing between processes, which greatly impacts overall performance.

Another crucial aspect is to ensure that the event processing does not degrade the overall application performance. Specifically, in the context of routing, this concern pertains to the router update step. In general, the update process can be divided into three phases: (i) aggregating current events into update messages that the service understands (**preprocessing**), (ii) communicating these updates (**communication**), and (iii) making the service apply these updates (**postprocessing**).

Preprocessing and communication can be influenced by the manner a service is integrated in the mobility simulation and by the chosen architecture. To take full advantage of the distributed architecture, it is essential to invest effort in preprocessing on each subdomain. As the number of domains increases, the runtime of preprocessing per domain decreases.

III. Distributed Prototype

The open-source prototype [11] is implemented in the programming language Rust and utilizes MPI for communication. Each process executes the QSim for a specific spatial domain, synchronizing with all neighbor processes after completing a simulated time step. It is important to note that this synchronization step is performed locally. When a process has received messages from all of its neighbors, it can proceed, even if its neighbors are still awaiting messages from other processes. Consequently, a process can advance one time step ahead of its neighbors.

Given that all communication relies on parallelization with MPI, we opt to integrate the router into each process rather than as a separate entity. This approach minimizes communication overhead and eliminates the need for asynchronous code.

A. *Choice of Routing Algorithm*

The prototype incorporates a within-day replanning engine that employs ad-hoc routing to demonstrate distributed event processing. If the engine is turned on, instead of adhering to predefined routes from their plans, agents will perform ad-hoc routing for the upcoming trip. This ad-hoc re-routing includes access and egress legs to and from the main leg, as well as the route of the main leg itself. The replanning engine is constructed based on the principles outlined by Dobler *et al.* [12].

Numerous well-established routing algorithms designed for graphs with dynamic travel times are available. Bast *et al.*[13] highlights two effective approaches for addressing this scenario: goal-directed algorithms such as the A* algorithm using landmarks [14], and hierarchical techniques such as CCH (Customizable Contraction Hierarchies) [15]. A* algorithms generally have slower query times but relatively fast travel time updates, while CCH is the opposite. Due to its comparably straightforward implementation, we opt for a custom A* landmarks implementation. Initially, we attempted to leverage the "rust_road_router" CCH implementation[2] to achieve fast query times. However, due to heavy reliance on unstable Rust features and limited hardware compatibility, we opted for an alternative approach.

B. *Ad-Hoc Router Implementation*

The simulation supports various vehicle types, each distinguished by varying maximum speeds. Preprocessing the landmark data per vehicle type improves the goal direction, as the preprocessed data is then closer to the actual travel times of each vehicle. The preprocessing is performed once before the simulation commences. If the travel times increase due to congestion on the simulated network, the algorithm still works correctly, albeit with slower query times.

[2]https://github.com/kit-algo/rust_road_router

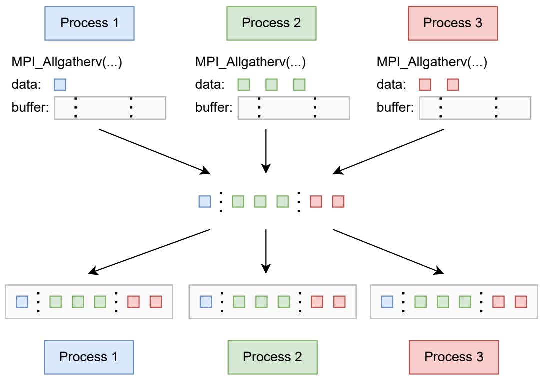

Figure 1: The `MPI_Allgatherv` Call. The squares correspond to the data that will be sent. The call is made on each process. A buffer with the corresponding size of all data must be allocated by the receiving process.

C. Router Update

To update travel times in the router modules, each process must share its travel times with every other process. To minimize global synchronization calls, travel times are shared every 15 simulated minutes. The router update generally works as follows.

a) Preprocessing: Each process computes the average travel times for each link within its spatial domain over the past 15-minute interval. For links that have not experienced any traffic during this interval, the initial travel time is assumed. Initial travel times are only communicated to other processes if a different travel time was communicated during the previous update step.

b) Communication: MPI's `Allgatherv` function is utilized to communicate preprocessed travel times between all processes. Figure 1 illustrates the data exchange for this MPI call. Local travel times computed for each domain are sent to all other processes. Technically, this entails one `MPI_Allgather` and one `MPI_Allgatherv` call per vehicle type. The initial call serves to share the actual size of each process' travel time message since all processes send travel time updates of varying message size. As buffers for the received messages must be pre-allocated, it is necessary to share the message sizes before initiating the `MPI_Allgatherv` call. The second call is to share the actual travel time messages. After that the messages are deserialized.

c) Postprocessing: Afterwards, each process updates the router with the new travel times. Since the initial travel times are derived from the speed limit per link, the current travel times will always be greater or equal. Therefore, in the case of the A* with landmarks algorithm, the task merely involves copying the new travel times into the graph's data structure.

It is important to note that postprocessing takes a fixed amount of time, regardless of the number of processes, and must be carried out on each process individually.

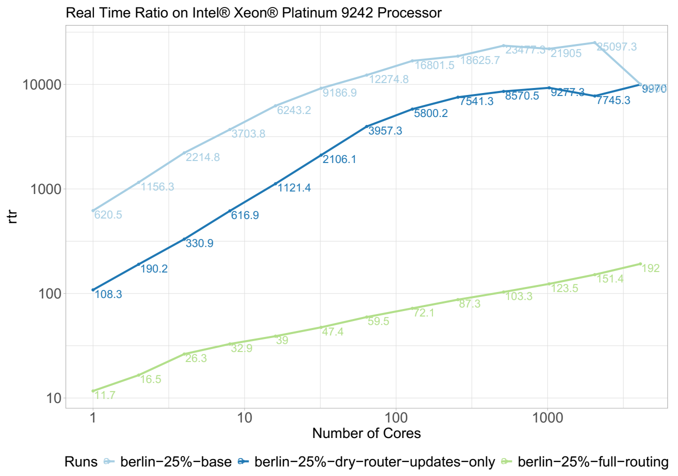

Figure 2: Real time ratios of full run with routing, dry run without routing but with router updates and base run with no routing component at all.

IV. Evaluation

In the following, we investigate the impact of routing on simulation runtime and evaluate the performance of router updates. Our performance measurements were conducted on the HLRN (Norddeutscher Verband für Hoch- und Höchstleistungsrechnen)[3] cluster. Each node is equipped with an Intel Xeon Platinum 9242 @ 2.3 GHz (71.5 MB cache, 48 cores and 96 threads). The Berlin 25% scenario[4] is used, simulating traffic for 24 hours. Measurements were conducted with 48 processes per node.

Evaluating the traffic state is beyond the scope of this paper. Studies such as [16] and [17] examine the effect of within-day replanning on the traffic assignment. Instead, our focus lies on the technical aspects and the evaluation of their performance within the distributed prototype.

A. Overall Runtimes

We ran the Berlin scenario with three configurations: one activating the entire ad-hoc replanning mechanism (full run), another with router update enabled but without rerouting (dry run), and a base case with no replanning enabled. We measure the real-time ratio (RTR), which represents the ratio between simulated time and the wall clock time required for the simulation. The use of RTR (Real Time Ratio) is based on research indicating that the runtime tends to saturate at the same level, determined by hardware latency and implementation, rather than the size of the simulated scenario. In contrast, the level at which speed-up saturates depends on the scenario size. Figure 2 shows the RTR of both runs.

During the dry run, we achieve a speedup of 92, while the full run exhibit a speedup of 16. The best RTR we achieve are 9970 for the dry run and 192 for the full run. Since the fastest dry run takes 2.5 times longer than the fastest base run,

[3]https://www.hlrn.de/doc/display/PUB/Compute+node+partitions
[4]https://zenodo.org/records/10973034

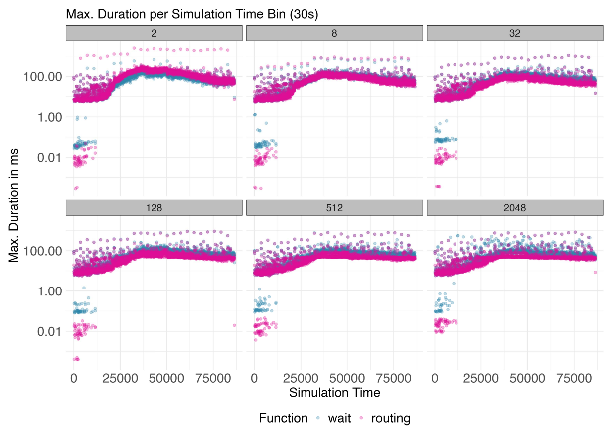

Figure 3: Maximum wait and routing durations in 30s time bins for different numbers of processes.

it can be inferred that this difference represents the overhead factor introduced by router updates. Notably, the curve for the full run does not decrease, suggesting the potential for further performance improvements with additional processes. However, due to limited resources, we can not go any further.

The substantial disparity between dry and full run is primarily attributed to the time-consuming nature of routing itself.

Routing consumes significantly more time than the actual simulation logic of moving vehicles through the network. Therefore, the distribution of routing requests strongly influences the synchronization of processes and is the primary obstacle to achieving higher speedups for the full run.

In an ideal scenario where routing requests are evenly distributed, the longer duration of routing compared to other QSim steps alone would not result in high wait durations. However, in the worst-case scenario, if only one process handles all the routing during a time step, every other process must wait exactly this amount of time. This situation is depicted in Figure 3.

The figure displays both wait times and routing times. Each data point corresponds to the maximum duration among all processes within 30-second time intervals. Specifically, routing times represent the time taken for a process to complete all routing tasks of one time step, while wait times indicate the durations after a process has completed all its work and awaits update messages from its neighbours. The hourly outliers are artifacts due to freight agents departing at full hours.

Figure 4 illustrates why wait and routing times in Figure 3 remain relatively constant and do not decrease. Although we determined that the mean routing duration is $7.3ms$, the maximum routing durations are around $50ms$. Since the maximum routing time is around that value for most time steps, we cannot expect shorter waiting times – even with more processes.

B. General Event-Handling Process

Figure 5 illustrates the mean durations of the router update steps. The preprocessing phase exhibits a substantial reduction

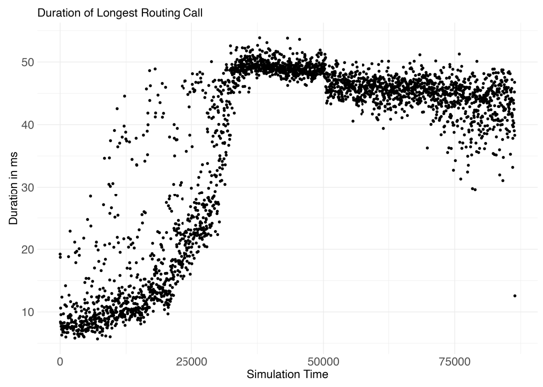

Figure 4: Maximum runtime of a routing call.

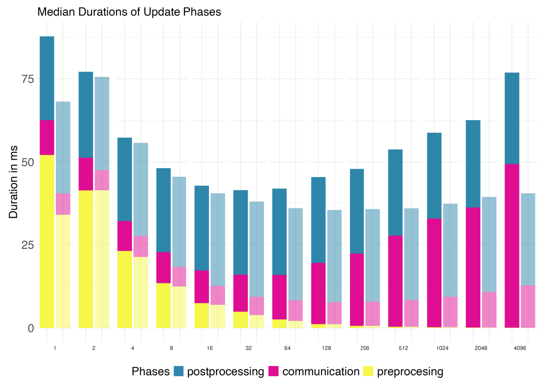

Figure 5: Mean durations of the router update process depending on the number of processes. Transparent bars are for the dry runs, others for full replanning including routing.

from $42ms$ with 1 process to $0.08ms$ with 4096 processes. This behavior is expected since preprocessing is performed on the spatial domains, benefiting from smaller spatial domain sizes. The duration of postprocessing remains constant in both runs. Since each process handles all the data during postprocessing, no speedup can be expected.

The most significant difference occurs during communication, which includes sync time, data exchange, and deserialization. Our assumption is that long routing times are the cause, but this requires additional investigation.

C. Implications on Further Architectural Choices

While further investigation is needed, our results already yield interesting implications. Routing accounts for the majority of wait times, and must be reduced to achieve higher runtime speedups.

One possible adjustment is to shorten routing time by using a faster algorithm, even for the case of longer post-processing during router updates. Given that these updates occur only every 900 time steps, this approach seems reasonable.

Alternatively, one could explore better utilization of the parallel architecture. Since the different routing queries are mutually independent, the problem is in principle “embarassingly parallel”. The main issue is that the routing requests do not arrive at the same time but rather sequentially in different time steps. Two options come to mind:

1) collect routing queries, e.g. by running all routing queries for departures during the next 900 time steps immediately after the travel times were updated; or
2) run routing processes in parallel with the execution of the mobility simulation.

For the second approach, a microservice architecture could be used, offering the advantages of an asynchronous and non-blocking implementation. With 15-minute update intervals, processes could (also here) submit all routing requests at the beginning of a new interval, allowing the router to re-order requests by priority. This approach enhances component isolation. However, it requires more communication and necessitates load balancing for the router instance. Regarding the router update itself, the preprocessing implementation scales effectively and could be employed in such an implementation.

V. Conclusion & Future Work

This study demonstrates the implementation of event processing in a distributed mobility simulation, focusing on ad-hoc routing using MPI. We find that high query times for routing significantly slow down the simulation. Nevertheless, utilizing more processes yields speedups for up to 4096 processes.

The global exchange of events is affected by the widely varying runtimes of the processes. While the preprocessing phase benefits from workload distribution, global synchronization can be costly, particularly if routing queries exceed the normal workload of a time step. In future studies, we intend to assess alternative architectural approaches, such as microservices, which could facilitate non-blocking event processing.

VI. Acknowledgements

We acknowledge the use of DeepL Write (https://www.deepl.com/write) and ChatGPT (OpenAI, https://chat.openai.com/ to correct grammar and expression.

The authors gratefully acknowledge the computing time made available to them on the high-performance computer "Lise" at the NHR Center NHR@ZIB. This center is jointly supported by the Federal Ministry of Education and Research and the state governments participating in the NHR (www.nhr-verein.de/unsere-partner).

References

[1] A. Rousset, B. Herrmann, C. Lang, and L. Philippe, "A survey on parallel and distributed multi-agent systems for high performance computing simulations," *Computer Science Review*, vol. 22, pp. 27–46, Nov. 2016.

[2] G. Cordasco, V. Scarano, and C. Spagnuolo, "Distributed MASON: A scalable distributed multi-agent simulation environment," *Simulation Modelling Practice and Theory*, vol. 89, pp. 15–34, Dec. 2018.

[3] N. Collier and M. North, "Parallel agent-based simulation with repast for high performance computing," *Simulation*, vol. 89, no. 10, pp. 1215–1235, Oct. 2013.

[4] L. S. Chin, D. J. Worth, C. Greenough, S. Coakley, M. Holcombe, and M. Gheorghe, *FLAME-II: A redesign of the flexible large-scale agent-based modelling environment*, https://epubs.stfc.ac.uk/manifestation/8576/RAL-TR-2012-019.pdf, Accessed: 2023-12-4.

[5] C. Chan, B. Wang, J. Bachan, and J. Macfarlane, "Mobiliti: Scalable transportation simulation using High-Performance parallel computing," in *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, IEEE, Nov. 2018, pp. 634–641.

[6] C. Chan, A. Kuncheria, and J. Macfarlane, "Simulating the impact of dynamic rerouting on metropolitan-scale traffic systems," *ACM Trans. Model. Comput. Simul.*, vol. 33, no. 1-2, pp. 1–29, Feb. 2023.

[7] Q. Bragard, A. Ventresque, and L. Murphy, "dSUMO: Towards a distributed SUMO," in *1st SUMO User Conference 2013*, Berlin, May 2013.

[8] A. Horni, K. Nagel, and K. W. Axhausen, *The Multi-Agent Transport Simulation MATSim*, en. London, England: Ubiquity Press, Jul. 2016.

[9] C. Dobler and K. W. Axhausen, *Design and implementation of a parallel queue-based traffic flow simulation*, en, 2011.

[10] J. Laudan, P. Heinrich, and K. Nagel, *Distributed parallel qsim implementation in rust [full paper submitted to conference ISPDC 2024]*, https://svn.vsp.tu-berlin.de/repos/public-svn/publications/vspwp/2024/24-09/LaudanEtAl2024DistributedQsimRust.pdf, Apr. 2024.

[11] J. Laudan and P. Heinrich, *Parallel qsim rust*, https://zenodo.org/records/10960723, Apr. 2024.

[12] C. Dobler and K. Nagel, "Within-Day replanning," in *Multi-Agent Transport Simulation MATSim*, London: Ubiquity Press, 2016.

[13] H. Bast, D. Delling, A. Goldberg, *et al.*, "Route planning in transportation networks," Apr. 2015. arXiv: 1504.05140 [cs.DS].

[14] D. Delling and D. Wagner, "Landmark-based routing in dynamic graphs," in *Experimental Algorithms*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 52–65.

[15] J. Dibbelt, B. Strasser, and D. Wagner, "Customizable contraction hierarchies," *ACM J. Exp. Algorithmics*, vol. 21, pp. 1–49, Apr. 2016.

[16] C. Dobler, M. Kowald, N. Rieser-Schüssler, and K. W. Axhausen, "Within-Day replanning of exceptional events," *Transp. Res. Rec.*, vol. 2302, no. 1, pp. 138–147, Jan. 2012.

[17] J. Illenberger, G. Flotterod, and K. Nagel, "Enhancing MATSim with capabilities of within-day re-planning," in *2007 IEEE Intelligent Transportation Systems Conference*, IEEE, Sep. 2007, pp. 94–99.