High-Performance Simulations for Urban Planning: Implementing Parallel Distributed Multi-Agent Systems in MATSim

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

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

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

***Abstract*—As semiconductor design approaches physical limits, computer processing speeds are stagnating. This poses significant challenges for traffic simulations, which are becoming more and more computationally demanding. To maintain fast execution times while accommodating more complex simulations, it is essential to utilize the parallel computing capabilities of modern hardware. This paper discusses the need for an updated architectural design in the MATSim traffic simulation framework to take advantage of parallel computing infrastructures. We introduce a prototype that adapts the existing traffic simulation logic to a distributed parallel algorithm. Extensive benchmarks have been conducted to evaluate the prototype's performance and identify its limitations. The results demonstrate that the prototype performs up to 100 times faster than the current implementation. Based on these findings, we advocate for the integration of a distributed traffic simulation within the MATSim framework and outline necessary steps to enhance the prototype.**

***Index Terms*—Mulit-Agent Transport Simulation, Distributed Computing, MPI, Parallel Computing**

I. Introduction

MATSim (Multi-Agent Transport Simulation) [1] is a well established software framework for agent-based traffic modelling. Operating on a mesoscopic scale, it supports simulating urban regions encompassing millions of individual travelers. As the capability of the model expands, so does the desire to enhance its scale or detail, increasing the computational demands of the software. At the same time, CPU (Central Processing Unit) clock rates have plateaued for almost two decades, and the miniaturization in semiconductor design is approaching physical limits [2]. Realizing performance improvements in MATSim to accommodate computationally expensive simulations while maintaining fast execution times requires leveraging the parallel computing capabilities of modern computer hardware.

Traffic in MATSim is modelled using synthetic persons which travel along a simulated road network to reach designated activity locations. Over multiple iterations of the same simulated day, synthetic persons explore various strategies to maximize their utility. One iteration consists of three phases: (1) the mobsim (mobility simulation) phase, where synthetic persons execute their daily plans; (2) the scoring phase, where executed plans are evaluated; and (3) the replanning phase, where a fraction of the synthetic population invents new plans to test during the next iteration.

Among the three phases outlined above, phases 2 and 3 are embarrassingly parallel problems[1]. In contrast, distributing the mobsim phase onto parallel computing hardware is more challenging, as synthetic persons interact while travelling within the simulation. Therefore, when developing a new parallel architecture for MATSim we focus on the mobsim phase, assuming that phase 2 and 3 can be integrated later.

The default implementation of the mobsim, QSim, already facilitates multicore systems using Java's concurrency primitives and a shared memory algorithm. Implemented more than a decade ago by Dobler and Axhausen [3], this approach effectively scales up to 8 processes. Executing the mobsim with more processes does not improve execution times, as noted by Graur, Bruno, Bischoff, *et al.* [4]. In response, their mobsim implementation, Hermes, is single threaded and focuses on optimizing cache locality as well as memory footprint at the cost of flexibility of what can be modelled. Their single threaded implementation already saturates the memory bus, which leads them to the conclusion that a parallel mobsim would not improve runtimes. Conversely, Strippgen [5] developed a mobsim implementation suitable for GPU (Graphics Processing Unit) hardware which supports massive parallelism, yet integration into the existing simulation framework proved to be challenging. Besides the MATSim ecosystem, other attempts for parallel traffic simulations were undertaken utilizing multicore hardware [6]–[8] and GPU accelerators [9]–[11].

Instead of improving execution times on a single machine, we propose a distributed message passing algorithm for the mobsim that facilitates scaling traffic simulations across multiple machines. This approach offers numerous advantages. Firstly, in contrast to programming hardware accelerators, it preserves the flexible programming model inherent to CPU-based programming. Secondly, by distributing the computa-

[1]https://en.wikipedia.org/wiki/Embarrassingly_parallel

tional load across multiple machines, we effectively prevent the saturation of the memory bus, as each machine manages only a portion of the traffic model. Implementing a distributed algorithm allows us to tap into the power of HPC (High-Performance Computing) clusters, which are typically composed of many interconnected machines. Moreover, with each process handling just a fraction of the traffic simulation, it should also become possible to improve cache locality, improving execution times on single multicore machine as well.

A distributed queue simulation was initially proposed by Nagel and Rickert [12] and Cetin, Burri, and Nagel [13] with promising results. More recently, Wan, Yin, Wang, *et al.* [14] implemented a distributed version of MATSim. However, their fastest execution times were observed with four processes achieving a speedup of two — less than what the current parallelization approach offers. Another recent development is Mobiliti independent of MATSim, introduced by Chan, Wang, Bachan, *et al.* [15], a discrete event simulation that focuses on optimistic synchronization methods, which scales very well on HPC infrastructure.

In this work, we describe the prototype of a distributed mobsim implementation. We apply the concept of parallelization from Cetin, Burri, and Nagel [13] to MATSim's default mobsim implementation, QSim, which, in contrast to Mobiliti, uses conservative synchronization. We believe this approach better fits the time step-based simulation model of MATSim. The prototype is tested on the HRN@ZIB HPC cluster, capable of scaling to several thousand processes. Based on the executed benchmarks, we discuss additional enhancements to take full advantage of modern distributed computing hardware.

II. Methodology

The proposed algorithm by Cetin, Burri, and Nagel [13] employs domain decomposition of the network graph to distribute the simulation workload across processes. Multiple processes may be executed in parallel on the same multicore machine, or each process can be run on a separate machine. In most HPC setups, a combination is used, with multiple machines executing several processes. Each process is responsible for one domain derived from the domain decomposition phase, performing a single threaded traffic simulation for the network segment within its domain. Vehicles crossing a domain boundary are transmitted as messages to the corresponding process. Likewise, information on available storage capacities is communicated to neighbor processes. Messages are exchanged between neighboring domains once per simulated time step, consolidating all vehicle crossings and capacity updates into a single message per neighboring process.

Cetin, Burri, and Nagel [13] find that the runtime of their algorithm is constrained by the latency of message exchanges, indicating that a new implementation should support low-latency networking hardware, such as Infiniband[2] or Omni-Path[3]. The conventional high-level abstraction to utilize this hardware is MPI (Message Passing Interface) [17] which Java — the programming language for MATSim — does not natively support. Attempts to run a Java setup with available libraries, such as Open MPI [18][19], were unsuccessful. Consequently, we opted to develop a prototype in Rust, which allows direct interfacing with the C implementation of Open MPI. The prototype implementation is available as open-source code on GitHub[4] and this paper is based on v0.1.0 [20].

This prototype can process standard MATSim input files, enabling the use of existing simulation scenarios. Domain decomposition is performed using the METIS library [21], with the option to assign node weights that reflect the anticipated computational load for each network node. METIS balances the node-weights across different graph partitions. For the presented results, the computational load for each network node is estimated by parsing all plans of the synthetic population and counting the number of vehicles crossing each node. This way, the computational load is balanced over the simulated day, but may vary for particular time steps in the simulation. As the domain decomposition is node-based, all links pointing toward a node are assigned to the same network partition as the node. Figure 1b shows the result of the domain decomposition for the traffic scenario used in section III, with 32 processes. It is evident that subdomains in the center of the traffic network tend to be smaller as more traffic — and consequently, more computational work — is anticipated.

The diagram in figure 1a illustrates how nodes and links of the traffic network are distributed across subdomains. In the example, the network is divided into four partitions, each assigned to corresponding processes. Process 0 has process 1 and 3 as direct neighbors. Their neighbor relationship is established by the shared links, displayed as thick dashed lines. To explain how links are shared across processes, we focus on the single link shared between process 0 and 1. Assuming that it points towards the node located on process 0, the link is attributed to this process and its representation — including the vehicle queue — is located on process 0. Process 1 maintains a mirrored representation of this link, which tracks the storage capacities of the full representation on process 0 and serves as a buffer for vehicles entering the link, before they are transmitted to process 0.

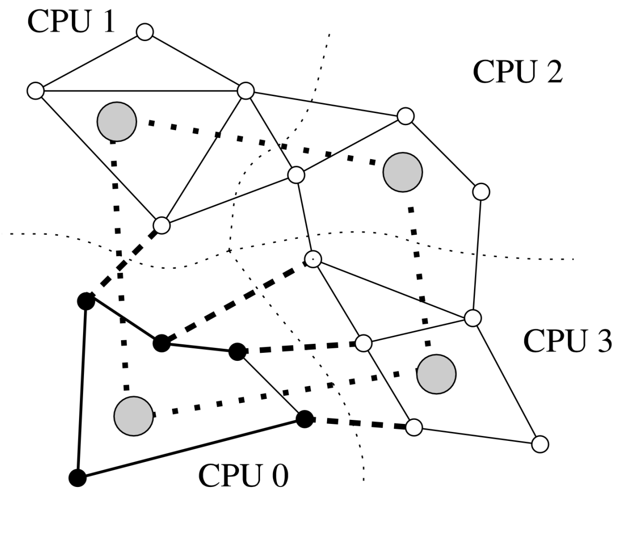
After partitioning the network graph, each process loads its portion of the network along with all synthetic persons performing their first activity within its domain. The traffic simulation executed in each process mirrors the current QSim implementation, as described in Horni, Nagel, and Axhausen [1, ch. 1]. The isolation of processes — interacting only through message exchanges — simplifies the implementation by eliminating the need for managing parallel access to data structures. The simulation work is executed on each process using the following phases:

1) **Activities:** Synthetic persons performing activities are stored in a priority queue, ordered by the end time of

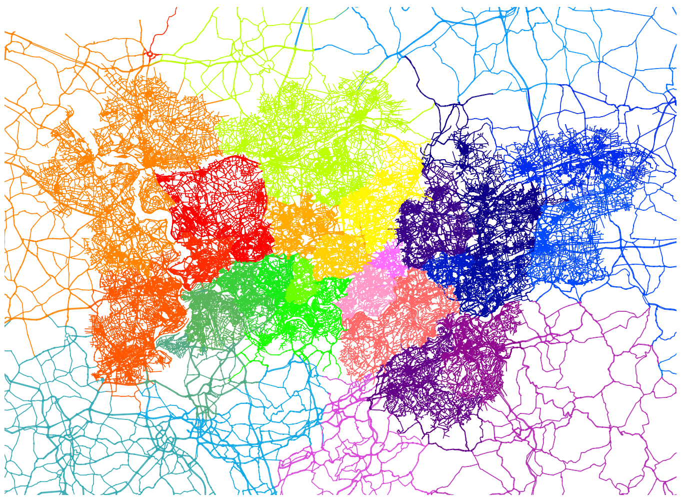
[2]https://en.wikipedia.org/wiki/InfiniBand

[3]https://en.wikipedia.org/wiki/Omni-Path

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



(a)



(b)

Fig. 1: (a) General concept of domain decomposition. Small circles are nodes in traffic network, large circles represent a process simulating a domain, lines are links in the road network, thick dashed lines are shared links between processes, thin dotted lines are domain boundaries, thick dotted lines represent communication channels between processes [16]; (b) Section of the network split into 32 domains, which is used to conduct benchmarks explained in section III

their activities. Those who have reached the end time of an activity are removed from the queue and start the next plan element, which involves either a teleportation leg or a leg performed on the simulated network.

2) **Teleport:** Persons who finish their teleportation leg start the next plan element and are placed into the activity queue (see [22, ch. 12.3.1, 12.3.2] for the concept of teleported legs).
3) **Nodes:** This step involves iterating over all nodes of the network partition, moving vehicles that have reached the end of a link onto the next link in the vehicle's route.
4) **Links:** The movement of vehicles on the network is constrained by the flow and storage capacities of links. During this step, the bookkeeping of these capacities is updated for the next time step. Additionally, vehicles on mirror links that must be transferred to another process, as well as changes to the storage capacities that must be sent to mirroring links on other processes, are collected.
5) **Send:** The collected information is sent to the corresponding neighbor domains in a single message, ensuring that only one message per neighbor domain is sent per time step. This step includes the serialization of messages into the wire format and the issuance of a non-blocking `ISend` call to the underlying MPI implementation.
6) **Receive:** A blocking `Recv` call to the underlying MPI implementation is issued for each direct neighbor domain. This step accounts for wait times due to workload imbalances and the time required to transmit messages over the communication network.
7) **Handle:** Received messages are de-serialized from wire format back into the data structures used in the computation.

Phases 1 to 4 can be summarized as simulation work, while phases 5 to 7 relate to inter-process communication. The main output of the mobsim is an events log that captures events such as activities starting and ending or vehicles entering and leaving network links. During the execution of the mobsim, each process generates the same events as the current MATSim implementation and writes them into an events file. This ensures that, once the files are merged, the results of the simulation run can be compared to the original mobsim implementation.

III. Results

The prototype implementation is tested using the existing MATSim Metropole Ruhr scenario [23], which includes a synthetic population of 491,175 persons for the 10% sample and a detailed traffic network of 547,011 nodes and 1,193,056 links. This scenario simulates 36 hours (129,600s) of simulation time. Of the four modes covered, car and bicycle trips are executed as network modes, while ride and walk are simulated as teleported modes (see [22, ch. 12.3.1, 12.3.2]). Synthetic persons using pt (public transit) in the original scenario are excluded from the test setup as the prototype does not cover pt simulation. Both the current QSim and the prototype are run for a single iteration of the 10% sample, across various numbers of processes, and additionally, a 1% sample is tested with the prototype. These tests are conducted on an HPC cluster equipped with Intel Xeon Platinum 9242 processors, each offering 48 CPU-cores, and an OmniPath 100 networking infrastructure.

Figure 2 presents the performance outcomes. The RTR (Real-Time Ratio), indicating the ratio between simulated time and wall clock time required for the simulation, is used as a measure. The 10% sample using the current QSim achieves an RTR of 82 on a single process and peaks at 202 with 24 processes, yielding a 2.5x speedup. Notably, the speedup plateaus beyond 6 processes, already achieving an RTR of 191. In contrast, the 10% sample simulated with the prototype reaches an RTR of 586 for a single process and a peak of 20,662 for 1024 processes, resulting in a 35x speedup compared to the single-process setup and a 102x speedup relative to the current QSim. The runs with up to 32 processes utilize a single physical machine, while runs with 64 or more processes span multiple machines, with process communication facilitated by the networking hardware.

The RTR for the smaller 1% sample is about 10x faster on a single process than the 10% sample. The optimal RTR for this setup was achieved with 64 processes, after which it levels off. It is notable that both the 1% and 10% samples simulated with the prototype achieve RTRs of similar magnitudes. Additionally, the 1% sample approaches the RTR of the dry run, described next.

To estimate the theoretical maximum performance, a dry run is included in the analysis (see III-B) with the fastest execution time achieved on a single process, as no inter-process communication impedes the execution. The RTR decreases with more processes and stabilizes around 31,000.

For more than 2048 processes, the RTRs observed in Figure 2 significantly drop below peak performance for both the 1% and 10% scenarios. As both scenarios directly write events files for the executed mobsim onto a shared network drive, and given that the dry run maintains a constant RTR for up to 16,384 processes, we suspect that the network connection to the storage becomes saturated with write requests from too many processes simultaneously.

A. Simulation Scenario

To understand where time is spent during the simulation, we conduct an analysis of the individual algorithm phases. Figure 3 presents the average durations of each phase during the execution of one simulation time step. Blue colors represent simulation work from phase 1 to 4 as detailed in Section II, while yellow colors depict execution times related to inter-process communication, covering phases 5 to 7 from the same section. In setups with only one process, the entire time is dedicated to actual simulation work. The bulk of this work is consumed by the 'nodes and links' logic, which drives the traffic simulation on the network. Starting with two processes, the prototype also allocates time to communication, which includes waiting for neighboring processes that may be slower in executing their share of the traffic simulation, as well as the time required to transfer messages over the network. As the simulation scales to more processes, the average time spent on simulation work decreases, eventually approaching timings

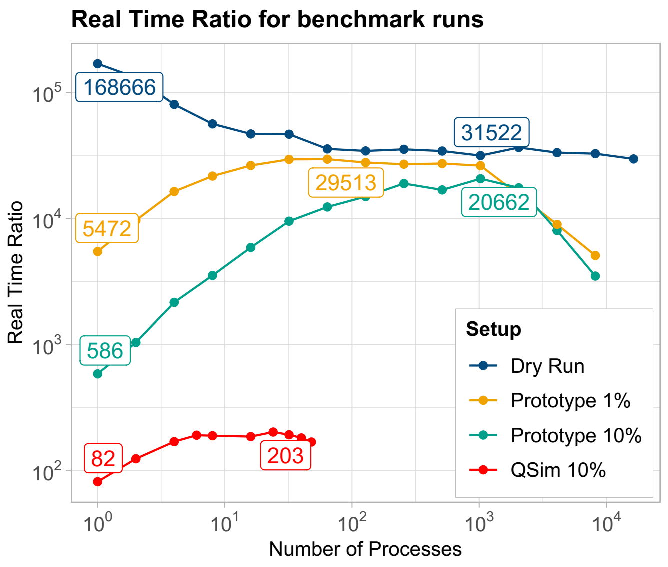


Fig. 2: RTR of different benchmark runs. RTR was used since results indicate that it tends to saturate at the same level for different scenario sizes on the same hardware, in contrast to speed-up, which depends on scenario sizes.

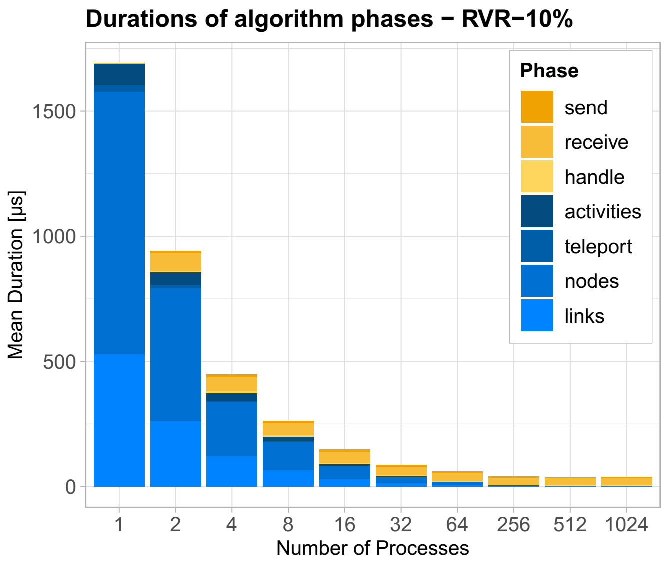


Fig. 3: Average durations for performing one time step in the simulation, distinguished by simulation work (blue) and message exchange (yellow) for the dry run

similar to those in the dry run — less than one microsecond per time step on average (see Figure 5). Conversely, the time spent on communication remains relatively stable as the number of processes increases, averaging around 26 microseconds per time step. Figure 3 illustrates that in setups with a large number of processes, the time spent on communication largely determines the overall execution times, explaining the plateauing RTR observed in Figure 2.

Investigating the computation time spent over the simulated day, as depicted in Figure 4, it becomes apparent that load balancing of computational work is less important when the scenario is divided into sufficiently many domains. The plot illustrates the difference between the fastest and slowest execution of simulation work in blue and the maximum durations dedicated to communication across all processes in yellow. Timings measured are averaged over 30 simulation time steps to reduce the volume of tracing data written to disk. For the setup with 16 processes, the differences between the slowest and fastest processes align with the traffic volumes simulated. Additionally, the maximum durations for communication are similar to the differences between fastest and slowest execution times for simulation work, suggesting that communication time is largely influenced by load imbalances. Specifically, the process that finishes simulation work first must wait in the receive phase (see phase 6) until the slowest process completes. Only then messages can be exchanged. In contrast, in the 1024-process setup, only marginal differences in execution times for simulation work can be observed at any point during the simulation. However, the durations for the communication phase significantly higher compared to the differences in simulation work durations, indicating that communication times in this setup are predominantly driven by the time necessary for message exchange in the communication layer. The 64-process setup exhibits a combination of both effects, showing variations in the time required to process a simulation step, and communication times being influenced by both, imbalances in simulation work and the time required to exchange messages over the network.

B. Dry run

The distributed mobsim implementation's performance primarily hinges on two factors: the computation time for the traffic simulation and the time required for inter-process communication. To analyze the issues with message exchange specifically, we conduct a dry run using the standard input data, but without loading any plans. This setup performs simulation steps without actual computational work, allowing us to estimate the lower bound for communication timings, as processes exchange empty messages at each simulation time step.

Figure 5, similar to figure 3, presents the average durations for a simulation time step, categorized by the number of processes used. With no real simulation tasks performed, the operation time for the simulation's framework remains below one μs per time step, regardless of the number of processes. However, communication times exhibit substantial variability, with durations for sending and handling messages staying consistent across various process counts, except when using just two processes. The time taken to receive messages increases with the number of processes up to 8 and then stabilizes at approximately 7μs for up to 32 processes. Beyond 64 processes, this synchronization time levels off around 15μs.

Setups with up to 32 processes are executed on a single machine, and the variable durations for communication are directly correlated to the maximum number of neighbors any process manages. More neighbors mean more messages to process, leading to longer execution times for communication. As all other processes must wait for the process with the highest number of neighbors, the communication time is determined by the maximum number of neighbors any process in the simulation manages. Dividing the average duration for the receive phase by the maximum number of neighbors yields 1μs on single machine setups and 1.5μs on multi machine setups for one message exchange. The OmniPath 100 specification notes latencies of 0.7μs for network requests. As one message exchange in our implementation consists of two MPI-network requests for `MProbe` and `MRecv`, our timings show that we achieve optimal latency values on the hardware utilized.

IV. Discussion

Twenty years ago, Cetin, Burri, and Nagel [13] speculated that the runtime of their proposed algorithm is bounded by the latencies in network communication, but they were unable to measure this hypothesis for low latency networking hardware. With more computing power at hand, we can show that the execution times for a distributed simulation are in fact latency bound even on high-performance networking hardware like OmniPath 100. As shown in figures 3 and 4 the time spent computing the traffic simulation becomes negligible while the time necessary for inter process communication remains stable for large number of processes.

The presented results demonstrate that switching from Java to a native language improves performance on a single machine by a factor of 7. Considering, that the prototype does not include all features implemented in the original QSim implementation, this speedup is comparable to what is achieved with Hermes [4]. Notably, no specific focus was placed on performance optimization for a single machine in the prototype, which uses complex data structures including pointers, vectors, hashmaps, and strings. This speedup can likely be attributed to the more compact memory layout and improved memory locality offered by structs in Rust compared to objects in Java.

Our findings further reveal that when the simulation domain is partitioned into multiple smaller segments, each process is tasked with only a small portion of the overall simulation workload. This decentralization significantly reduces the amount of data each process needs to handle, potentially relieving pressure on the memory bus, which we suspect to be a major bottleneck in the current QSim implementation. Such a reduction in data transfer between memory and CPU suggests that also a Java-based implementation of our distributed algorithm could achieve RTRs comparable to those observed with the Rust implementation.

Additionally, the results suggest that transitioning the mobsim from a shared memory to a message passing algorithm benefits simulation setups run on a single multicore machine, as well as large-scale traffic simulations in a distributed HPC environment. Currently, our scenarios do not fully utilize the capabilities of modern HPC clusters like NHR@ZIB, which provides over 100,000 computing cores. The performance

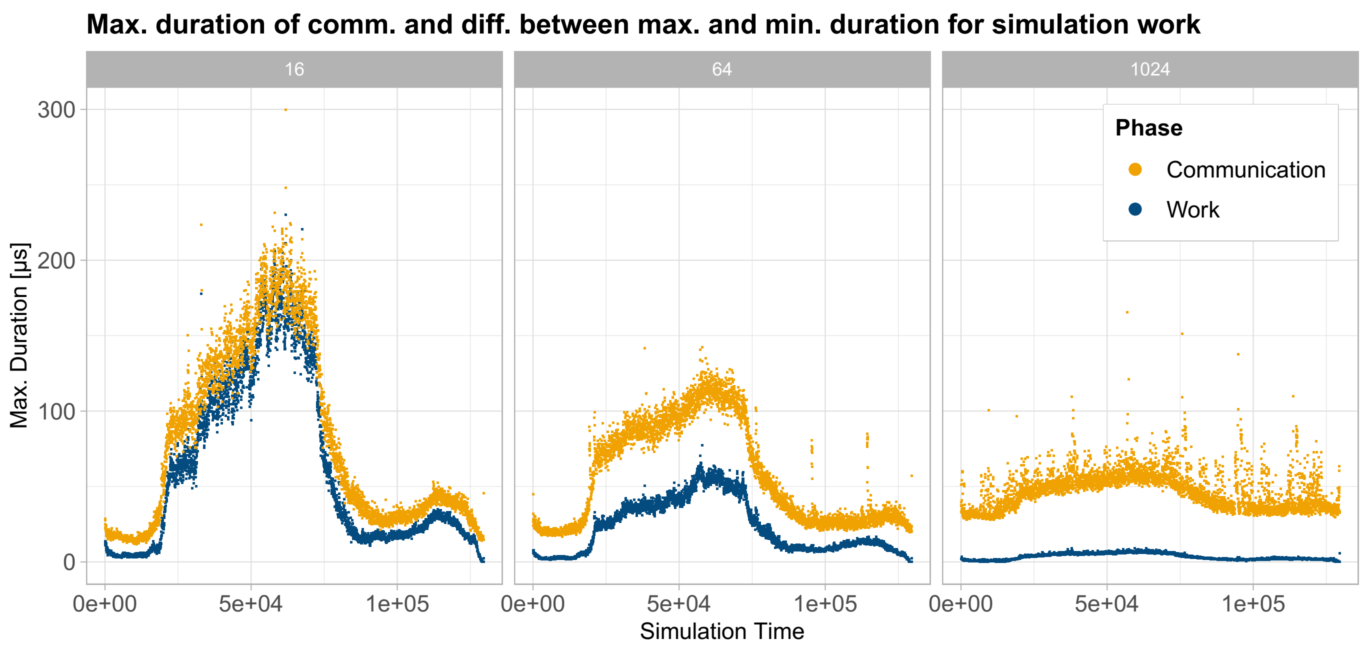


Fig. 4: Differences in fastest and slowest duration to execute simulation work of one time step across all processes (blue), max. duration to perform communication across all processes (yellow). Each data point is the average of 30 simulation steps.

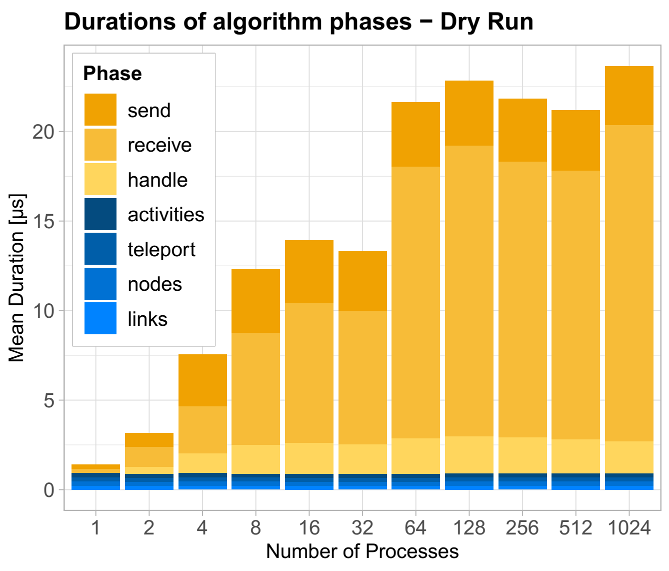


Fig. 5: Average durations for performing one time step in the simulation, distinguished by simulation work (blue) and message exchange (yellow) for the dry run

of our tested scenario peaks at 1024 processes. To better leverage available hardware, two strategies are recommended: (1) increase the amount of computation, and (2) reduce the time necessary for inter process communication.

Increasing the computation is straightforward; the proposed architecture allows for running larger scenarios, potentially encompassing entire countries rather than just regions, without degrading RTR. Reducing the time spent on inter-process communication poses more challenges, but there are several viable approaches:

- As indicated in III-B, the communication time is primarily determined by the process with the highest number of neighbors. In our test runs involving large numbers of processes, the maximum number of neighboring domains reached up to 10, while the average remained around 5. By optimizing domain decomposition to reduce the maximum number of neighbors, we could decrease the number of exchanged messages, thereby reducing communication times. The objective of this optimization is to bring the maximum number of neighbors closer to the average value.
- The current architecture requires synchronization of neighbor processes at every simulated time step. By adapting the synchronization to account for the time vehicles travel across links, as well as *backwards travelling holes* as suggested by Charypar, Axhausen, and Nagel [24], the number of messages exchanged could be reduced.

For the remaining inter-process communication, improvements can be implemented based on insights from Ghosh, Halappanavar, Kalyanaraman, *et al.* [25] and Gropp, Hoefler, Thakur, *et al.* [26, ch. 2 and 3]:

- Currently, our point-to-point communication involves three MPI-calls: `ISend`, `MProbe`, and `MRecv`, necessitating two network requests per message exchange. By adopting non-blocking counterparts for `MProbe` and `MRecv`, we could alleviate the constraints on the order of execution, though the total number of network requests would remain unchanged.

- Transitioning to a higher level of abstraction for inter-process communication could also be beneficial. MPI v3 introduces collective communication primitives for distributed graph topologies, which could optimize the physical locality of adjacent simulation domains. This could be particularly effective using the `neighbor_alltoallv` function.
- MPI 3's one-sided communication infrastructure allows asynchronous access to memory sections of remote processes through MPI windows, reducing the synchronization required for data exchange. However, this method requires careful management of memory offsets, making it more complex to implement.

Ghosh, Halappanavar, Kalyanaraman, *et al.* [25] found that using neighbor collectives and one-sided communication can yield speedups of 1.5 to 6 times compared to point-to-point communication. Combining architectural improvements with enhanced process synchronization could potentially multiply the speedups observed in the benchmark.

The RTR presented in Figure 2 indicates a performance drop for large numbers of processes, likely due to I/O (Input/Output) performed during the simulation. Maintaining high performance with many processes necessitates a robust I/O strategy. Instead of frequent I/O operations on a network drive, local SSDs could be utilized, with output data transferred to the final location at once. Alternatively, distributed I/O solutions, like MPI-I/O [26, ch. 7], could be explored.

V. Conclusion

We have successfully developed a prototype of a distributed mobsim, demonstrating the feasibility of extending the MATSim framework into a distributed traffic simulation. Our prototype outperforms the existing QSim implementation, achieving a speedup of 100 in a real-world scenario. With an RTR of 20,000, it's now possible to simulate an entire day in just 4.3 seconds. Profiling indicates that with a sufficient number of processes, the computational time for traffic simulation becomes negligible, with the bulk of runtime spent on inter process communication. While this limits further speedups under the implemented communication strategy, it enables the execution of significantly larger simulation scenarios with similar RTRs. Nonetheless, we still see room for improvement by reducing the time spent on inter-process communication, as well as in developing a comprehensive I/O strategy to ensure the scalability for even more processes.

The results of the conducted benchmark indicate that optimizing the domain decomposition for minimal edge cuts or computational load on each partition is less critical compared to reducing the maximum number of neighbors for all partitions of a simulation setup, as the computation time is dominated by the number of message exchanges, which is related to the maximum number of neighbors.

Transitioning from a shared memory to a message-passing algorithm benefits simulation setups run on single machines as well. Unlike the current mobsim implementation, which achieves a 2.5x speedup through parallelization, the prototype implementation scales well up to 32 processes, on a single machine, with a speedup of 20 compared to a simulation executed on a single core.

The achieved speedups suggest that a distributed mobsim implementation should be integrated into the existing MATSim framework. To preserve the comprehensive functionality of the current framework without the need for extensive redevelopment, integration must be compatible with the JVM (Java Virtual Machine) ecosystem and capable of leveraging high-performance networking hardware. Infinileap [27] and hadroNIO [28] are promising candidates for utilizing high-performance networking within the JVM. As the architecture of MATSim will change fundamentally with the introduction of a distributed mobsim, we must consider how to incorporate other parts of the framework into the architecture. One of the main challenges will be the propagation of mobsim events to other modules, such as the routing and scoring module in a distributed computing environment, which is currently investigated.

VI. Acknowledgment

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. Horni, K. Nagel, and K. W. Axhausen, *The Multi-Agent Transport Simulation Matsim*, en. Ubiquity Press, Jul. 2016, ISBN: 9781909188754. DOI: 10.5334/baw.

[2] C. E. Leiserson, N. C. Thompson, J. S. Emer, *et al.*, "There's plenty of room at the top: What will drive computer performance after moore's law?" en, *Science*, vol. 368, no. 6495, Jun. 2020, ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.aam9744.

[3] C. Dobler and K. W. Axhausen, *Design and implementation of a parallel queue-based traffic flow simulation*, en, 2011. DOI: 10.3929/ETHZ-B-000040273.

[4] D. Graur, R. Bruno, J. Bischoff, *et al.*, "Hermes: Enabling efficient large-scale simulation in MATSim," *Procedia Comput. Sci.*, vol. 184, pp. 635–641, Jan. 2021, ISSN: 1877-0509. DOI: 10.1016/j.procs.2021.03.079.

[5] D. Strippgen, "Investigating the technical possibilities of real-time interaction with simulations of mobile intelligent particles," en, Ph.D. dissertation, Technische Universität Berlin, Oct. 2009. DOI: 10.14279/depositonce-2272.

[6] Q. Bragard, A. Ventresque, and L. Murphy, "Self-Balancing decentralized distributed platform for urban traffic simulation," *IEEE Trans. Intell. Transp. Syst.*, vol. 18, no. 5, pp. 1190–1197, May 2017, ISSN: 1524-9050, 1558-0016. DOI: 10.1109/TITS.2016.2603171.

[7] Y. Qu and X. Zhou, "Large-scale dynamic transportation network simulation: A space-time-event parallel computing approach," *Transp. Res. Part C: Emerg. Technol.*, vol. 75, pp. 1–16, Feb. 2017, ISSN: 0968-090X. DOI: 10.1016/j.trc.2016.12.003.

[8] Z. Gong, W. Tang, D. A. Bennett, and J.-C. Thill, "Parallel agent-based simulation of individual-level spatial interactions within a multicore computing environment," *Int. J. Geogr. Inf. Sci.*, vol. 27, no. 6, pp. 1152–1170, Jun. 2013, ISSN: 1365-8816. DOI: 10.1080/13658816.2012.741240.

[9] J. Xiao, P. Andelfinger, D. Eckhoff, W. Cai, and A. Knoll, "A survey on agent-based simulation using hardware accelerators," *ACM Comput. Surv.*, vol. 51, no. 6, pp. 1–35, Jan. 2019, ISSN: 0360-0300. DOI: 10.1145/3291048.

[10] H. Zhou, J. L. Dorsman, M. Snelder, E. d. Romph, and M. Mandjes, "GPU-based parallel computing for activity-based travel demand models," *Procedia Comput. Sci.*, vol. 151, pp. 726–732, Jan. 2019, ISSN: 1877-0509. DOI: 10.1016/j.procs.2019.04.097.

[11] A. Saprykin, N. Chokani, and R. S. Abhari, "GEMSim: A GPU-accelerated multi-modal mobility simulator for large-scale scenarios," *Simulation Modelling Practice and Theory*, vol. 94, pp. 199–214, Jul. 2019, ISSN: 1569-190X. DOI: 10.1016/j.simpat.2019.03.002.

[12] K. Nagel and M. Rickert, "Parallel implementation of the TRANSIMS micro-simulation," *Parallel Comput.*, vol. 27, no. 12, pp. 1611–1639, Nov. 2001, ISSN: 0167-8191. DOI: 10.1016/S0167-8191(01)00106-5.

[13] N. Cetin, A. Burri, and K. Nagel, "A large-scale agent-based traffic microsimulation based on queue model," in *IN PROCEEDINGS OF SWISS TRANSPORT RESEARCH CONFERENCE (STRC), MONTE VERITA, CH*, 2003.

[14] L. Wan, G. Yin, J. Wang, G. Ben-Dor, A. Ogulenko, and Z. Huang, "PATRIC: A high performance parallel urban transport simulation framework based on traffic clustering," *Simulation Modelling Practice and Theory*, vol. 126, p. 102 775, Jul. 2023, ISSN: 1569-190X. DOI: 10.1016/j.simpat.2023.102775.

[15] 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, ISBN: 9781728103235, 9781728103211. DOI: 10.1109/ITSC.2018.8569397.

[16] K. Nagel, P. Wagner, and R. Woesler, "Still flowing: Approaches to traffic flow and traffic jam modeling," *Oper. Res.*, vol. 51, no. 5, pp. 681–710, Oct. 2003, ISSN: 0030-364X. DOI: 10.1287/opre.51.5.681.16755.

[17] Message Passing Interface Forum, *MPI: A Message-Passing interface standard version 4.1*, Nov. 2023.

[18] O. Vega-Gisbert, J. E. Roman, and J. M. Squyres, "Design and implementation of java bindings in open MPI," *Parallel Comput.*, vol. 59, pp. 1–20, Nov. 2016, ISSN: 0167-8191. DOI: 10.1016/j.parco.2016.08.004.

[19] E. Gabriel, G. E. Fagg, G. Bosilca, *et al.*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Springer Berlin Heidelberg, 2004, pp. 97–104. DOI: 10.1007/978-3-540-30218-6_19.

[20] J. Laudan and P. Heinrich, *Parallel qsim rust*, Apr. 2024. DOI: 10.5281/zenodo.10960723.

[21] G. Karypis and V. Kumar, "Multilevelk-way partitioning scheme for irregular graphs," *J. Parallel Distrib. Comput.*, vol. 48, no. 1, pp. 96–129, Jan. 1998, ISSN: 0743-7315. DOI: 10.1006/jpdc.1997.1404.

[22] A. Horni, K. Nagel, and K. Axhausen, *MATSim user guide*, Technische Universität Berlin, Mar. 2024.

[23] C. Rakow, K. Nagel, G. Leich, *et al.*, *MATSim metropole ruhr v1.4.1-parallel-benchmark*, Apr. 2024. DOI: 10.5281/zenodo.10959019.

[24] D. Charypar, K. W. Axhausen, and K. Nagel, "Event-Driven Queue-Based traffic flow microsimulation," *Transp. Res. Rec.*, vol. 2003, no. 1, pp. 35–40, Jan. 2007, ISSN: 0361-1981. DOI: 10.3141/2003-05.

[25] S. Ghosh, M. Halappanavar, A. Kalyanaraman, A. Khan, and A. H. Gebremedhin, "Exploring MPI communication models for graph applications using graph matching as a case study," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, May 2019, pp. 761–770, ISBN: 9781728112466, 9781728112473. DOI: 10.1109/IPDPS.2019.00085.

[26] W. Gropp, T. Hoefler, R. Thakur, and E. Lusk, *Using Advanced MPI: Modern Features of the Message-Passing Interface*, en. MIT Press, Nov. 2014, ISBN: 9780262527637.

[27] F. Krakowski, F. Ruhland, and M. Schöttner, "Infinileap: Modern High-Performance networking for distributed java applications based on RDMA," in *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*, IEEE, Dec. 2021, pp. 652–659, ISBN: 9781665408783, 9781665408790. DOI: 10.1109/ICPADS53394.2021.00087.

[28] F. Ruhland, F. Krakowski, and M. Schöttner, "hadroNIO: Accelerating java NIO via UCX," in *2021 20th International Symposium on Parallel and Distributed Computing (ISPDC)*, IEEE, Jul. 2021, pp. 25–32, ISBN: 9781665432818, 9781665432825. DOI: 10.1109/ISPDC52870.2021.9521601.