Design and implementation of a parallel queue-based traffic flow simulation

Working Paper

Author(s):
Dobler, Christoph; Axhausen, Kay W.

Publication date:
2011

Permanent link:
https://doi.org/10.3929/ethz-b-000040273

Originally published in:
Arbeitsberichte Verkehrs- und Raumplanung 732

1 **Design and Implementation of a Parallel Queue-Based Traffic Flow Simulation**

2 Date of submission: 2010-jul-28

3 Christoph Dobler
IVT, ETH Zurich, CH-8093 Zurich
phone: +41-44-633 65 29
fax: +41-44-633 10 57
dobler@ivt.baug.ethz.ch

4 Kay W. Axhausen
IVT, ETH Zurich, CH-8093 Zurich
phone: +41-44-633 39 43
fax: +41-44-633 10 57
axhausen@ivt.baug.ethz.ch

5 Words: 6032, Figures: 5

ABSTRACT

Today, agent based micro-simulations are widely used in the field of transport planning and traffic management. One important requirement is the ability to simulate large scale scenarios in reasonable time. An obvious approach to reduce the computation time of such scenarios is to use multiple CPU cores.

This paper presents the implementation of a parallel queue simulation for MATSim written in Java. Existing parallel traffic micro-simulations are reviewed concerning their parallelization approaches as well as the reached performance gains. Various concepts how to model the progress of time and how to distribute computational workload among multiple CPU cores are discussed. Based on an analysis of the MATSim framework regarding its structure, performance and extensibility the concepts for the parallel queue simulation are selected and implemented. Performance tests with different sized scenarios are conducted. An analysis of the results shows that especially for large scale scenarios a significant performance gain is reachable.

INTRODUCTION AND RELATED WORK

Today, agent based micro-simulations are widely used in the field of transport planning and traffic management. One important requirement is the ability to simulate large scale scenarios in reasonable time. Until the end of the last millennium, the main focus in CPU development was to increase the computing power of a single CPU core. As a result, a simulation could be simply sped up by using a faster CPU.

Within the last years, the development focus has changed dramatically. Today, it can be assumed that in the near future computers with multi core CPUs will become state of the art. Increasing the computing power of a CPU is mainly based on the usage of multiple cores where each core for itself will not have a significantly better performance than an old single core CPU.

As a result, existing program code has to be adapted to be able to benefit from this new multi-core architecture. Typically, this makes considerable changes in the program structure necessary because the program logic has to be switched from sequential to parallel. This paper presents the implementation of a parallel queue simulation which results in a major speedup and therefore reduces the simulation time of large scale scenarios significantly.

Here, existing work in the field parallel transport simulations is determined. This includes an overview of existing parallel simulations tools as well as the techniques which they use for the parallelization. Additionally commonly used approaches to model the progress of time within a simulation are described and analyzed regarding their suitability for a parallel implementation. Moreover it is discussed how the computation effort of a simulation can be distributed among multiple CPU cores. Subsequently, MATSim, a framework for iterative, agent-based micro-simulations, is described with a special focus on its simulation modules. Based on the findings from the previous sections, in the implementation section the selection and implementation of a parallelization approach is described. Afterwards the performance of the implementation is measured and evaluated using various real world scenarios. The paper closes with some conclusions and the outlook on further work.

Parallel Traffic Flow Micro-Simulations

In this section we present a selection of previous work related to parallel traffic flow micro-simulations. An overview on traffic flow simulations in general is for example given by *(1)*. *(2, 3, 4, 5)* give a detailed overview on the topics of (multi-)agent-systems and simulations.

Various existing micro-simulations have been ported to parallel computers. As described by *(6)* AIMSUN2 uses a shared memory approach based on a parallel threads. Each of these threads is a sequence of instructions executed within the context of a process. If a process hosts multiple threads they can access the same data at the same time which may lead to inconsistencies or deadlocks. Therefore, it has to be ensured that changes on the data are only allowed by one thread at a time. The distribution of the calculation effort is done by introducing a system with so called blocks and layers. A block contains objects which interact with each other in a simulation step. A layer groups blocks which do not influence each other and therefore can be simulated simultaneously. By using 8 parallel threads, they reach performance gains up to a factor of 3.5.

A mesoscopic traffic simulation model is implemented by DYNEMO *(7)*. A parallel implementation for distributed multiprocessor systems with distributed memory has been developed. The parallelization is based on the usage of subnetworks which are created by splitting the network along intersections. As a result the split intersections are duplicated and exist in multiple subnetworks. To simulate traffic between the subnetworks so called transit-storage links are introduced. They accumulate cars which want to proceed to links belonging to another subnetwork. After each simulation step the subnetworks exchange the cars on those transit-storage links. A speed-up of factor 15 using 19 processors is reported.

The parallel implementation of TRANSIMS uses a cellular automata approach *(*8*)*. The calculation effort is distributed among parallel distributed processors by splitting the network into domains. The cuts are performed in the middle of links. Each of the divided links is fully represented in both domains. The consistency between different processors is maintained by exchanging information about the divided links. TRANSIMS uses an iterative simulation approach to do adaptive load balancing. During each iteration the calculation times for all intersections and links are measured. By using this data the load balance is optimized from iteration to iteration.

An event-driven parallel queue-based micro-simulation for MATSim is introduced by Charypar *et al.* *(*17*)*. In contrast to the other described parallel simulations it can be run on shared memory computers utilizing multiple CPUs. The workload is distributed by an adaptive domain decomposition approach. A small test scenario is sped up by a factor of 53 when using 64 CPU cores.

There is a multiplicity of other parallel agent-based traffic micro-simulations that are not discussed here because they employ similar concepts *(*e.g. 9*,* 10*)*.

As can be seen, a major part of those parallel micro-simulations use the concept of distributed computation. Distributed systems consist of multiple computers which are loosely coupled—e.g. through a computer network—where interactions between the computers are relatively slow. Parallel computing in contrast means parallel execution of calculations on multi-processor (and / or multi-core) computing platforms. Interactions between different processors are significantly faster than on distributed computers *(*11*)*.

When most of those micro-simulations were written distributed computation was a commonly used technology. Multi-processor systems were expensive and possible scenarios sizes limited by the available amount of memory. However, today the situations has changed dramatically. Even typical workstations use multi-core CPUs and several GB of memory. Therefore parallel computing has gained an enormous amount of attractiveness. Especially large scale scenarios—as they are frequently used today—may profit from such a paradigm shift. E.g. applying a domain decomposition approach to a high resolution network will create a huge amount of shared links and / or nodes which again will result in many interactions between different processors. While those interactions can be handled in reasonable time by a parallel computing implementation, they may significantly slow down an approach based on distributed computing.

Modelling the Progress of Time

A common criterion to classify simulations is grouping them by the way they model the progress of time. The two mainly used approaches in the field of traffic flow simulations are time step based and event based.

A simple method to model the progress of time is to divide the simulated period into equal sized time slices (*time slice*, *time bin* and *time step* are used synonymous in this context). For each of these time slices the state of the simulated system has to be evaluated—which is one major drawback of this approach. Even if nothing happens between two time steps—

and, therefore, the system state does not change—the state of the system has to be calculated. Another problem is determining the size of the time steps. On the one hand, using too short time slices results in unnecessary long calculation times. Too long time bins, on the other hand, may lead to poor or even wrong simulation results. In many simulated systems, the number of events occurring during a time step varies significantly. Thus, it is necessary to choose the time step size according to the peak times.

Think of a road where on average every 60 seconds a car is driving along. During the peak hour, significantly more vehicles may pass that road, e.g. one every 10 seconds. Having a time step of 10 seconds seems to be appropriate when looking at the average flow rate but clearly is too large with respect to the rate during the peak hour. One obvious solution for this problem is to adapt the size of the time slices during the simulation, which can be done dynamically depending on the results of previous time steps or based on predefined rules resulting from existing knowledge (e.g. the peak hours of the call center are known). However, a problem that cannot be solved by adapting the size of the time bins is load balancing. Again, this can be illustrated with the simulation of roads. If not a single road but an entire road network is simulated, it is obvious that the traffic flow rate differs depending on the location of a road. As a result, the time step size has to be small during the whole simulation, which again results in a high computational effort.

Typically, simulation software based on a time step approach can be parallelized quite simply. The main requirement is that the events which occur within a time step can be separated into groups that are independent from each other. In the road network example, this could be e.g. a group for every road in the network. In a parallel implementation, each of those groups could be handled by a separate simulation thread which synchronizes its data with the other threads at the end of each time step.

Another even more intuitive possibility to simulate time is an event driven approach. In a discrete-event simulation, the operations within a simulated system are represented as a chronologically ordered list of events. Each event occurs at a given point in time and causes a change of the system state *(*12*)*. Using again a road as example, every car entering or leaving a road would create such an event.

Classic event driven simulation modules use internally a list of events which have to be processed at their scheduled future point in time *(*13*,* 14*)*. During the simulation the events are processed in chronological order—when the list is empty the simulation ends. The scheduled events can be predefined before the simulation starts and / or be created during a running simulation. In a distributed event driven simulation employing multiple threads leads to a situation where each simulation thread uses its own simulation clock. This clock is not linked to the clocks of the other threads. Combined with the different calculation efforts of the threads, this results in varying current simulation times. Thus, situations will occur where the events are not processed in a chronological order anymore.

Several solutions to solve this problem have been proposed which can be divided into optimistic and conservative approaches *(*e.g. 13*,* 15*,* 14*,* 11*,* 16*)*. Optimistic approaches assume that such timing problems will not occur very often. Therefore, the threads can process the events without checking whether other events should be handled before. However, if a timing problem appears, a roll back procedure has to be performed which turns the multiple simulation clocks back to the point in time where the events can be processed in the correct order. Another possible solution is a conservative approach where each thread has to check whether its simulation time can be advanced or not. Doing so ensures on one hand that no roll back procedures have to be performed but on the other hand causes additional calculation effort for the consistency checks.

Depending on the simulated problem the one or the other approach performs better. However, experiments with real world scenarios in the field of traffic flow simulations show that event driven approaches tend to perform better than time step based ones. In such scenarios the traffic volumes and their distribution in the network varies significantly in space and time which results in a large computational overhead for time step based simulations *(*17*)*.

Distribution of the Workload

The computational power of parallel computers can be utilized by one of two fundamentally different approaches. On the one hand, new simulation software can be developed which incorporates algorithms that are designed to be run on parallel computers. On the other hand, existing software can be adapted to be able to do parallel simulations *(*e.g. 18*)*. Regardless of the approach pursued a parallel simulation has to split up the total computational effort into small packages which can be handled by the parallel executed modules of the simulation.

In a *functional decomposition*, the tasks that have to be performed are assigned to different simulation modules. In a traffic flow simulation, one module could do the routing while another one could execute the movement of the vehicles. Such a decomposition is often easy to implement but the achievable speed-up is limited by the number of tasks that can be performed simultaneously *(*8*)*.

Another approach commonly used in the field of parallel traffic flow simulation *(*e.g. 9*,* 10*,* 8*,* 7*)* is *domain decomposition*. The aim is to divide the simulation problem into pieces with approximately equal computational effort. Each of those pieces is handled by one CPU core. Such an approach performs best if the domains do not interact with each other. Then, almost linear performance gains can be realized. However, in typical traffic simulation interactions between domains occur quite frequently. Thus, their influence on the overall performance cannot be ignored. Each time such interactions take place a certain amount of calculation overhead—overhead in this context are calculations that would not be necessary in a non-parallel simulation—is created. Depending on that overhead the reachable performance gain is limited.

Another factor that can significantly influence performance is the load balance between the domains. The domain with the highest calculation effort affects the total duration of a simulation run. Depending on the kind of simulated problem various solutions can be used to reach an approximately even balance. Using static domains is often sufficient for simple and well known problems where the calculation effort can be predicted with high accuracy. If the problem gets more complex adapting the size of the domains dynamically is an obvious possibility to keep the calculation effort balanced. Various different dynamic load balancing strategies are for example discussed by *(*19*)*. However, adapting the domain sizes again produces additional overhead. Hence, accepting a certain amount of imbalance between the calculation efforts may be preferable.

When applying *domain decomposition* to a traffic flow simulation it is feasible to create the domains based on the simulated network structure. The infrastructure objects like links, nodes and traffic lights are assigned to the domains. The agents are dynamically assigned to the thread that handles the infrastructure object on which they are physically present.

Selecting the objects that belong to a domain again can be done in different ways. Using a random assignment typically results in a good load balance between the domains and therefore no further mechanisms to check and adapt the balance are needed. Another advantage of a random approach is that it is simple to implement and no problem specific knowledge is necessary. A clear drawback is that the amount of interactions between different domains is extremely high.

Another approach is to create the domains based on the network structure. Areas with high connectivity are consolidated into domains and domain boarders are placed in areas with only low connectivity. A significant advantage of such an approach is that the level of communication between different domains is minimized because most simulated actions only involve objects which belong to the same domain. However, creating such domains with comparable computational workloads is very complicated for typical simulation problems. In a traffic flow simulation, the computational effort typically depends more on the traffic volume than on the number of network links. Accordingly, the domain sizes should be chosen based on traffic flow information. Yet, the load balance may fluctuate significantly during a simulation—e.g. a domain that contains only housing zones has high traffic volumes in the morning and evening but only low ones in between.

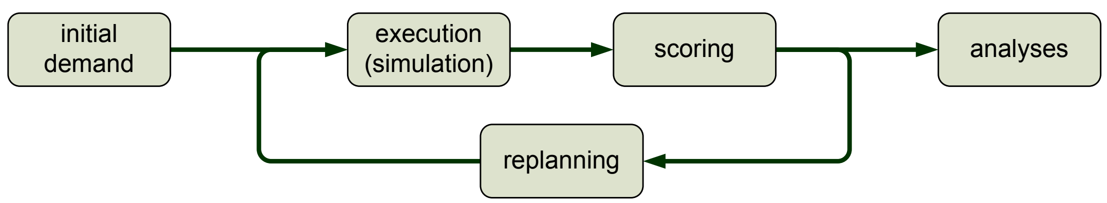
MATSIM

Overview

MATSim (Multi Agent Transport Simulation) is a framework for iterative, agent-based microsimulations of transport systems that is currently developed by teams at ETH Zurich and TU Berlin. It consists of several modules that can be used independently or as part of the framework. It is also possible to extend the modules or replace them with new implementations. Balmer *(*20*)* and Balmer *et al. (*21*)* give a detailed description of the framework, its capabilities and its structure. Because of its agent-based approach, every person in the system is modeled as an individual agent in the simulated scenario. Each agent has personalized parameters such as age, sex, available transport modes and scheduled activities. Due to the modular structure of the simulation framework, the agent's parameterset can be easily extended my new parameters, for example for the routing strategy that should be used or areas of the road network that the agent knows. The application of MATSim to a large scale scenario of Switzerland (over 6 million agents simulated on a high resolution network with 1 million links) is presented by Meister *et al. (*22*)*.

Figure 1 shows the structure of a typical, iterative MATSim simulation run. After the creation of the initial demand, the plans of the agents are modified and optimized in an iterative process until a relaxed state of the system has been found. The analysis of the results can be performed afterwards.

FIGURE 1 Iterative MATSim Loop



The loop contains the elements *execution (simulation)*, *scoring* and *replanning*. Within the simulation module, the plans of the agents are executed. Afterwards, the scoring module uses a utility function to calculate the quality of the executed plans. The utility function for MATSim is described by Charypar and Nagel *(*23*)*. Based on the results by scoring module, the replanning module creates new plans by varying start times and durations of activities as well as the routes to travel from one activity to another. Replanning modules currently under development will additionally allow to change order of the planned activities *(*24*)* as well as the locations where they are performed *(*25*)*.

Simulation of the traffic behavior is also part of the iterative loop. Currently, four different simulation modules are available. Their task is to execute the plans of the agents within the simulated scenario. The following section describes these four simulation modules.

Simulation Modules

QueueSimulation

The *QueueSimulation* is a deterministic, Java based re-implementation of Cetin's *SQSim (*26, 20*)*. The simulation is based on a queue model and uses a time step based approach with a step size of one second. Within each time step, the state of the queues is considered. As a result the duration of a simulation run increases proportionally to the number of links in the network and is independent of the number of simulated agents. A major disadvantage of the *QueueSimulation* is its single core architecture. While other tasks in an iteration of MATSim can be executed in parallel threads (for example the replanning), the *QueueSimulation* still only uses one CPU core. The *QueueSimulation* offers some benefits like well documented code and its simulation listener concept which allows additional modules to interact with the simulation while it is running.

QSim

Basically the *QSim* can be described as an extended version of the *QueueSimulation*. It contains several additional recently developed features like traffic signals *(*27*)* or simulated public transport *(*28*)*. While the *QueueSimulation* can be seen as a default implementation of a traffic simulation module with a stable state, the *QSim* is still under development. Some new features like a redesigned *Within Day Replanning Framework (*based on 29*)* will be fully implemented in the near future.

DEQSim

Another implementation is the *DEQSim*, which implements an extended queue model and is described in detail by Charypar *et al. (*1*)* and Charypar *et al. (*17*)*. In addition to the FIFO (first in, first out) behavior of the queues, a gap is simulated that moves backwards through the queues which allows to simulate congestion more realistically. Two major attributes of this implementation are its multi-threaded architecture and its event based approach. As a result the calculation effort scales with the number of agents. Compared with the time step based approach of the *QueueSimulation* the event based implementation of the *DEQSim* achieves significantly shorter calculation time. A disadvantage of the *DEQSim* is that it is implemented in C++ whereas MATSim is written in Java. Therefore the communication between them is done using a time consuming file input/output interface which produces noticeable longer computation times.

JDEQSim

The *JDEQSim* is the fourth simulation module currently available in MATSim. It is a re-designed re-implementation of the *DEQSim* in Java that is described in detail by Waraich *et al.* *(*30*)*. Due to conceptual differences between C++ and Java it was not possible to reach performance gains by implementing the multi-threaded architecture of the *DEQSim*. Therefore, the *JDEQSim* uses only a single CPU core. However, due to its event based approach the calculation effort is significant lower compared to the *QueueSimulation* and the *QSim*.

IMPLEMENTATION

General Conditions of the Parallelization Approach

The first decision that has to be made is whether a new simulation module should be written from scratch or if an existing one should be adapted. As presented, already multiple different simulation modules for MATSim are available. They offer a wide range of functionality and have already been used for various projects. Additionally, several simulations are documented which can be used for performance comparisons. Therefore, reusing one of those simulations is preferred.

As second step, it has to be decided whether the parallel simulation should base on a distributed or a parallel computing approach. Using the second one is preferred for two reasons. On the one hand, the implementation in the existing MATSim framework should possible with less effort and higher performance due to the fast data exchange between multiple threads. On the other hand, the performance of desktop computers has increased significantly within the last years—concerning computational power as well as available memory.

The next decision to be made concerns the workload distribution. A first implementation of a functional decomposition for the MATSim simulation modules has already been presented *(*30*)*. By handling the events occurring in a separated thread, a remarkable reduction of computation times is reached. However, the remaining computational effort of the simulation cannot be divided into further functional blocks. Thus, implementing a domain decomposition approach is necessary to reach further performance improvements.

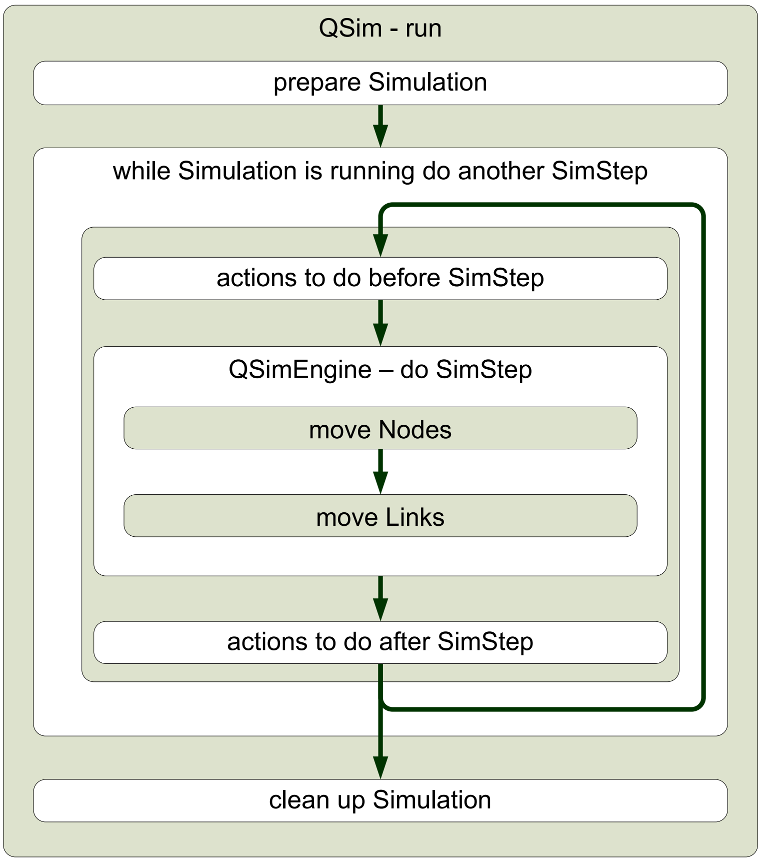
Finally, it has to be decided whether a simulation with a time step based or a event driven approach should be used for the implementation. While event driven approaches tend to perform better in the field of traffic flow simulations time step based approaches seem to be easier to parallelize. There, the time steps can be used as fixed synchronization points which should reduce the communication overhead between multiple threads dramatically. The consideration of above factors and additional analysis of the source codes leads to the conclusion that the time step based *QSim* is best basis for the implementation of a parallel micro-simulation.

Analysis of QSim

A simplified picture of the structure of *QSim* is shown in Figure 2. At first the simulation module has to be prepared, for example to create the simulated agents. Afterwards the simulation itself is started. In a loop the state of the simulated scenario is calculated for each time step. When no further time steps have to be simulated, some data structures, which were only used by *QSim*, are removed from memory.

A performance analysis shows that the *doSimStep* method in the *QSimEngine* consumes over 90% of the computation time of a simulation run. In this context, only the computation

FIGURE 2 Simplified structure of QSim



time of the simulation itself is considered, efforts for the scoring and replanning modules are ignored. Thus, the main focus is on the parallelization of that method. Within *doSimStep* two methods with comparable computational effort are called—*moveNodes* and *moveLinks*.

The *moveNodes* method handles vehicles that leave one link and enter another one. Typically a *Random* object is used to select in which order the ingoing links are handled (If all nodes are controlled by light signals, the *Random* object is never used). Therefore the result of a simulation is influenced by the order in which the nodes are processed. This can be avoided by assigning a *Random* object to each node. Doing so will create deterministic simulation results that will slightly differ from results calculated with *QSim* because other sets of random numbers will be used. When using multiple *Random* objects on parallel threads it is necessary to guarantee that the random numbers are independent from each other. This for example would not be the case if each *Random* object is initialized with the same initial value.

moveLinks simulates the actions (e.g. agents which start and end activities) on the links as each link can be treated independently from the other ones. Therefore the links can be simulated on multiple threads without concerning about race conditions with one exception (Race conditions occur in situations where the result of an operation depends on the timing of events that are created concurrently on parallel running threads, which leads to an indeterministic behavior of the system). *QSim* can teleport vehicles from one link to another one. If within one time step multiple vehicles are teleported from different threads to one link, their order may vary. In that case they have to be ordered by their agent Id to ensure that the simulation result is deterministic.

At some points within *moveNodes* and *moveLinks* calls to methods in global objects (*QSim*, *QSimEngine* and *Simulation*) are executed. If multiple threads performs such method calls concurrently this may result in an unpredictable behavior of the simulation. This can be avoided by using one of two strategies. A simple but slow approach is to allow only one thread at a time to call such a method. Especially if many concurrent calls from multiple threads occur this will be a performance bottleneck. The second strategy is more complex and requires more changes in the code but results in better performance. The method is moved from the global object to one which exists once per parallel thread. Additionally it may be necessary to create an additional, supervising method that is executed from the main thread.

This can be illustrated with a simple example. Links that do not contain active vehicles are deactivated by *QSim* to reduce the calculation effort. When a vehicles enters the link, the link has to be reactivated which is done by calling a method in the *QSimEngine*. There the link is added to a list which is processed at a later point in time. In a parallel *QSim* each thread could contain such a list. Finally the additional supervising method can instruct all threads concurrently to reactivate the links which they have marked before.

Structure of ParallelQSim

If the current design of *QSim* that contains a single *QSimEngine* would be used in a parallel implementation, a lot of method calls in the *QSimEngine* would have to be synchronized to avoid problems with indeterministic behavior. This would result in a poor performance. This problem can be avoided by using one *QSimEngine* per thread. The *ParallelQSim* introduces a code structure where a single *MultiThreadQSimEngine* manages an array of *QSimEngineThreads* that extend the Java *Thread* class and can act as *QSimEngines*. This results in the structure shown in Figure 3(a).

The *MultiThreadQSimEngine* is a wrapper class that manages the communication between the *ParallelQSim* and the *QSimEngineThreads*. As a result of that structure, the *ParallelQSim* sees only the *MultiThreadQSimEngine* and is not involved in the handling of the threads—it does not even recognize that there are multiple threads involved in the simulation.

The *QSimEngineThreads* are created once per iteration of the simulation and reused in every sim step which is considerably faster than creating new threads in each sim step. As shown in Figure 3(b) this is realized by two *CyclicBarriers* (*StartBarrier* and *EndBarrier*) that are part of the Java *concurrent* package. A third *CyclicBarrier* (*SeparationBarrier*) is used to synchronize the *moveNodes* and *moveLinks* actions. The threads must have handled all their nodes before they can continue with the links. When the *doSimStep* method of the *MultiThreadQSimEngine* is called, it starts the threads by triggering the *StartBarrier* and then waits until all threads have reached the *EndBarrier*.

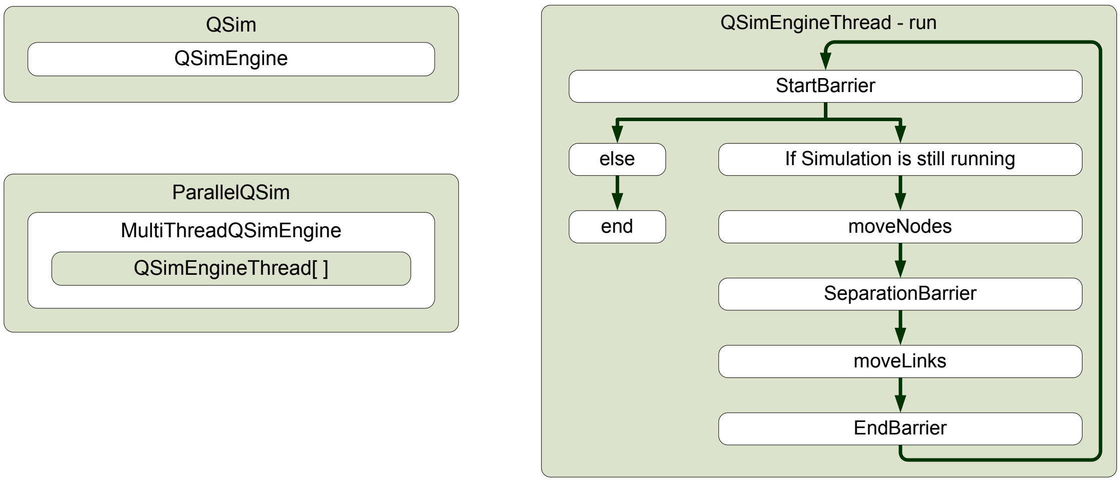
PERFORMANCE MEASUREMENTS

Hardware

The experiments employed to compare the performance of the *ParallelQSim* with the existing *QSim* are run on a computer with two quad core CPUs (each a AMD Opteron 2380) and 24 GB of shared memory. A maximum of 7 cores is used for the *ParallelQSim*. The remaining core is used for (parallel-)events handling and some background processes.

FIGURE 3 Structure of the Implementation

(a) Comparison of QSim and ParallelQSim (b) Structure of a QSimEngineThread



Scenarios

As a first scenario, a 1% example of Berlin is used which is a basic example scenario used by MATSim. It contains about 16K agents who perform 28K trips and is simulated on a network with about 11k nodes and 28K links. During a simulation run 1M events are created.

For the second and third scenario a model of Canton Zurich is used—once as 25% sample with 400K agents and 1.3M performed trips and once as 100% sample with 1.6M agents and 5.1M performed trips. The network contains 73K nodes and 163K links. A simulation run creates 47M and 158M events, respectively.

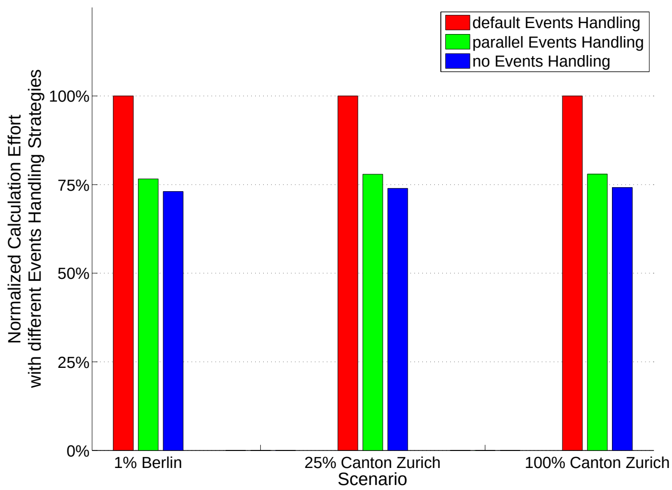
These are real world scenarios that are typically simulated with MATSim. It is assumed, that the results of the performance measurements can be reached on other, comparable, scenarios as well.

Results

The *ParallelQSim* uses the same simulation logic as *QSim*. However, simulation results produced by the *ParallelQSim* are slightly different from the ones created by *QSim*, which is a result of using multiple Random objects instead of a single one. From a traffic planning point of view the results are absolutely comparable and therefore the results of the simulations runs in this section are only analyzed regarding the performance of the used simulation setup (queue simulation and events handling strategy). Conclusions concerning the results from a traffic planning view have already been drawn *(*21*,* 22*)*.

Figure 4 shows the calculation effort for the events handling in the three scenarios. The results show that the effort constitutes 25% of the total calculation effort and is not influenced by the size of the scenario. According to Amdahl's Law *(*31*)*, which describes the maximum achievable speedup of a programm with partially parallelized code, this affects significantly the performance gain reachable. The influence of the non-parallel code can be illustrated with a simple example. If code that consumes 5% of the computation time of a program cannot be parallelized, the total calculation time cannot be reduced by more than a factor twenty—even if the remaining code could be handled in zero seconds. As a result, the events handling limits the possible speed gain to a factor four of the calculation time of *QSim* with non-parallel events handling. Relative to the runs of *QSim* with parallel events handling, a performance gain of factor three is possible.

FIGURE 4 Performance of different Events Handling Strategies



Another important finding, that is also depicted in Figure 4, is the high efficiency of *ParallelEventsManager* in combination with *QSim*. Almost the entire calculation effort of the events handling is moved from the main thread to a separate thread. As a result, the simulation is nearly as fast as it would be without any events handling.

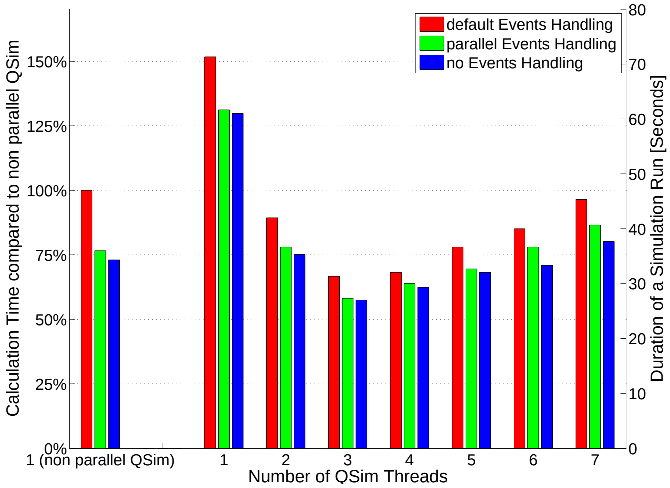
Figures 5(a) to 5(c) show the results of the runs with the three test scenarios. Each figure contains the results of runs employing *ParallelQSim* using one to seven cores and different event handling strategies. Additionally the same scenarios have been run with the non-parallel *QSim*.

When the computation times of the *QSim* and the *ParallelQSim* using only one thread are compared, the difference between the calculation times is the overhead caused by the parallelization such as distributing and synchronizing data between the threads. In the Berlin scenario, a significant overhead of over 50% is found. As a result, the *ParallelQSim* cannot reduce the calculation time significantly compared to the *QSim*. However, the *ParallelQSim* itself performs quite well. The calculation time decreases by 50% if three threads are used instead of a single one. The Canton Zurich scenarios show that the calculation overhead is less significant if the scenario gets more complex. The overhead reduces to 30% (25% scenario, using 5 threads) and 20% (100% scenario, using 6 threads). Therefore, the performance gains rise up to the—according to Amdahl's Law—highest reachable value of a factor three when using parallel events handling.

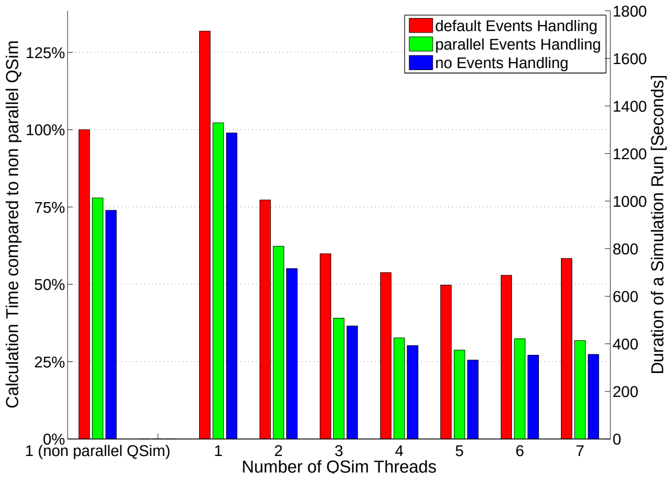
The comparison of the results of Canton Zurich runs with parallel events handling and runs without events handling shows that there is still only a small difference in computation time. Hence, we can assume that the computation times of the events handling and the *ParallelQSim* are almost alike in these scenarios. The runs without events handling would have a noticeable shorter computation time if the events handling had become a bottleneck. However, if the simulated scenarios get even bigger events handling could clearly become a performance

FIGURE 5 Results of the Sample Scenarios

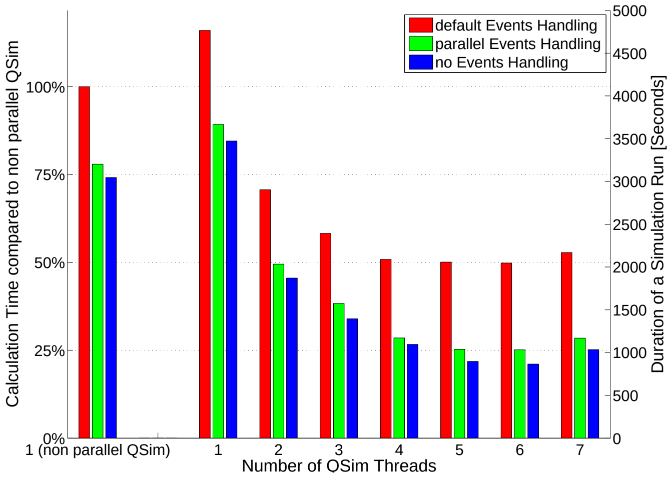
(a) Results 1% Berlin Scenario



(b) Results 25% Canton Zurich Scenario



(c) Results 100% Canton Zurich Scenario



bottleneck.

Considering only the results of the *ParallelQSim*, the number of cores used which results in the lowest calculation times rises with the total calculation effort for the scenario. While the Berlin scenario performs best with only three cores, the 25% Canton Zurich scenario should be run with five cores and the 100% Canton Zurich scenario benefits from up to six cores. An important detail is that using too many cores results in increased computation times which is a consequence of the synchronization effort that increases with the number of used cores.

CONCLUSION AND OUTLOOK

This paper describes the the development and implementation of a new simulation module in MATSim that reaches short calculation times by using multiple CPU cores. An adaption of the *QSim* was chosen because its time steps can be used to synchronize the data between the parallel calculation threads. Distributing the workload is done by a simple approach where the assignment of the network's links and nodes to the threads is done randomly.

The results of the performance tests show that—depending on the scenario size—the calculation time can be reduced by a factor of four. Based on Amdahl's Law it is shown that the events handling could become a bottleneck when simulating large scale scenarios. A speedup of more than a factor of four is not possible. However, it is shown that the existing parallel events handling reduces the calculation effort in the main thread very efficiently. Moving the events that are created in the main thread to another thread where they are handled is done within negligible time.

Another important results of the performance tests concerns the number of CPU cores used. Depending on the complexity and size of the scenario the number of cores resulting in the best simulation performance varies. Hence, a user should keep the results of the sample scenarios in mind when choosing the number of cores for another scenario. Comparing the scenario with the given samples in this paper should lead to a reasonable choice.

Although the results are already very satisfying there are still some further performance optimizations possible and desirable. One major point concerns the synchronization effort between the threads. Especially in smaller scenarios, this performance bottleneck reduces the attractiveness of using the *ParallelQSim*. Another topic for further developments is the events handling. Having a setup where each thread has its own set of events handlers would reduce the amount of synchronized method calls significantly and therefore should results in further performance gains.

REFERENCES

1. Charypar, D., K. W. Axhausen and K. Nagel (2007) An event-driven queue-based traffic flow microsimulation, *Transportation Research Record*, **2003**, 35–40.

2. Ferber, J. (1999) *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*, Addison-Wesley, Boston.

3. Wooldridge, M. (2000) *Reasoning about Rational Agents*, MIT Press, Cambridge.

4. Klügl, F. (2001) *Multiagentensimulation - Konzepte, Werkzeuge, Anwendung*, Addison-Wesley, Munich.

5. Eymann, T. (2003) *Digitale Geschäftsagenten - Softwareagenten im Einsatz*, Springer, Berlin.

6. Barceló, J., J. L. Ferrer, D. Garcia, M. Florian and E. Le Saux (1998) Microscopic traffic simulation, in P. Marcotte and S. Nguyen (eds.) *Equilibrium and Advanced Transportation Modelling*, chap. 1, 1–26, Kluwer, Dordrecht.

7. Nökel, K. and M. Schmidt (2002) Parallel DYNEMO: Meso-scopic traffic flow simulation on large networks, *Networks and Spatial Economics*, **2** (4) 387–403.

8. Nagel, K. and M. Rickert (2001) Parallel implementation of the TRANSIMS micro-simulation, *Parallel Computing*, **58** (2) 1611–1639.

9. Niedringhaus, W. P., J. M. Opper, L. Rhodes and B. L. Hughes (1994) Ivhs traffic modeling using parallel computing: Performance results, in H. J. Siegel (ed.) *Proceedings of the 8th International Symposium on Parallel Processing*, 688–693, IEEE Computer Society, Washington, D.C.

10. Cameron, G. D. B. and G. I. D. Duncan (1996) Dynamic process and equilibrium in transportation network: Towards a unifying theory, *Journal of Supercomputing*, **10** (1) 25–53.

11. Fujimoto, R. M. (2001) Parallel and distributed simulation systems, in B. A. Peter, J. S. Smith, D. J. Medeiros and M. W. Rohrer (eds.) *WSC '01: Proceedings of the 33nd conference on Winter simulation*, 147–157, IEEE Computer Society, Washington, D.C.

12. Robinson, S. (2004) *Simulation: The Practice of Model Development and Use*, John Wiley & Sons, Chichester.

13. Hartrum, T. C. and B. J. Donlan (1988) HYPERSIM: Distributed discrete-event simulation on an iPSC, in G. Fox (ed.) *Proceedings of the third conference on Hypercube concurrent computers and applications: Architecture, software, computer systems, and general issues*, vol. 1, 745–747, Association for Computing Machinery, New York.

14. Ferscha, A. (1996) Parallel and distributed simulation of discrete event systems, in A. Y. H. Zomaya (ed.) *Parallel and Distributed Computing Handbook*, 1003–1041, McGraw-Hill, New York.

15. Fujimoto, R. M. (1989) Parallel discrete event simulation, in E. A. MacNair, K. J. Musselman and P. Heidelberger (eds.) *WSC '89: Proceedings of the 21st conference on Winter simulation*, 19–28, Association for Computing Machinery, New York.

16. Logan, B. and G. Theodoropoulos (2001) The distributed simulation of multi-agent systems, *Proceedings of the IEEE*, **89** (2) 174–186.

17. Charypar, D., K. W. Axhausen and K. Nagel (2007) An event-driven parallel queue-based microsimulation for large scale traffic scenarios, paper presented at the *11th World Conference on Transportation Research*, Berkeley, June 2007.

18. Hanebutte, U. R. and A. M. Tentner (1995) Traffic simulations on parallel computers using domain decomposition techniques, paper presented at the *2nd World Congress on Intelligent Transport Systems*, Yokohama, November 1995.

19. Willebeek-LeMair, M. H. and A. P. Reeves (1993) Strategies for dynamic load balancing on highly parallel computers, *IEEE Transactions on Parallel and Distributed Systems*, **4** (9) 979–993.

20. Balmer, M. (2007) Travel demand modeling for multi-agent traffic simulations: Algorithms and systems, Ph.D. Thesis, ETH Zurich, Zurich, May 2007.

21. Balmer, M., M. Rieser, K. Meister, D. Charypar, N. Lefebvre, K. Nagel and K. W. Axhausen (2008) MATSim-T: Architektur und Rechenzeiten, paper presented at the *Heureka '08*, Stuttgart, March 2008.

22. Meister, K., M. Balmer, F. Ciari, A. Horni, M. Rieser, R. A. Waraich and K. W. Axhausen (2010) Large-scale agent-based travel demand optimization applied to Switzerland, including mode choice, paper presented at the *12th World Conference on Transportation Research*, Lisbon, July 2010.

23. Charypar, D. and K. Nagel (2005) Generating complete all-day activity plans with genetic algorithms, *Transportation*, **32** (4) 369–397.

24. Feil, M. (2010) Choosing the daily schedule: Expanding activity-based travel demand modelling, Ph.D. Thesis, ETH Zurich, Zurich.

25. Horni, A., D. M. Scott, M. Balmer and K. W. Axhausen (2009) Location choice modeling for shopping and leisure activities with MATSim: Combining micro-simulation and time geography, *Transportation Research Record*, **2135**, 87–95.

26. Cetin, N. (2005) Large-scale parallel graph-based simulations, Ph.D. Thesis, ETH Zurich, Zurich.

27. Neumann, A. (2008) Modellierung und Evaluation von Lichtsignalanlagen in Queue-Simulationen, Master Thesis, Technical University Berlin, Berlin.

28. Rieser, M. (2010) Adding transit to an agent-based transportation simulation, Ph.D. Thesis, Technical University Berlin, Berlin.

29. Dobler, C. (2009) Implementations of within day replanning in MATSim-T, *Working Paper*, **598**, IVT, ETH Zurich, Zurich.

30. Waraich, R. A., D. Charypar, M. Balmer and K. W. Axhausen (2009) Performance improvements for large scale traffic simulation in MATSim, paper presented at the *9th Swiss Transport Research Conference*, Ascona, September 2009.

31. Amdahl, G. M. (1967) Validity of the single processor approach to achieving large scale computing capabilities, paper presented at the *Spring Joint Computer Conference*, New York, April 1967.