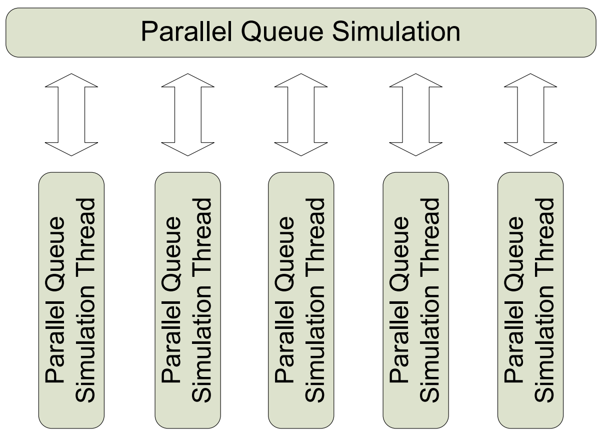
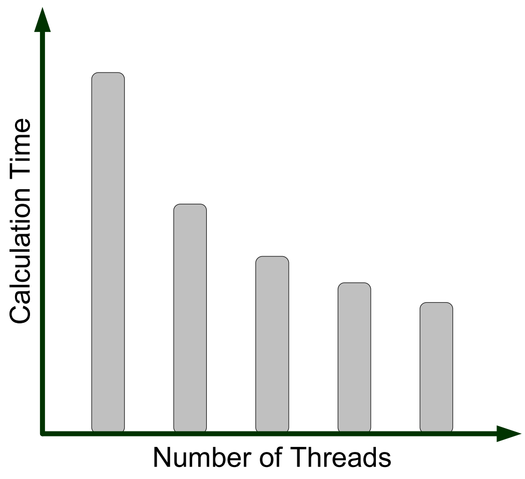

Parallel Queue Simulation
Parallel Queue Simulation Thread
Parallel Queue Simulation Thread
Parallel Queue Simulation Thread
Parallel Queue Simulation Thread
Parallel Queue Simulation Thread



Implementation of a Time Step Based Parallel Queue Simulation in MATSim

Christoph Dobler

STRC 2010 **August 2010**

Institut für Verkehrsplanung und Transportsysteme
Institute for Transport Planning and Systems

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

STRC 2010

Implementation of a Time Step Based Parallel Queue Simulation in MATSim

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

August 2010

Abstract

Today, agent-based micro-simulations are commonly used in the field of transport planning and traffic management. Besides modeling of realistic traffic behavior, another 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 time step based, parallel queue simulation for MATSim written in Java.

In a first step, the structure of the MATSim framework and especially the existing simulation modules are analyzed with regards to their structure, performance and extensibility. The results of this analysis show that one of those simulation modules can be re-used. The approach that is implemented by that module can be adapted in a way that the computation is run concurrently on multiple cores. A strategy to distribute the computation workload between multiple cores is proposed and implemented after comparing the advantages and drawbacks of different approaches.

Furthermore, performance bottlenecks in the existing MATSim code that occurred in combination with the new simulation module are identified and removed. Performance tests with different sized scenarios are conducted. An analysis of the results shows that especially on large scaled scenarios a significant performance gain is reachable.

Keywords

Parallel Queue Simulation, MATSim

1 Introduction

Today, agent-based micro-simulations are commonly used in the field of transport planning and traffic management. Besides the modeling of realistic traffic behavior, another 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 speed up by using a faster CPU.

Within the last years, the development focus has changed dramatically. Today we can assume 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 time step based parallel queue simulation that reduces the simulation time of large scale scenarios significantly. Chapter 2 describes MATSim—a toolkit for multi-agent micro-simulations of transport systems. Subsequently different simulation approaches are discussed. In a next step strategies to distribute calculation effort among parallel threads are analyzed. Chapter 6 gives detailed information on the implementation of the parallel simulation. Afterwards the results of performance tests with three different sized sample scenarios are described and analyzed. Finally conclusions are drawn and an outlook on future developments is given.

2 MATSim

2.1 Overview

MATSim (Multi-Agent Transport Simulation) is a framework for iterative, agent-based micro-simulations 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 (2007) and Balmer *et al.* (2008) give a detailed description of the framework, its capabilities and its structure. Because of its agent-based approach, each person in the system is modeled as an individual agent in the simulated scenario. Each of these agents has personalized parameters such as age, sex, available transport modes and scheduled activities per day. Wooldridge (2000), Klügl (2001), Eymann (2003) and Ferber (1999) give a detailed overview on (multi-)agent-systems and simulations. Due to the modular structure of the simulation framework, the agent's parameters can be easily extended with new parameters, for example for the routing strategy that should be used or the areas of the road network that the agent knows.

Figure 1: Iterative MATSim Loop

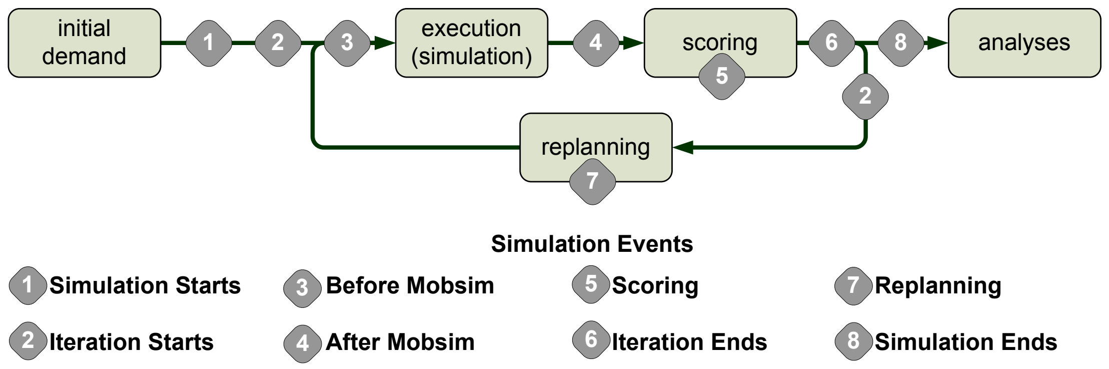


Figure 1 shows the structure of a typical, iterative MATSim simulation run. After the creation of the initial demand, the plans of the agents are modified and optimized in an iterative process until a relaxed state of the system has been found. The analysis of the results can be performed afterwards.

The loop contains the elements *execution (simulation)*, *scoring* and *replanning*. Within the execution module the plans of the agents are simulated. Afterwards the scoring module uses a utility function to calculate the quality of the executed plans. A commonly used utility function for MATSim runs is described by Charypar and Nagel (2005). Based on the scoring module, the replanning module creates new plans by varying values like start time and duration of activities as well as the routes to travel from one activity to another one. Currently developed replanning modules will additionally allow to change the elements and their order of performed activity chains (Feil, 2010) as well as their locations (Horni *et al.*, 2009). More detailed information on the loop and its modules can be found in the previously mentioned papers.

One part of the iterative loop is the simulation of the traffic behavior. 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.

2.2 Simulation Modules

2.2.1 QueueSimulation

The *QueueSimulation* is a deterministic, Java based re-implementation of Cetin's *SQSim* (Cetin, 2005; Balmer, 2007). The simulation is based on a queue model and uses a time step based approach with a step width of one second, meaning that the system state is calculated every second. Within each time step, the state of the queues is considered. As a result, the duration of a simulation run increases proportional 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), several attempts to write a multi-threaded version of the *QueueSimulation* have ended without satisfying results. A major reason for the failure of those approaches was the complexity of the source code—especially of the *QueueSimulation*, but also of the other components of the MATSim project. Implementing a parallel simulation based on that code resulted in a variety of consistency checks to ensure that the simulation does not produce indeterministic results caused by race conditions. Therefore, the performance of the parallelized code was not satisfying. However, today the situation is a different one. Extensive work on the source code, which was done between autumn 2009 and spring 2010, improved the quality of the code sustainable regarding its suitability for parallelization. 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.

2.2.2 QSim

Basically, the *QSim* can be described as an extended version of the *QueueSimulation*. It contains several additional recently developed features like traffic signals (Neumann, 2008) or simulated public transport (Rieser, 2010). While the *QueueSimulation* is some kind of 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 Dobler, 2009), will be fully implemented in the near future.

2.2.3 DEQSim

Another implementation is the *DEQSim*, which implements an extended queue model and is described in detail by Charypar *et al.* (2007). Additionally to the FIFO (first in, first out) behavior of the queues, also 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, which means that calculations within the simulation are only necessary if an event occurs. 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 times. However, 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 by using a time consuming file input/output interface which produces noticeable longer computation times.

2.2.4 JDEQSim

The *JDEQSim* is the forth simulation module currently available in MATSim. It is a redesigned re-implementation of the *DEQSim* in Java that is described in detail by Waraich *et al.* (2009). 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*.

3 Simulation Approaches

In a first step, the existing MATSim simulation modules and their simulation approaches are reviewed in detail to decide whether they can be used as basis for a parallel simulation module or if a new module has to be created from scratch. At the moment, there are two different simulation approaches that are implemented by the existing simulation modules. Both of them have assets and drawbacks, so a detailed analysis of them is necessary.

3.1 Event Driven Approach

The *DEQSim* and *JDEQSim* use an event driven approach. In this approach, the simulation modules perform actions each time an event is created by an simulated object. As a result, a simulation will end immediately if nothing happens that create events. Typically, only a single event happens at a specific point in time. This leads to problems when trying to handle multiple events on parallel threads. If an event is handled, it has to be ensured that every event that happened previously and that could influence the currently handled event has already been processed.

The *DEQSim* is already able to run in a parallel mode but has the drawback that it is written in C++ whereas MATSim is written in Java. Due to some differences in the handling of threads and the data exchange between them an implementation of the same parallelization strategy in Java would not deliver satisfying results. C++ allows to exchange data between threads with only little delay, in Java the threads have to be synchronized which consumes much more computation time. Synchronization in this context means to ensure that if an object is edited in a thread the performed changes are visible in all other threads.

3.2 Time Step Based Approach

The second approach is time step based and is used by the *QueueSimulation* and the *QSim*. The simulated period is split up into equal pieces with a given duration—the time step size. Each of these time steps and events occurring within such a step are treated time discrete. All events are handled as if they had happened at the same point in time.

On the one hand, a time step based approach produces some overhead when nothing happens in a scenario but the simulation module has to update the system state. Such situations may especially occur in scenarios with only a couple of simulated agents and at times where the traffic volume is very low like early in the morning. On the other hand, the time steps can be used as break and synchronization points. Actions within one such time step can be performed on parallel threads without additional synchronization effort as long as they do not interfere with each other. This can be guaranteed by writing an events handling logic that does not take other events into account that occurred in the present time step.

3.3 Selection of a Simulation Approach

Takeing the *QSim* as basis for a parallel queue simulation seems promising for multiple reasons. Using the time steps as fixed synchronization points between all threads should reduce the time consuming synchronization actions, which in turn reduces the computation time significantly. Several recently implemented modules use the *QSim*. It should be possible to use them also with a parallel QSim without significant adaptions in their code. Also further additional modules will likely use the *QSim* due to its simple and easily understandable structure.

4 Distribution of the Workload

As a next step, the distribution of the workload has to be defined. Typically this is done by assigning the objects (e.g. agents, links and nodes, signal lights) that perform actions and create events to different threads. Depending on the type of simulation, it is possible to use a static and / or dynamic assignment of the objects. Static means in this context that an object is always handled by the same thread, dynamic that an object may be handled by various threads. In a traffic simulation it is reasonable to assign infrastructure object (nodes, links and facilities) statically, whereas the agents are assigned dynamically to the thread that handles the infrastructure object where they are physically present. Two commonly used distribution strategies are described in the following sections.

4.1 Object Based Distribution

In an object based workload distribution, the simulated objects can be assigned to different threads which handle the actions and events that are produced by the objects. Using a random distribution algorithm should ensure that the workload on all threads is comparable. If data is exchanged between different objects it has to be ensured that they are handled by the same thread. Otherwise the threads and their data have to synchronized.

An object based distribution is easy to implement and produces even with a random assignment strategy sufficiently balanced workloads. Additionally, a control instance can be implemented that controls the workload of the calculating threads and adapts the distribution strategy if necessary.

4.2 Area Based Distribution

An area based distribution separates the simulated area into connected sectors. Each of this sectors is handled by a thread. An advantage of this approach is that there is no synchronization necessary as long as only objects within the sector are involved in an action or event. A major drawback is that typically the identification of sectors with comparable workloads is very complicated and computation time intensive. Using equal sized areas will often produce poor results because the traffic volume in a road network depends on the simulated scenario.

A possible solution would be to interactively resize the sectors. If a sector has a higher computation time than his neighbors the load could be balanced by handing over some objects to the neighbor sectors. This load balancing could be done during a running simulation or after an iteration, if an iterative simulation approach is used.

4.3 Selection of a Distribution Approach

Due to the code structure of the *QSim* and the possibilities in Java to handle multiple threads, using an object based distribution approach is most practicable. Implementing such an approach is considerably easier compared with an area based distribution. The usage and handling is less complex and especially no information about the behavior of the traffic flow in the simulated scenario is needed to get an even workload distribution.

One of the main benefits of an area based approach is the reduction in data synchronizations but this is not possible when using the *QSim*. There it is necessary to synchronize the data after each simulated time step. By contrast using such an approach when developing a parallel *JDEQSim* would be complex but also very promising. When implementing such a combination it may be meaningful to minimize the number of links that connect the different areas. If a vehicles drives over a link that connects two areas, it is necessary that the threads which handle those areas synchronize their data. The necessity for doing this can be illustrated by having a look at a simple example. Thread A wants to move a vehicle from one of its links over a node to a link of thread B—but before doing so thread A has to ensure that there is free space on that link. To do so, thread A has to communicate with thread B which checks the available capacity on that link and returns the result. If there is free space available, the vehicle is handed over to thread B which will take over its further simulation. As a result, accepting a certain amount of differences in the thread workloads may still have a higher total performance that perfectly balanced areas with a huge amount of links between different areas.

5 Implementation

5.1 Objectives of the Implementation

Before starting to develop the parallel simulation, some objectives have been defined that should be respected by the implementation. They affect aspects like the usability as well as the maintainability and the extensibility.

- Respect typical programming guidelines and rules within the MATSim project, keep the code simple and extendable.
- Use the existing structure of the simulation (*QSim*, *QSimEngine*, etc.) and adapt it where necessary.
- Make it usable like any other already existing simulation module without significant, additional configuration effort.
- Create deterministic results, which do not depend on the number of parallel simulation threads.
- Detect potential bottlenecks which affect the performance and
  - remove them if the effort and volume of the necessary changes in the code is justifiable.
  - document them so they may be considered and / or removed in future refactorings.

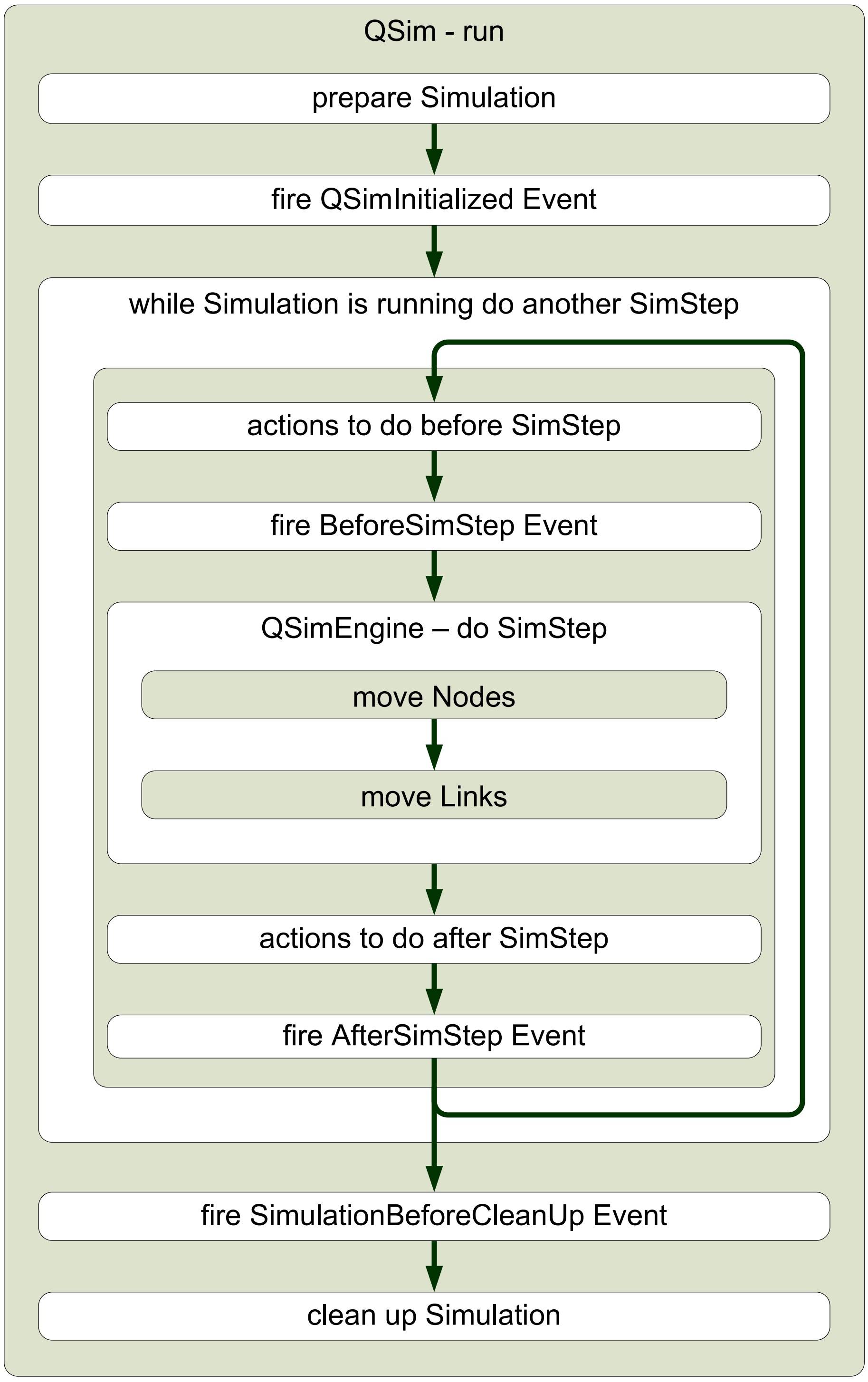
5.2 Analysis of the QSim

In a first step, the structure and the performance of the *QSim* is analyzed. As shown in Figure 2, it contains methods which simulate the traffic behavior and creates events which can be used to interact with listeners. If such an event occurs, all registered listeners are informed and can perform then the required actions while the simulation is waiting.

A performance analysis shows, that the *doSimStep* method in the *QSimEngine* consumes over 90% of the computational time of a simulation run. Therefore, 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

Figure 2: Structure of the QSim


QSim - run
prepare Simulation
fire QSimInitialized Event
while Simulation is running do another SimStep
actions to do before SimStep
fire BeforeSimStep Event
QSimEngine – do SimStep
move Nodes
move Links
actions to do after SimStep
fire AfterSimStep Event
fire SimulationBeforeCleanUp Event
clean up Simulation

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 the *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. 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. The *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 the *Id* of the agent who drives the vehicle 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, what could cause race conditions. This can be avoided by using one of two simple strategies. A simple but slow approach is to add a *synchronized* tag to the methods. Doing this ensures that only one thread at a time executes 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.

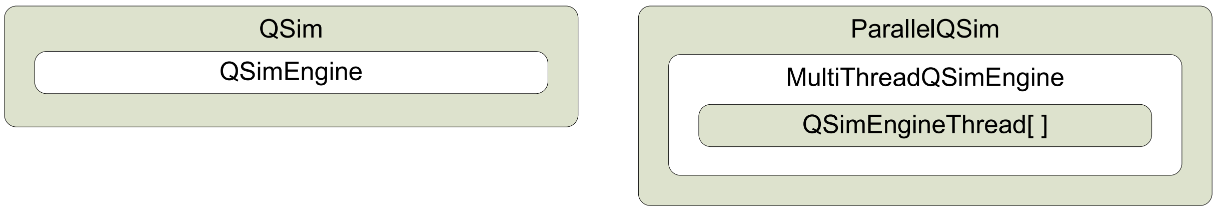
This can be illustrated with a simple example. Links that do not contain active vehicles are deactivate by the *QSim* to reduce the calculation effort. When a vehicles enters the link, the link has to be reactivated which is done by calling a method in the *QSimEngine*. There the link is added to a list which is processed at a later point in time. In a parallel *QSim* each thread could contain such a list. The additional supervising method could then collect the vehicles from the threads and add them to a global list which then is finally processed.

5.3 Structure of the ParallelQSim

If the current design with a *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 race conditions which would result in a poor performance. This problem can be avoided by using one *QSimEngine* per thread. The *ParallelQSim* introduces a code structure where this is realized—it contains a single *MultiThreadQSimEngine* that manages an array of *QSimEngineThreads* which extend the Java *Thread* class and can act as *QSimEngines*. This results in a structure as shown in Figure 3.

Basically, 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

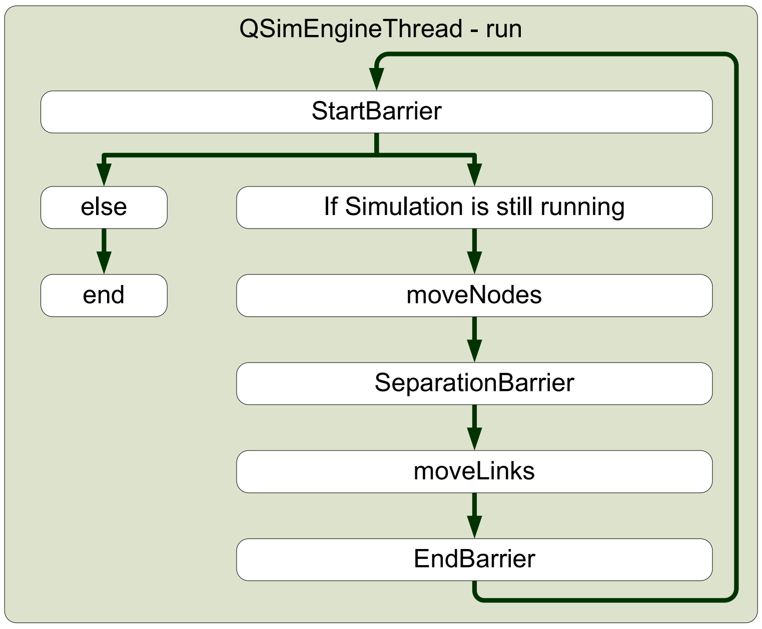
Figure 3: Comparison of the QSim and the ParallelQSim



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 4, this is realized by using two *CyclicBarriers* (*StartBarrier* and *EndBarrier*) which 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 go on with the links. When the *doSimStep* method of the *MultiThread-QSimEngine* is called, it starts the threads by triggering the *StartBarrier* and then waits until all threads have reached the *EndBarrier*.

Figure 4: Structure of a QSimEngineThread



5.4 Changes in the MATSim Core

In the course of development of the *ParallelQSim* some changes have been made in the core. Some of them were necessary to ensure the proper function of the *ParallelQSim*, others were mainly implemented to improve the performance. A last group of changes has due to different reasons not been implemented yet but should nevertheless be mentioned here—they may be considered in future code refactorings.

5.4.1 Agents Counter

The *QSim* uses a class called *Simulation* to count the simulated agents (alive and lost ones) which contains some static methods. Those methods are called from different places in the simulation what can cause race conditions if multiple parallel threads are involved. This problem was solved by using atomic variables from the Java *concurrent.atomic* package. They can be increased or decreased in a single operation that cannot be interrupted. Still this may be a performance bottleneck but due to the fact that the number of method calls is relative small this solution is acceptable.

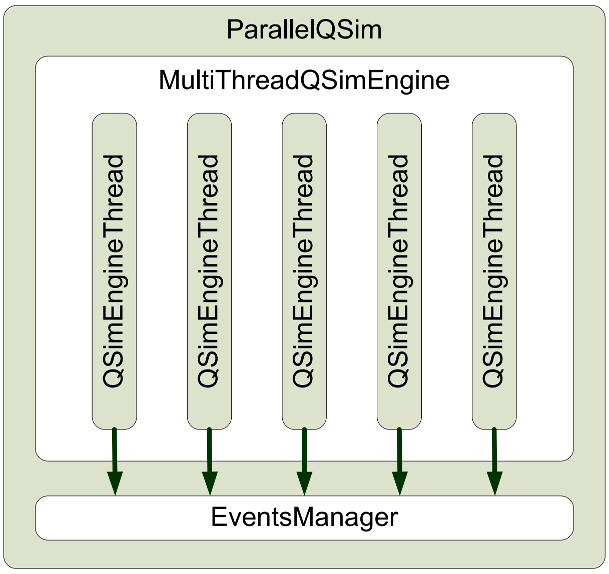
Another possible solution would be to add the counting functionality to the *QSimEngineThreads*. Doing so would eliminate the concurrent method calls but would also require quite a lot of changes in the code of the MATSim core. Because of the expected negligible small performance gain compared to the implementation effort required, this alternative version has not been realized yet.

5.4.2 EventsCollector

Basically, events are not necessary to run simulations with MATSim—but without them, no analyses could be performed. Every time something happens in the simulation (e.g. an agent starts an activity or a vehicle is moved from one link to another one) this is documented by creating an event that contains information about what happened and when did it happen. This information is used by *EventHandlers* to calculate values like trip travel times or mean travel time of a link at a given time.

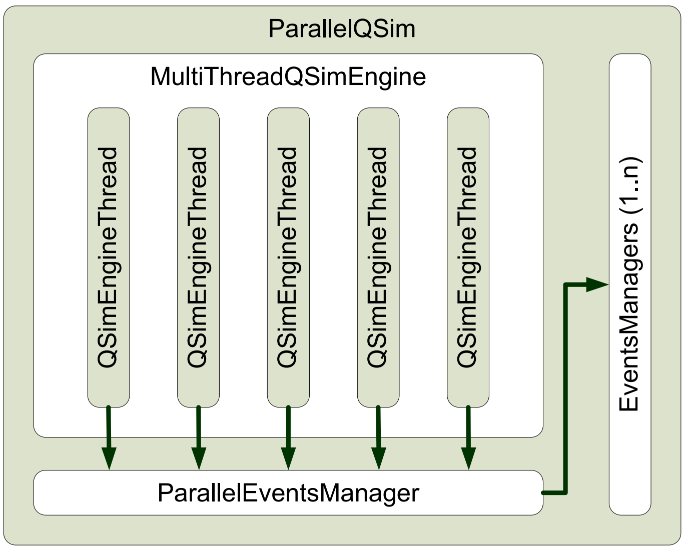
All events that are created by the simulation are sent to the *EventsManager*. There, they are processed by sending them to the *EventHandlers*. Performance measurements show, that this structure with a single *EventsManager* becomes a major bottle neck if a multi-threaded simulation is used. In such a scenario, the threads create multiple events concurrently but the *EventsManager* has to handle them sequentially. To ensure that only one event at a time is handled, it is necessary to synchronize the method calls in the *EventsManager*. Thus, the performance is significantly influenced by the duration of the events handling.

Figure 5: EventsManager



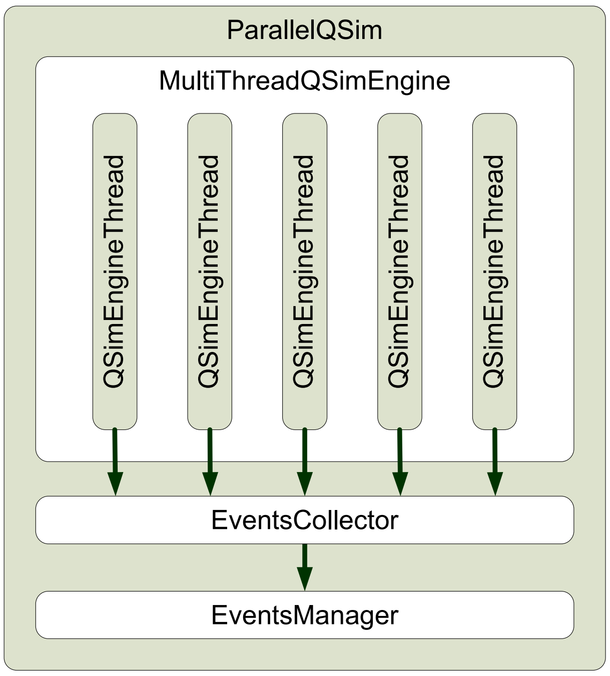
Waraich *et al.* (2009) presents an approach where separate threads are used to handle the events. Still a single *ParallelEventsManager* is used, but instead of handling the events, they are only collected and handed over to other *EventsManager* instances that run in separate threads, where the events are finally handled. Due to the fact that the *ParallelEventsManager* was designed to be used with a single thread simulation module, it cannot handle multiple concurrently created events without synchronizing their access.

Figure 6: ParallelEventsManager



To avoid the time consuming synchronizations, a new *EventsManager* was developed for the *ParallelQSim*—the so called *EventsCollector*. Its basic concept is comparable with the *ParallelEventsManager* but instead of handling events when they occur, the are only collected. After each simulated time step they are handed over to an *EventsManager* which handles them. A major performance gain is reached by using a *ConcurrentLinkedQueue* to collect the events. This special type of queue is unbounded, thread-safe and based on linked nodes. Internally it is an implementation of an efficient, wait-free algorithm that is described by Michael and Scott (1996). A *ConcurrentLinkedQueue* is thread-safe which means that it is not necessary to use time consuming synchronized methods—the queue is able to handle concurrent access. It is also possible to use a *ParallelEventsManager* which handles the collected events in a separated thread. Doing so allows the main simulation thread to continue immediately without waiting until all events have been processed.

Figure 7: EventsCollector

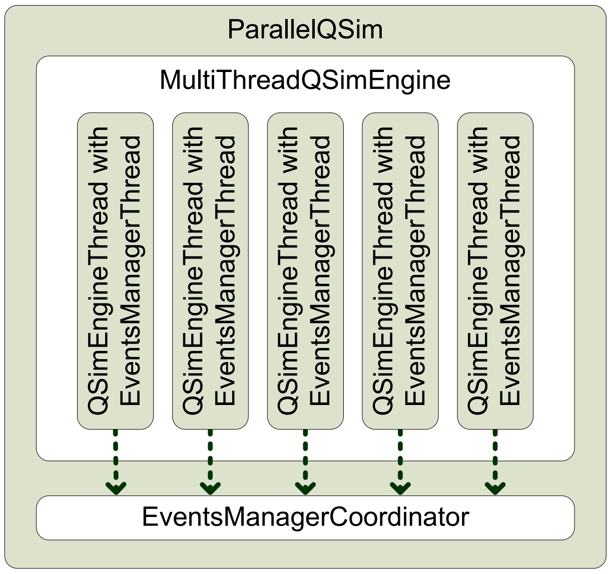


5.4.3 Decentralized Events Handling

All previously described strategies to handle the simulation events are due to their performance bottlenecks only limited capable to be used with the *ParallelQSim*. Avoiding this leak is possible but requires major changes in the existing *EventHandlers*.

The basic idea of a first approach is to create separate *EventsManagers* and *EventsHandlers* for every *QSimEngineThread* where each thread handles only those events that are caused by its assigned simulation objects (agents, links, nodes, etc.). This is basically possible, but leads to a more complex structure as shown in Figure 8. Some *EventsHandlers* are able to act independently within their *QSimEngineThread*, others have to synchronize their data with the other threads externally by using a *EventsManagerCoordinator*.

Figure 8: EventsManagerCoordinator


ParallelQSim
MultiThreadQSimEngine
QSimEngineThread with EventsManagerThread
QSimEngineThread with EventsManagerThread
QSimEngineThread with EventsManagerThread
QSimEngineThread with EventsManagerThread
QSimEngineThread with EventsManagerThread
EventsManagerCoordinator

An *EventsHandler* that calculates leg travel times for example would have to be synchronized with the corresponding handler instances in the other threads because the leg could end on a link that is handled by a different thread than the start link. Without synchronization, the handler in the first thread would not be informed that the agent has ended its current trip and therefore could not calculate the leg travel time. The second thread on the other hand could not handle an arrival event because the corresponding departure event is unknown. To solve this problem, a centralized *EventsManagerCoordinator* would have to merge the incomplete information from the *EventsManagers* in the different *QSimEngineThreads*.

Another approach for decentralized events handling would be to add an event container to each object that can create events. In that case, an *EventsHandler* should be able to get all required information from that set of collected events. A disadvantage of this approach would be that the number of events per object would increase during the simulation what would cause longer lookup times in the collection when searching for specific events. This could be avoided by using a separate list of events for every used *EventsHandler* at the expense of a higher memory consumption. In that case, the handler would decide which events have to be stored and which ones can be removed from the list because they are not needed anymore.

6 Performance Measurements

6.1 Hardware

The experiments that are run to compare the performance of the *ParallelQSim* with the existing *QSim* are run on a computer with two quad core CPUs (each an 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 the (parallel-)events handling and some background processes that are running on the computer. Doing so ensures that the measured performance is not influenced by other processes.

6.2 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% with 1.6M agents and 5.1M performed trips. The used network contains 73K nodes and 163K links. A simulation run creates 47M respectively 158M events.

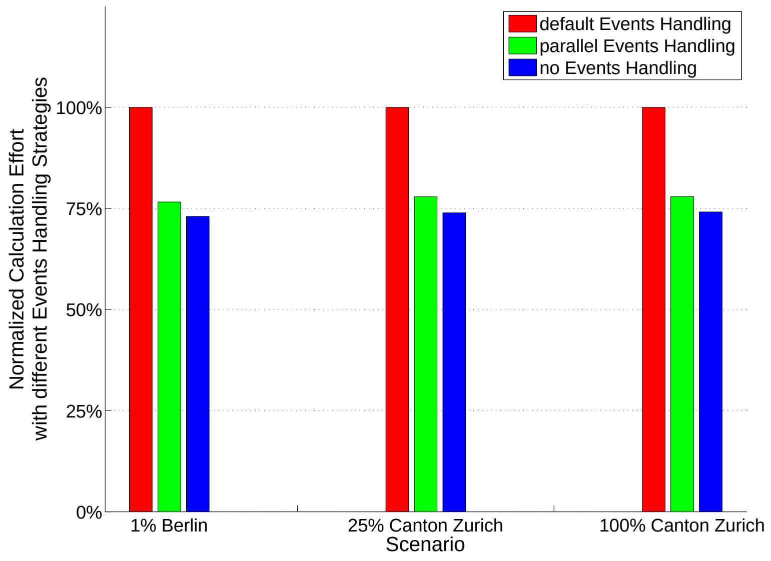
These real world scenarios represent scenarios that are typically simulated with MATSim. It can be assumed that the results of the performance measurements can be reached on other, comparable scenarios as well.

6.3 Results

The results of the simulations runs that are discussed in this section are only analyzed regarding the performance of the used simulation setup (queue simulation and events handling strategy), but no conclusions concerning the results from a traffic planning view are drawn.

Figure 9 shows the calculation effort of the events handling in the three scenarios. The results show, that the effort is 25% of the total calculation effort and is not influenced by the size of the scenario. According to Amdahl's Law (Amdahl, 1967), which describes the maximum achievable speedup of a programm with partially parallelized code, this affects significantly the reachable performance gain. 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. Thus, the events handling limits the possible speed gain to a factor of four related to the calculation time of the *QSim* with non-parallel events handling. Related to the runs of the *QSim* with parallel events handling a performance gain of factor three is possible.

Figure 9: Performance of different Events Handling Strategies



An important finding is the high efficiency of the *ParallelEventsManager* in combination with the *QSim*—almost the entire calculation effort of the events handling is moved from the main thread to a separated thread. As a result, the simulation is nearly as fast as it would be without any events handling.

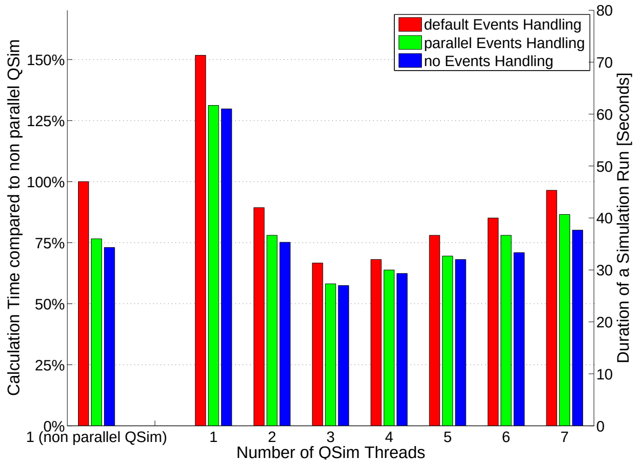
Figures 10(a) to 10(c) show the results of the runs with the three test scenarios. Each sub-figure contains the results of runs with the *ParallelQSim* using one to seven cores and different events handling strategies. Additionally, the same scenarios have been run with the non-parallel *QSim*. In a first analysis step, 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 like distributing and synchronizing data between the threads.

In the Berlin scenario a significant overhead of over 50% is found. As a result, using the *ParallelQSim* cannot reduce the calculation time significantly compared to the *QSim*. However, the *ParallelQSim* itself performs quite good. The calculation time reduces by 50% if three threads are used instead of a single one.

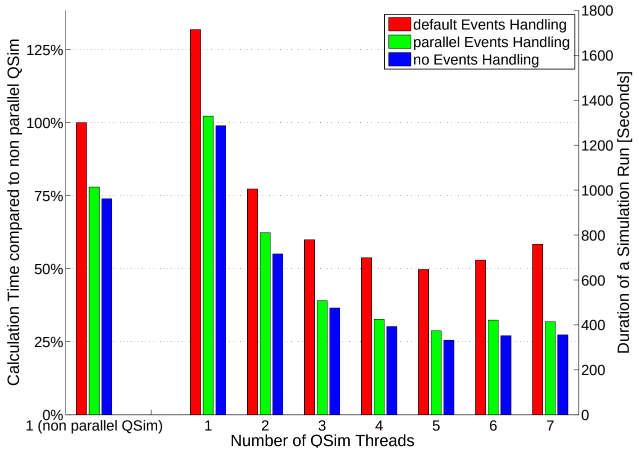
The Canton Zurich scenarios show that the calculation overhead is less significant if the scenario get more complex—the overhead reduces to 30% (25% scenario, using 5 threads) and 20% (100% scenario, using 6 threads). Therefore the performance gains rise up to the—

Figure 10: Results of the Sample Scenarios

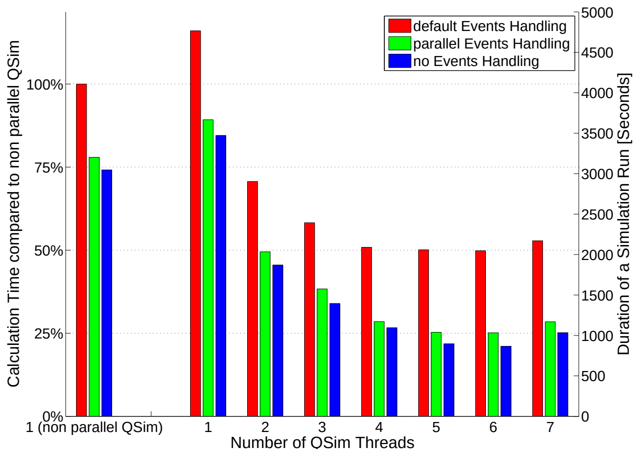
(a) Results 1% Berlin Scenario



(b) Results 25% Canton Zurich Scenario



(c) Results 100% Canton Zurich Scenario



according to Amdahl's Law—highest reachable value of a factor three when using parallel events handling.

Comparing the results of the Canton Zurich runs with parallel events handling and runs without events handling shows that there is still only a small difference in computational 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 bottleneck.

Considering only the results of the *ParallelQSim* shows that the number of cores used that 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.

7 Conclusion and Outlook

This paper describes the development and implementation of a new simulation module in MATSim that reaches short calculation times by using multiple CPU cores. Starting with a overview on the MATSim framework and its existing simulation modules, different simulation approaches are described. Analyzing these approaches regarding their suitability for parallelization leads to the conclusion that using the existing *QSim* as basement for the parallel queue simulation is most reasonable.

In the next step, various possibilities to distribute the calculation workload between the CPU cores are discussed. Using a simple implementation that distributes the workload evenly among the cores seems meaningful. The assignment of the network's links and nodes to the threads can be done randomly to avoid clustering of network areas with very high or very low traffic load. Using a scenario specific distribution that respects the expected load of the network would probably create slightly better results. A drawback of such a strategy is that it would require information about the network and the traffic volume which is usually not available in the required level of detail.

Having a detailed look on the implementation gives an insight into the structure of the *QSim* and shows how its concepts have been adapted in the *ParallelQSim*. Changes in the code that were necessary to keep the created simulation results deterministic have been described. Additionally some concepts are discussed which could be used to speed up the events handling.

The results of the performance tests show that—depending on the scenario size—the calculation time can be reduced by a factor of four. According to 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 also 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. In addition, two approaches to reduce the calculation time of the events handling have been discussed. A third alternative is using some time consuming events handlers that are only necessary for analyzes but not for MATSim runs itself not for every iteration of a simulation run. Typical examples for such handlers are plans and events file writers—in general it is sufficient to create those file only in every tenth iteration.

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 that results in the best simulation performance varies. Therefore, 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.

Even though 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 where further developments are preferable 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 a further performance gain.

References

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.

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

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.

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

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

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

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

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

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

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

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.

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

Michael, M. M. and M. L. Scott (1996) Simple, fast, and practical non-blocking and blocking concurrent queue algorithms, in J. E. Burns and M. Yoram (eds.) *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, 267–275, Association for Computing Machinery, New York.

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

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

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.

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