Diploma Thesis
Parallel Within-Day Replanning in Traffic Simulation

Author: Ch. Zwicker
Supervised by: M. Balmer
Professor: K. Nagel

03.11.2003 - 02.03.2004

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

In a modern society, traffic is an important part of everyone's life. An average Swiss person travels around 37 km per day, most of which ($\approx$ 25 km) by car. Together with traffic contributed by people passing through the country, this adds up to 110 billion ($110 \cdot 10^9$) km covered annually by passenger traffic. Adding the approximately 34 billion ton-km (distance multiplied by weight carried) of goods transport yields an annual traffic volume of around 144 billion km in Switzerland alone (data from the year 2000) [4].

These numbers - along with everyone's daily experience of traffic jams (statistical data reveals that it takes people 35.3 minutes to cover the 25 km by car mentioned above [4]) - show the importance of traffic planning. Traffic simulation can provide a scientific basis for improvements in traffic guidance systems or support optimality claims for existing solutions.

For traffic simulations to be accepted, results need to be as close to reality as possible. This means not only that traffic networks need to be modeled carefully, but also that as many parts of human behavior as possible should be taken into consideration when determining the behavior of the agents in the simulation. On the other hand, the more information is processed, the more complex the simulation packages tend to get.

1.2 Goals of this Diploma Thesis

The starting point for this diploma thesis is an existing traffic simulation package, which is described in detail in [2]. The three main goals of this diploma thesis concern the extension of that simulation package's features: (i) find out how much overhead results from exchanging information about the traffic situation between the different CPUs involved in a parallel computation, (ii) add the possibility of parallel within-day replanning, so that multiple replanning modules may be added to the simulation easily, and (iii) create two such replanning modules, namely (iii.a) a within-day route replanning module and (iii.b) a module which allows agents to drop an activity to which they will arrive late.

The first goal (i) shows the basic feasibility of within-day replanning. If the overhead resulting from the exchange of traffic information between the different CPUs is too high, it is not likely that within-day replanning is the way to go into the future. On the other hand, if the overhead is low, within-day replanning could be the way bring traffic simulations closer to reality and maybe also to decrease the number of iterations needed to get certain results. Thus the second goal (ii) of this thesis is to open up an easy way to add replanning modules to the simulation package.

In the existing simulation package (see [2]), agents which are stuck in a traffic jam just continue on their journey as planned at departure time. In contrast to that behavior, most

people would think about changing their route when they realize there is probably a faster path to their destination. The first replanning module to create during the course of this thesis is a route replanning module (iii.a), which should give agents the possibility to replan their routes while traveling. This within-day replanning is done based on the current travel times for different links, which are calculated from information about the current traffic situation. Up to now, simulations did only day-to-day replanning, which means that a certain number of agents were selected (randomly) at the end of a simulation and these agents were given new plans, with which the simulation was rerun.

People may not only realize they are running late due to a traffic jam, for example, but they could see they have no chance to arrive at their destination in time for the activity they want to carry out there. In this case, they might decide to reschedule their activities planned for the day. A simple way to do that is to just drop the next activity planned and to continue to the following one, which is exactly what is implemented in the activity-replanning module (iii.b).

1.3 Structure of this Document

Chapter 2 gives an overview of traffic simulation in general and also introduces some models used in the current simulation package. In chapter 3, the conceptual design of the new features added is explained. Chapter 4 shows more specific implementation details, while an introduction to the different third party products used in the current simulation package may be found in chapter 5. Results from the extensive testing of the application can be found in chapter 6.

Appendix A explains some commonly used terms. In appendix B, a user manual has been compiled as the basis for running and using the simulation package. A class overview for the new application is listed in appendix C. Finally, some general hints which would have been useful during the implementation of the new software have been added to appendix D.

2 Traffic Simulation Background

2.1 Simulation Models

Most early traffic simulations were based on cellular automata, where links are modeled by a number of cells. In each simulation step, agents were moved from the cell they were currently in to the next one - if it was free. Each link was processed from back to front, so the dynamics of traffic flow were preserved. The back draw of this type of simulation model is that it is fairly slow, as each agent's position on the link it is currently travelling on needs to be updated in each step.

The current simulation package uses a queue model, first introduced in [1]. The implementation is based on the simulation presented in [2]. In a queue simulation, links are modeled as first-in first-out (FIFO) queues. There is a free flow velocity associated with each link ($v_{free\_flow}(link)$) and when an agent enters, the time of its earliest arrival at the end of the link is calculated (based on the link's length $l(link)$ and free flow velocity). At each time step, the foremost agents in the queue will be checked. If the agent's destination link has space to accommodate it, the simulation first checks if the agent is ready to leave the link, which is the case if the current time $t_{current}$ is greater or equal than the agent's earliest arrival time. This means that an agent can leave the link if

$$t_{current} \geq t_{enter\_link}(agent, link) + \frac{l(link)}{v_{free\_flow}(link)}$$

where $t_{enter\_link(agent,link)}$ is the time the currently checked agent entered the link it is trying to leave. Each link's capacity, $capacity(link)$, free flow velocity and length are read from the network description file specific to the current scenario executed (the format of network description files is detailed in B.3).

There is a second constraint which says that only a certain number of agents can leave a link at a certain time step. This is modeled by associating a capacity with each link. The average number of agents which leave the link in each time step may not be greater than that capacity. This constraint is implemented most easily by the following expression:

$$P(agent\ leaves\ link) = \begin{cases} 1 & , \ if\ count < capacity(link) \\ rand < \lfloor capacity(link) \rfloor & , \ else \end{cases}$$

where $rand$ is a uniformly distributed variable selected from $[0, 1[$ and $count$ is the number of agents which already left the link in the current time step.

The big advantage of the queue model over cellular automata model is speed. There is a back draw, though: as space becomes free immediately when an agent leaves a link and as agents "automatically" move up immediately, backwards traveling kinematic waves are modeled incorrectly and jams resolve too quickly in this model. Tests have shown that this is not a real restriction, as overall results stay as close to reality as with simulations based on cellular automata, so the model is widly accepted.

2.2 Route (Re-)Planning

In the original simulation, routes are replanned only between two iterations of a scenario. A certain percentage of agents are allowed to replan their routes based on travel times computed from the events which occurred during the past iteration. Then, the next iteration is started, and so on. In a simulation with within-day route replanning, agents have the additional possibility of replanning their routes while still in the simulation.

Section 3.1 shows the conceptual design of the route replanning module added to the current simulation.

2.3 Activity (Re-)Planning

Each agent has a number of activities planned at the beginning of the simulation of a scenario. These activities were planned for the agents by an independent module. Explanations on how this might be done are given in [3]. In the original simulation, agents just drive from one activity to another, even if they arrive too late to still carry out the activity they have planned (for example, they might still drive to a shop at 20:00, even if the shop closed at 18:30). In a simulation with within-day activity replanning, agents try to find out if they are going to be late for an activity and if they think they are, they may replan their day, for example by dropping the next activity and continuing to the following one.

An activity dropping module has been implemented for the current simulation. The concept this module is based on is described in section 3.2. More complex day replanning could easily be added to the new simulation at any time.

3 Conceptual Design

In the new simulation, agents may decide their current situation is not optimal at any point of time. Based on this decision, they may be given the possibility to do some sort of replanning. There are various things an agent may wish to change, e.g. the path to its next activity or the places it lives and works.

To make this possible, the simulation is to be extensible by various replanning modules. For this diploma thesis, one module was developed to allow route replanning and another one lets an agent change its day by dropping a planned activity and continuing straight to the next one.

As to the point of time when agents should be allowed to replan, there are basically two choices: (a) let them do it as soon as they decide they want to or (b) allow them to replan at specified intervals only. The former has the advantage that it is closer to reality, while the latter one allows for faster parallel execution of the simulation. Each module designer may choose which one is better suitable, but the recommended way is to use the second possibility, which was done for the route replanning and activity dropping modules as well.

Most replanning modules will need some kind of information about the current situation of the simulation in order to perform their task adequately. This information is provided in the form of events, which are distributed to all processes at regular (configurable) intervals. These events are then made available to all modules interested in processing them.

3.1 Within-Day Route Replanning

The first module implemented to show the correctness of the concept allows agents to replan their routes while on their way from one activity to another one. An agent will choose to replan its path if it decides it is going to be late for its next activity.

As an agent does not know its exact position on a link in a queue simulation, time checks should happen at intersections. To make this possible, an agent's route stores the ids of the nodes to cross, along with the time they should be passed. Each time an agent arrives at a node, it will compare the planned arrival time to the current time and then decide if it is running late.

If an agent notices its journey has not progressed as fast planned, it will decide it is late for its next activity with a probability which is proportional to the estimated late time divided by the late time tolerated for the next activity (this time is given in the initial plans file for each agent and activity. See section B.4 for details). For example, if an agent is late by 30 seconds and it may arrive 5 minutes late for the next activity with no further consequences, the probability that the agent will want to replan its route is $\frac{30s}{300s} = 10\%$.

Every so many seconds, a certain percentage of all agents which want to replan their routes are allowed to do so. Both the interval and the percentage are configurable (see section B.2). These agents call the routing subroutine, which will find a new fastest path based on the current travel times computed from the events which occurred up to the current point of time. All other agents will not remember they ever wanted to replan their routes, which is the same thing as if they had never decided they were late at all.

The routing subroutine has access to link travel times calculated from events which occurred during the running simulation. Route replanning requests will be served with routes computed based on these link travel times.

The module implements a strategy which supposes that traffic data for all streets in a certain region is available to any person at some cost. This means that agents have an improved navigation system which can request current traffic situation data at a certain cost per request. This means that navigation systems will ask for the data when the driver requests a new path to his or her destination.

3.2 Within-Day Activity Replanning

The second module implemented allows agents to drop an activity they had planned if they realize they will not reach it on time anyhow. Agents check their progress as described above (section 3.1) and then check if they can arrive on time for their next activity by comparing their estimated arrival time - which is the sum of their planned arrival time and their current late time - to the sum of the next activitiy's start time and the activity's late time tolerance.

Each time after a specific interval, a percentage of all agents which have decided they will be late for their next activity are allowed to drop that activity and continue straight to the following one (only if activity to drop is not the last one planned, which is assumed to be the agent's home location). Again, the interval as well as the percentage of agents allowed to replan are configurable.

Naturally, if an agent drops an activity, it will have to get a new path from its current location to the next activity it has planned to go to. Thus, the agent requests a new route by calling the routing subroutine just as it would have if it had replanned its route, only that the destination link is now the the link where the agent's next activity takes place.

It is of course possible to run the within-day activity replanning module independently from the within-day route replanning module. If the two of them are run together, though, it makes sense to let them do their work at the same intervals and to run the route replanning module first. The reason this is advisable is that due to route replanning, an agent may decide it will still be on time for its activity even if it would have been late had it noch changed its path.

4 Implementation

In the traffic simulation world, almost everybody works with C++ on some UNIX version, so using the same programming language under LINUX seemed to be the most sensible choice. This was further encouraged by the fact that the computing clusters I had access to also ran under LINUX. This choice of operating system again was motivated by the fact that stable and robust MPI implementations have been around for that system for a while now.

As a start of this diploma thesis, the original simulation was taken apart and reassembled with an object-orientated design in mind. The basic structure of the original simulation was largely preserved. The new features implemented as part of this thesis were added to that new version.

4.1 Microsimulation

In the current queue simulation, the network consists of two main parts: nodes and links. The links represent streets, and intersections are modeled by nodes. Each node may have multiple incoming and outgoing links, and each link has exactly one incoming and one outgoing node (links are directed, so more than one link may connect two specific nodes).

Links consist of four parts. The main part is (i) the queue, which is the actual representation of the street. When an agent arrives at the end of the queue and is ready to leave the link it will enter the (ii) outgoing buffer as soon as there is enough space available there. In each time step, vehicles may also want to enter the simulation coming from the (iii) car park. They are moved into a (iv) wait queue, which they will leave going to the outgoing buffer in the same order as they entered. These agents also have to wait for the outgoing buffer to have enough free space, but they are at the further disadvantage (compared to the agents coming from the queue) that they do not have the priority. Figure 4.1 may help in thinking about the parts a link consists of.

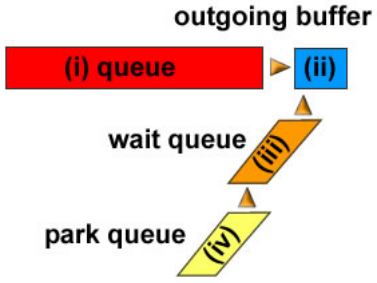


Figure 4.1: Parts of a Queue Simulation Link

The basic microsimulation runs in two essential steps, which are intersection and link movement. During intersection movement, for each node, all incoming links are checked for agents

ready to cross the intersection. For each link, agents are allowed to pass over the node as long as each one of them can enter the link it wants to go to.

During the link movement phase, as many agents as possible are moved into the outgoing buffer. Then agents are moved from the park queue into the wait queue and from there into the outgoing buffer if there is space left.

4.2 Support for Parallel Execution

The current simulation package provides support for parallel execution by using MPI. The entire simulation is parallelized by distributing the nodes and links of the street network over the available CPUs (partitioning is done by using METIS - see section 5.4). Nodes are distributed directly according to the mapping METIS returns. Links are assigned to CPUs so that links are processed on the same CPU as their destination node (for illustration, see figure 4.2).

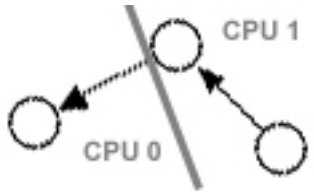


Figure 4.2: Parallel Distribution of Nodes and Links

When the program executes, each CPU only has to compute actions of the agents which are on links or nodes it owns. To make sure this works, some communication between the CPUs has to be added. For one, CPUs need to exchange information about how many spaces are free on links which are located on the border between two CPUs. This happens in `Parallelism::exchangeNumSpaces`: each CPU sends the number of free spaces on links directed toward it (other CPUs will want to send agents onto these links) and receives equivalent data for outgoing links (the current CPU will need to check if these have enough space to accommodate agents it wants to send over).

As already noted above, CPUs will need to send agents from one to another. This happens in two parts:

1. In `Parallelism::moveAgent`, origin and destination CPU of an agent allowed to cross an intersection are compared. If they are not the same, the agent is packed into an array of char and thus prepared to be sent by MPI (this happens in `Agent::pack`). If they are, the agent is simply put onto its destination link.

2. `Parallelism::moveAgents` actually exchanges agents between the different CPUs. On the destination CPU, each agent is recreated and its features are restored by using `Agent::unpack`. Then the agent is put onto its destination link.

The addition of within-day replanning to the simulation leads to the further requirement that information about all events triggered by agents should be known to the replanning modules even if these events have occurred on a different CPU. This means that events need to be communicated to all CPUs every so often.

The fastest way to distribute information among a number of processes (in a system where native broadcast or multicast are not available) is to send it along the edges of a hypercube with dimension $\lceil \log_2(\#\ of\ CPUs) \rceil$. As an example, let's say there are eight CPUs involved in a computation and they all need to communicate information to all others. Say CPU 0 has information A, CPU 1 possesses information B, and so on. In the end of the communication phase, all CPUs thus need to be able to access all of A, B, C, D, E, F, G, and H.

As mentioned above, the communication will proceed along the edges of a hypercube of dimension $\lceil \log_2(\#\ of\ CPUs) \rceil$, which is $\lceil \log_2(8) \rceil = 3$ in this case (which means the hypercube is a regular cube in this example). Figure 4.3 shows the communication paths along the 3D hypercube for this example, while table 4.1 lists the information available to the different CPUs after each phase of the cummunication.

This communication is implemented in four parts in the simulation. At setup time, any module interested in receiving information about the events which occur in the simulation should (i) call `Parallelism::registerEventsListener` (the class must be a subclass of `EventListener`). During the simulation process, (ii) events are registered by calling `Parallelism::sendEvent`. After a certain interval (*currentEventsInterval* defined in the configuration - see section B.2), (iii) the events are sent to all CPUs in the way described above when `Parallelism::communicateEvents` is called. Finally, (iv) all events are distributed to the individual modules calling their implementation of `EventListener::processCurrentEvent`.

\ CPU	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Start	A	B	C	D	E	F	G	H
Information Exchange with	1	0	3	2	5	4	7	6
Result	AB		CD		EF		GH	
Information Exchange with	2	3	0	1	6	7	4	5
Result	ABCD				EFGH			
Information Exchange with	4	5	6	7	0	1	2	3
Result	ABCDEFGH							

Table 4.1: Information Exchange along a Hypercube

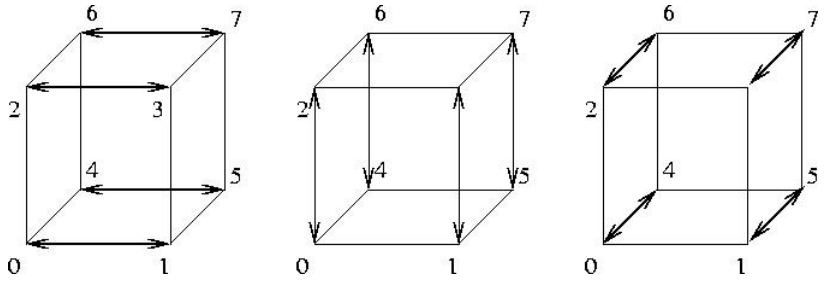


Figure 4.3: Information Exchange along a Hypercube

4.3 Integration of Replanning Modules

Each type of replanning module is called by the agents in the simulation whenever an agent is given the possibility (for example every X simulation steps) and has decided that it would profit from that specific kind of replanning. Each replanning module can obtain access to the events which occur during the run time of the simulation - by registering as an `EventListener` as described above - and process them according to its needs.

4.3.1 Within-Day Route Replanning

One part of the addition of within-day route replanning to the simulation was the extension of the `Agent` class to have each agent check if it is running late by comparing the current simulation time with the time it had planned to arrive at an intersection whenever it crosses one. If it has arrived later than it had planned to, the agent may decide to replan its route with a probability proportional to its late time divided by the late time tolerated by the activity the agent is going to. This late time tolerance is defined for each activity, its value is read from the plans file (see section B.4) if it is available. If it is not, its value is set to the default of 300 seconds.

The route replanning module makes use of the class `Router`, which extends `EventListener` and processes events in order to compute current link travel times. Every X time steps, Y% of all agents which have decided they are running late are allowed to replan their routes. Both X and Y are user-configurable values (see section B.2 for more information). An agent which is allowed to replan its route will call `Router::reroute` to obtain a new path to its next activity.

4.3.2 Within-Day Activity Replanning

The activity replanning module is very simple in design, all it does is to drop activities if an agent decides it will not arrive on time to the next activity planned. The implementation of this module is thus a very simple addition to the `Agent` class. Whenever an agent crosses an intersection, it will check if it can still arrive on time for its next activity. This is done by adding the agent's current late time (computed by comparing planned arrival time at the intersection and current simulation time) to the time the agent has planned to arrive at the

last intersection before it arrives at its next activity. If the agent sees that it will arrive later than the next activity tolerates (this value is read from the agent's plan, see section B.4 for reference), the agent will decide it is too late for that activity.

As in route replanning, every X time steps, Y% of all agents which have decided they are going to be too late for their next activity are allowed to replan their activities. Both X and Y are user-configurable values (see section B.2 for more information). They do not have to be the same as for the route replanning module, but in most cases it probably makes sense to set them to the same value.

The activity replanning module is also integrated with the route replanning module. If both modules are activated, an agent wich is allowed to replan its route may get a route which will not allow it to reach its destination on time. The agent will then immediately decide it is going to be too late and may thus later be allowed to replan its activities. On the other hand, an agent may already have decided it is going to be too late for its next activity, but get a better path to it if it is allowed to replan its route. If this is the case, the agent will revise its decision to replan its activities and just follow its new route to its next activity as originally planned.

4.4 Route Planning at Simulation Start

The `Router` class mentioned above has an additional use. When the simulation starts, agents and their plans are imported from a file (for the file description, see section B.4). In the traditional aproach, the routes between each pair of activities are given in that file. These routes are adapted for some agents between two iterations of a simulation by an independent module. As a routing subroutine is already available, there is no real need to provide routes between activities, these routes can be generated at the beginning of the simulation, based on the free flow speed associated with each link (this information is available in the network description file - see section B.3 for reference).

Further on, there is no need to have an independent module do the route replanning between two iterations of the simulation. If a file containing events from a previous iteration is specified in the configration file (refer to section B.2), these events are distributed to all registered subclasses of `EventListener` (see above). The router (which actually does extend the mentioned interface-class) receives these "historic'" events and calculates link travel times for a number of different points of time in the simulation (The entire time for which events are known is divided in time bin of the size of the event communication interval specified in the configuration file. For each of these time bins, travel time information is stored for all links in the network). At the beginning of the simulation, a certain (configurable) percentage of all routes may be computed based on the historic travel times stored by the router.

5 Third Party Products Used

5.1 Standard Template Library

The current implementation of the simulation package makes heavy use of the following container classes provided by the Standard Template Library (STL - see [5]):

	access time	**sorted?**	**implementation**	**STL extension**
vector	O(1)	no	array	no
map	O(log N)	yes	RB-Tree	no
multimap	O(log N)	yes	RB-Tree	no
hash_map	O(1)	yes	hash-table	yes
slist	O(n)	no	linked list	yes
list	O(n)	no	doubly linked list	no

Table 5.1: STL Containers Used

As listed above, hash_map and slist do not belong to the original set of STL containers, but were added as an extension. The header files for this extension are located at different places depending on the compiler version used: for g++-2.X, they are in the usual search path (e.g. include slist as `<slist>`), for g++-3.X, they are in ext (e.g. include slist as `<ext/slist>`). In addition to the usual includes, "`using namespace __gnu_cxx`" needs to be added for g++-3.X compilers.

If one plans to use std::string with STL extensions, it is a good idea to add the following code snippet:

```
template<> struct hash<std::string> {
  size_t operator()(const std::string &x) const {
    return hash<const char*>()(x.c_str());
  }
}
```

For g++-3.X, surround the above by `namespace __gnu_cxx { }`.

5.2 Expat

Expat is an XML parser written in C [6]. It is used for parsing configuration files, network descriptions, and plans files, which are all encoded in XML. The parser allows the programmer to specify handlers for element start and end. Thus the programmer gets one element at a time, together with its attributes, and has to take some action based on the element's type, name and attributes.

There is another approach to XML handling, which is based on the Document Object Model (DOM). Parsers using this approach parse the entire XML file once and create an internal representation of the document. Then, the programmer can traverse that tree and get the required elements.

The advantage of the first approach is the much smaller memory requirement, as the XML can be parsed part for part and no special internal representation has to be built. The disadvantage is that the code tends to get messier and that some additional class-global variables are needed to store information from earlier calls to the element handlers. As there are huge XML files used in the simulation program, the first approach was chosen in the final implementation.

5.3 MPICH

The MPI-implementation used for parallel execution of the simulation was MPICH [7]. The implementation is freely available for multiple operating systems and distributed as source.

5.4 METIS

METIS [8] is a set of graph partitioning algorithms, used to distribute the network over the available CPUs in a parallel execution. In the current simulation, there are two algorithms used, one in case there are 8 or more CPUs available, and the other one in case there are not.

Both methods take as input the following arguments (note that idxtype is of type int for the current simulation):

- **int *n:** Number of vertices in the graph
- **idxtype *xadj:** Starting indices to adjncy for each vertex, last entry is the final index
- **idxtype *adjncy:** For each vertex i, a list of connected vertices, starting at `xadj[i]`, ending at `xadj[i+1] - 1`.
- **idxtype *vwgt:** Weights of the vertices, NULL if all weights are equal
- **idxtype *adjwgt:** Edge weights, NULL if all weights are equal
- **int *wgtflag:** Indicates which weights are to be used
    0. No weights
    1. Weights on edges
    2. Weights on vertices
    3. Weights on edges and vertices

- **int *numflag:** Numbering scheme: use 0 for arrays starting at index 0
- **int *nparts:** Number of CPUs available
- **int *options:** Some options, just use defaults: `options[0] = 0`)
- **int *edgecut:** Number of edges connecting different CPUs
- **idxtype *part:** Assignment of the vertices to CPUs

As an illustration of how `xadj` and `adjncy` interact, here is an example from the metis manual. For the following graph, xadj and adjncy would be as listed below:

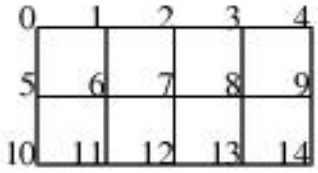


Figure 5.1: Metis Data Structures: xadj and adjncy

xadj: 0 2 5 8 11 13 16 20 24 28 31 33 36 39 42 44
adjncy: 1 5 0 2 6 1 3 7 2 4 8 3 9 0 6 10 1 5 7 11 2 6 8 12 3 7 9 13 4 8 14 5 11 6 10 12 7 11 13 8 12 14 9 13

6 Results

6.1 Hardware Used

All tests were conducted on a beowulf cluster consisting of 32 nodes connected by myrinet [9]. The individual nodes are dual-CPU machines, the individual CPUs stepped at 1GHz, with 1GB of RAM shared between them. There are two series of results, one with only one CPU used per node, the other one with both of them used.

6.2 Networks and Scenarios

The different tests were run mainly for four scenarios on three different networks. Some scenarios were used to verify the correct working of the program (corridor and triangle), the others were used to run performance tests (gotthard and switzerland).

6.2.1 Corridor

The corridor scenario is very simple. The network used is displayed in figure 6.1. The scenario contains 6000 agents driving from the leftmost six nodes to the rightmost three nodes in the network (which contains 24 nodes and 41 links). This scenario was used to test the within-day route replanning capability added to the simulation package. The network is displayed in figure 6.1

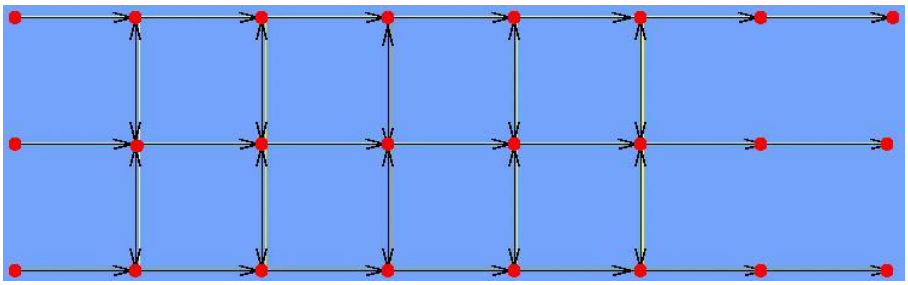


Figure 6.1: Corridor Network

6.2.2 Triangle

This network was created specifically to test the new activity replanning functionality added. Agents drive counter-clockwise around the triangle (links are unidirectional) and their activities take place at each corner of the triangle (one corner after the other). There are additional links connecting each side of the triangle with the opposite corner (leading only toward that corner). At each corner, 43 agents start their journey which is planned to lead them three times around the triangle. Figure 6.2 shows what the network looks like.

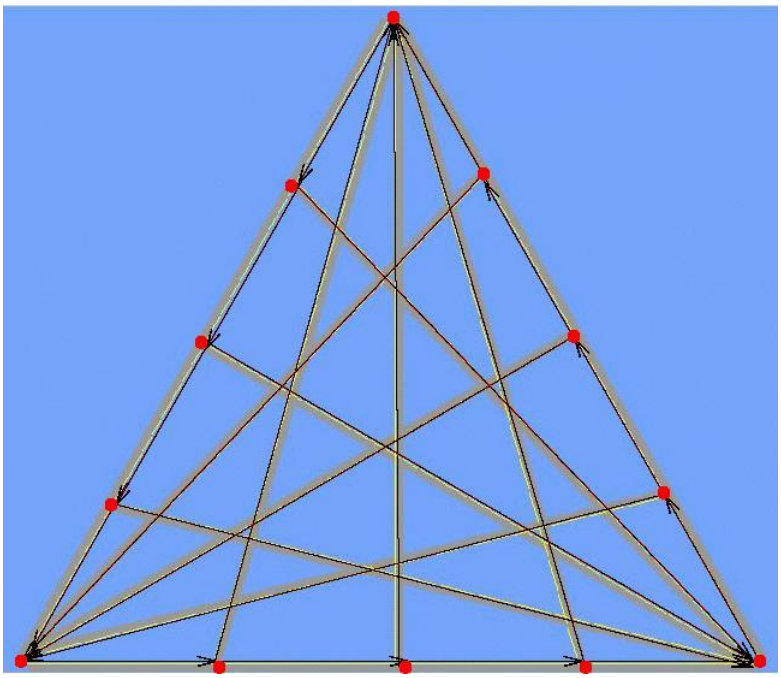


Figure 6.2: Triangle Network

6.2.3 Switzerland Network

The switzerland network is a medium resolution network covering all of Switzerland as well as some parts of neighbor countries in very low resolution. The network contains 10,564 nodes and 28,624 links. Figure 6.3 displays the network and figure 6.4 shows the part of the network which is relevant to the simulation. Both the gotthard and the switzerland scenario execute on this network.

6.2.4 Gotthard Scenario

In the gotthard scenario, 49,998 agents drive from all parts of Switzerland towards the Ticino and pass through the Gotthard tunnel. The network the agents drive on is not very high resolution (see section 6.3), but the resolution seems to be sufficient to get close to reality results.

6.2.5 Switzerland Scenario

The switzerland scenario simulates the situation in the morning, when everybody is driving to work between 6 am and 9 am. 991,471 agents from all over in Switzerland drive to their work locations which are situated at many different locations as well.

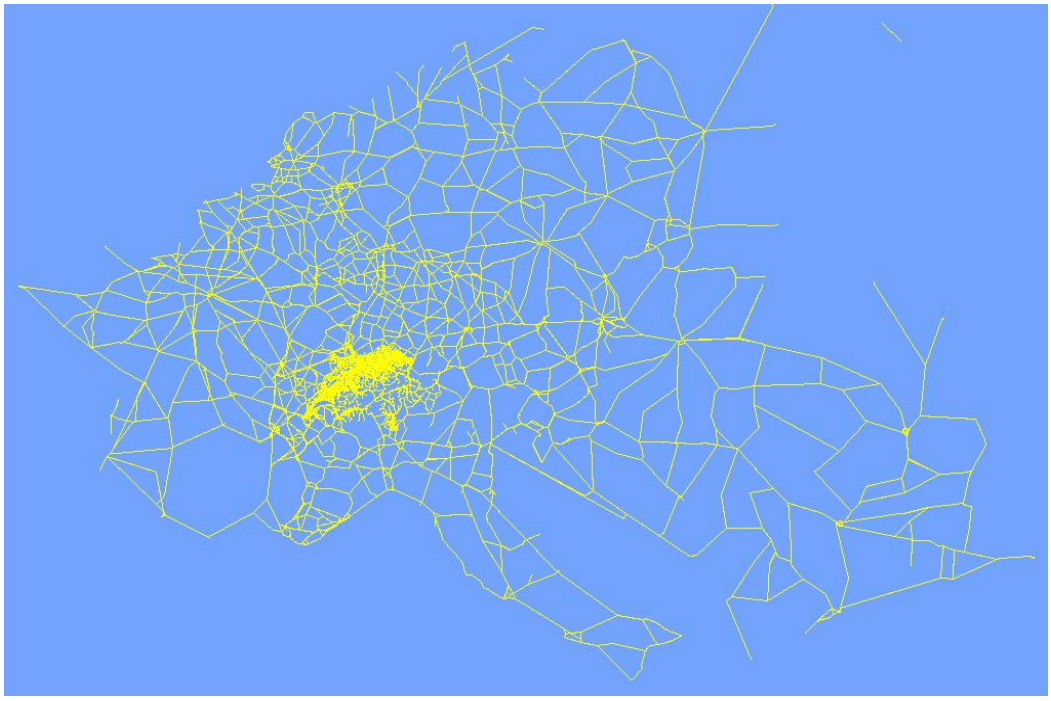


Figure 6.3: Switzerland Network

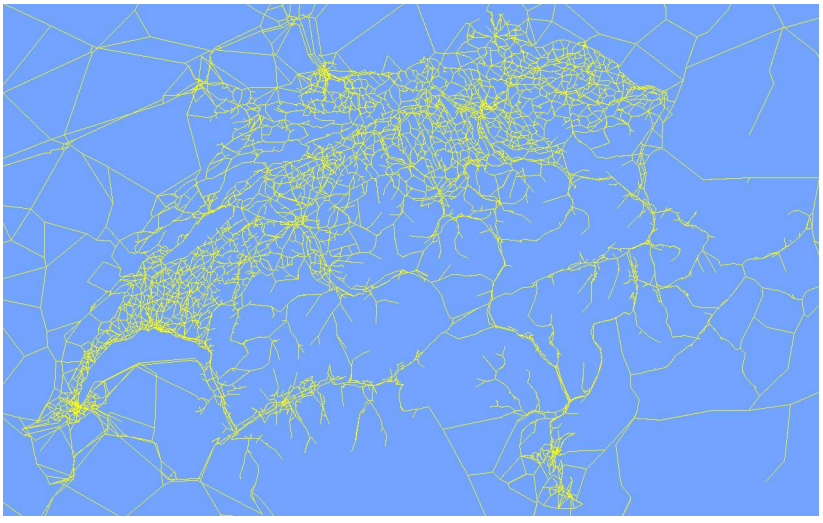


Figure 6.4: Switzerland Network Detail

6.3 Verification

6.3.1 Basic Implementation

The correctness of the basic new simulation (for both the sequential and the parallel case) was verified by comparing event output for the corridor scenario (see section 6.2.1 with the output from the original simulation. As the new simulation introduced more locations where a random number was drawn and writes events output in a different format, a back-compatibility mode was introduced to allow easier comparison between the two simulations. In that mode, random numbers are only drawn where this happens in the original simulation and all kinds of replanning are disabled to have agents follow the same routes they would have followed in the first iteration of the original simulation.

6.3.2 Events Communication & Route Replanning

The next step was to verify the correct distribution and processing of events across CPU borders. This was done by running the corridor scenario (described in section 6.2.1) with a predefined distribution of nodes (and links) across CPUs in a special manner (see section 6.1). This distribution made sure that a traffic jam on links 13, 18 and 23 would only make agents reroute to links 06, 10, 11, 15, 16, 20, 21 and 25 if events were communicated correctly.

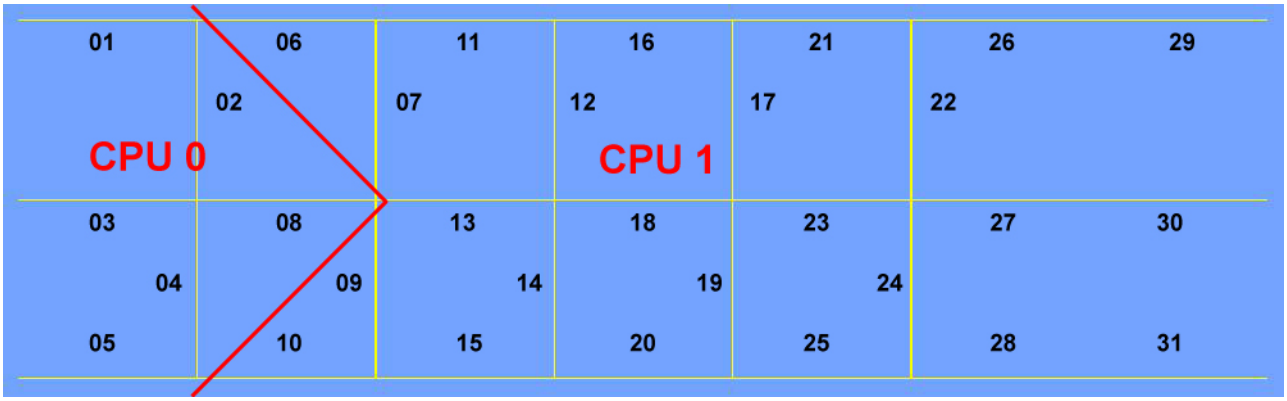
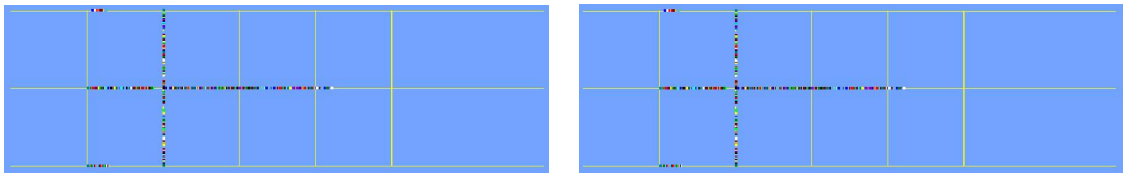


Figure 6.5: Corridor Network Subdivision (Replanning Verification)

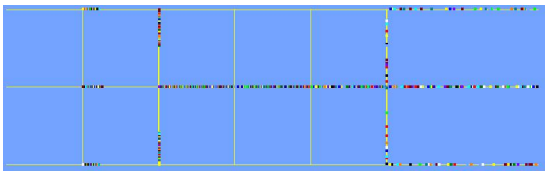
In addition to that, the run also proved the correctness of the rerouting routine and showed that within-day rerouting makes sense: Instead of waiting in traffic jams, agents drove on longer paths but arrived earlier at their destinations. This is shown nicely by the fact that in the same simulation the last agent left the network at 10:46:18 in the case without rerouting, but already at 09:16:01 if 10% route replanning was enabled. In images 6.6 the states of the simulation at different times is displayed. In these images, it is clearly visible that many agents chose to reroute because their initial route was slower than they thought. Please note that agents' positions on links are not exact (in a queue simulation, exact locations are not known to agents). The current visualization mode allows for nearly exact positioning on many

links, but the draw back is that positions are very inexact on the first link they enter after their departure.

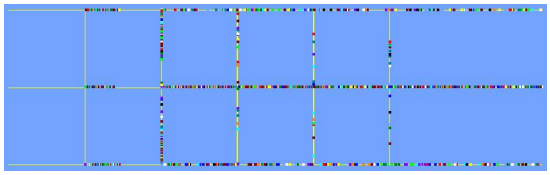


(a) 07:14, no replanning

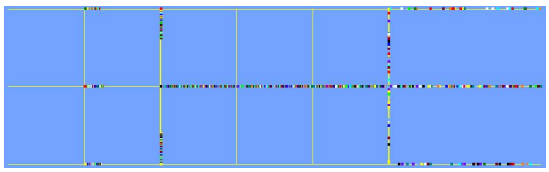
(b) 07:14, 10% replanning



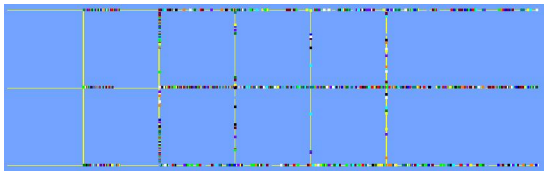
(c) 07:44, no replanning



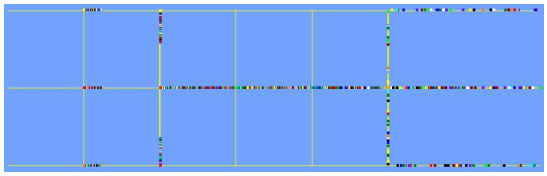
(d) 07:44, 10% replanning



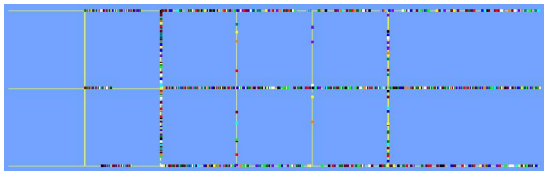
(e) 08:14, no replanning



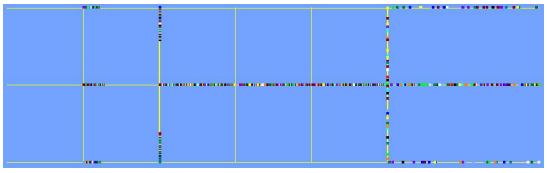
(f) 08:14, 10% replanning



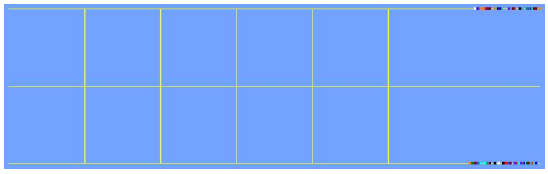
(g) 08:44, no replanning



(h) 08:44, 10% replanning



(i) 09:14, no replanning



(j) 09:14, 10% replanning

Figure 6.6: Corridor Scenario: Simulation States

6.3.3 Activity Replanning

One more feature was left for verification: activity replanning. The current version implements a very simple kind of day replanning, which is activity dropping. An agent which decides it will arrive late to its next activity can choose to continue directly to the following one in its plan. This was verified by running a special simulation on the new triangle network.

The triangle scenario (described in section 6.2.2) has 43 agents start at each corner of the triangle network. These agents have planned to go from one corner to the next one in counter-clockwise direction. The agent's time plan is very tight and some of them are bound to arrive late. In the scenario run, 10% of all agents which thought they would arrive late were allowed to replan their days. So if any agent chooses to drop an activity (which proves the algorithm works as it is supposed to) it will drive from one of the sides of the triangle directly to the opposite corner using one of the three links connecting the two. Figure 6.7 shows that some agents actually did replan their days and chose to leave out one of their activities!

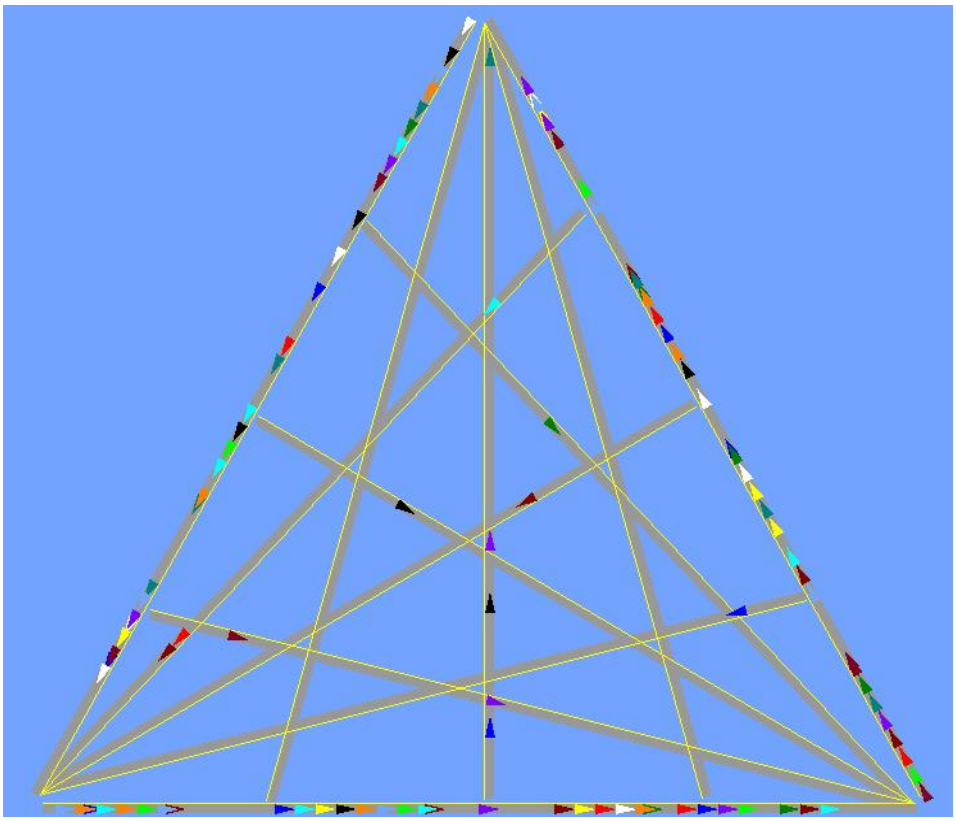


Figure 6.7: Triangle Network - Activity Dropping

6.4 Execution Speed

Execution speed tests were conducted by measuring wall clock time before and after the main simulation loop. Each simulation was run for the time between 6:00:00 am and 9:00:00 am. To be consistent with measurements from the original simulation, the first iteration of the simulation loop was discarded (measurements started at 6:00:01 am). All tests were performed for both the gotthard (see section 6.2.4) and the switzerland (see section 6.2.5) scenario.

Theoretical maximum speeds will be given for each type of simulation and each scenario. In practice, the speed of a simulation run with a certain number of CPUs also depends on the quality of the load balancing across the CPUs involved. This is the case because there are parts of the simulation where information has to be exchanged between CPUs involved in the computation. These parts can only be completed when information from the last CPU involved is received, so the maximum computation times of any CPU determines the time a simulation step takes to execute.

For the speed tests performed for the current simulation, no special load balancing was done, the network was always partitioned without knowledge about the situation during the simulation run. On the other hand, the results for the original simulation taken as a comparison are only available for a simulation where near-perfect load balancing was attempted by gathering link-load statistics during a first run and using these values as an input for the network decomposition algorithm for the test run. *Theoretical values are based on a perfect load balancing.*

Simulations distributed on a fixed number of CPUs run much faster if each CPU is on a separate node than if both CPUs of a node are used. At first, the reason for this seemed to be the fact that the two CPUs share their memory and thus there would be less memory around for each process to use. This may be the reason for the low performance in some cases where large simulations are run, but in smaller scenarios, this does not make any difference.

An alternative explanation for the slower execution time would be to assume that many collisions occur during the communication-intensive parts of the simulation and a big percentage of the packets sent are dropped, but this would mean that the time spent in communication would increase, not the calculation time. This does not happen substantially, the communication-parts of the simulation only increase by around 10%. On the other hand, computationally intense parts of the simulation increase by around 50% on average.

It is not easy to find out what could further contribute to the phenomenon, but there is one other possibility: even though myrinet uses less CPU time than other communication networks, network communication still puts some load on a system's CPUs. If only one CPU per node is used, all communication specific computation may be done by the unused processing unit. As soon as the second CPU is involved in the computation as well, CPU time which was available for computation before must be used for communication and the overall performance drops. It might be worthwile trying to run the simulation on a cluster consisting of nodes with 4 CPUs each, using only three of them for actual computation in the simulation.

6.4.1 Basic Simulation

The first set of tests was to see how fast the new simulation runs without any of the new features added. This was necessary both because the entire simulation was rewritten and also to lay a baseline to see how much the integration of the new features would slow down the simulation.

The theoretical minimum time a simulation takes for a given number of CPUs involved in the computation consists of both time spent doing actual computation $T_{comp}(N)$, and $T_{comm}(N)$, which is the time needed for communication between the different CPUs. This second time includes both actual network communication and the time to consolidate the information received with the existing local data.

For the basic simulation, $T_{comp}(N)$ is made up by the time used for link movement, which is $T_{comp}^{links}(N)$ and $T_{comp}^{nodes}(N)$, the time to compute agents' intersection movement. Both of these scale with the number of CPUs used (if the load balancing between them is perfect), so $T_{comp}^{links}(N) = \frac{T_{comp}^{links}(1)}{N}$ and $T_{comp}^{nodes}(N) = \frac{T_{comp}^{nodes}(1)}{N}$.

Each CPU needs to inform its neighbors of the number of empty spaces on the links crossing the border between them. The time this takes is $T_{comm}^{spaces}(N)$. Also, agents crossing a node connecting links on two different CPUs need to be passed over the network. This takes time $T_{comm}^{agents}(N)$. Both these terms are 0 for $N = 1$ of course as there is no communication necessary in the sequential case. As each CPU only has to communicate with it's neighbors - the number of which can be estimated by $\frac{2(3\sqrt{N}-1)(\sqrt{N}-1)}{N}$, where N is the number of CPUs involved in the computation, and which goes to an average of six for $N \to \infty$ - communication time stays approximately constant whatever the number of CPUs used in a simulation may be.

Summing it all up, the total run-time of a simution is at least

$$T(N) \approx \frac{T_{comp}^{links}(1)}{N} + \frac{T_{comp}^{nodes}(1)}{N} + par(N) \cdot \left(T_{comm}^{spaces}(N) + T_{comm}^{agents}(N)\right)$$

where $par(N) = \frac{sign(N-2)+1}{2}$, which evaluates to $\begin{cases} 0, & if\ N = 1 \\ 1, & if\ N > 1 \end{cases}$

Table 6.1 shows the results of the speed-tests for the **gotthard scenario** using one CPU per compute node. The maximum speed-up which can be observed for this simulation is at 17.79 for 31 CPUs used in the computation. The efficiency of the parallelization lies between 0.57 and 0.85 depending on the number of CPUs used.

CPUs	Time	RTR	speed-up	efficiency
01	249	043	01.00	1.00
02	146	074	01.71	0.85
04	089	121	02.80	0.70
08	047	230	05.30	0.66
16	028	386	08.89	0.56
31	014	771	17.79	0.57

Table 6.1: Gotthard, Basic Simulation, Single CPU / Node - Results

The numbers for the tests (displayed in table 6.2) with two CPUs per node look worse than the ones for runs where only one CPU is used from every node. The fact that that particular run has a very high efficiency compared to the runs with a different number of CPUs can only be explained by the far-from-perfect load balancing between the CPUs, which seems to be more even if the network is partitioned in 62 instead of maybe 32 or 16 parts. Still, it only takes seven seconds to execute the simulation on 62 CPUs, which means that the simulation runs 1540 times as fast as real time.

CPUs	Time	RTR	speed-up	efficiency
02	211	0051	01.18	0.59
04	109	0099	02.28	0.57
08	057	0189	04.37	0.55
16	033	0327	07.55	0.47
32	016	0675	15.56	0.49
62	007	1800	35.57	0.57

Table 6.2: Gotthard, Basic Simulation, Dual CPU / Node - Results

Figure 6.8 compares the minimum simulation time theoretically possible with the actual simulation times measured. Where simulations for both single CPU and double CPU per node were run, the difference between the time it took the simulation using two CPUs per node to execute is stacked on top of the time the simulation running on a single CPU per node took to complete.

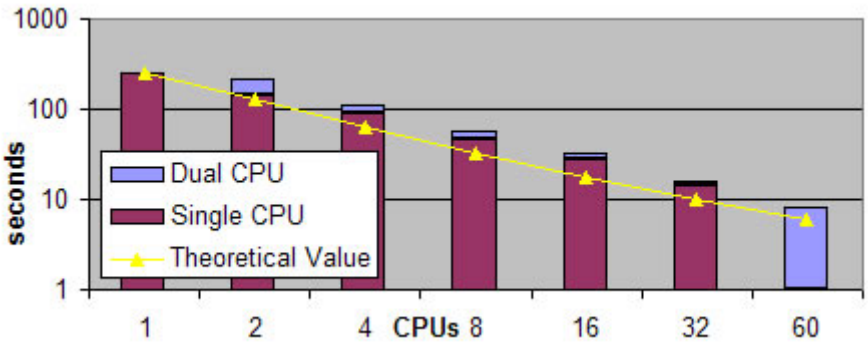


Figure 6.8: Gotthard, Basic Simulation - Speed-Up Chart

Table 6.3 shows the results for simulation of the **switzerland scenario**. The efficiency is generally lower than in the tests for the gotthard scenario. The reason is that there is much more computation to be done for this scenario, as there are 991,471 agents in the simulation instead of only 49,998 as in the gotthard scenario. This means that effects of poor load balancing are seen much earlier and more strongly.

For the **switzerland scenario**, there are results from the original simulation (see also [2] to which the new results may be compared. The new simulation runs much faster for one CPU and stays faster up to 8 CPUs involved in the computation. For a greater number of CPUs, the effects of the better load balancing for the original simulation are clearly visible, as the execution time decreases much faster than for the new simulation.

CPUs	Time	RTR	speed-up	efficiency	original time
01	387	028	01.00	1.00	2535
02	239	045	01.62	0.81	0850
04	133	081	02.91	0.73	0198
08	076	142	05.09	0.64	0100
16	055	196	07.04	0.44	0052
31	024	450	16.13	0.52	0025 (32 CPUs)

Table 6.3: Switzerland, Basic Simulation, Single CPU / Node - Results

As in the gotthard simulation, the **switzerland scenario** executed with both CPUs of each node involved, takes longer to execute than with only one CPU used per node (for the same number of CPUs). The maximum real time ratio for this simulation is at 514 for the run with 62 CPUs involved, which is also the only run for this scenario where the original simulation was faster than the new implementation. Table 6.4 shows detailed results for these tests.

CPUs	Time	RTR	speed-up	efficiency	original time
02	301	036	01.29	0.64	2817
04	165	065	02.35	0.59	1186
08	088	123	04.40	0.55	0140
16	064	169	06.05	0.38	0069
32	032	338	12.09	0.38	0034
62	021	514	18.43	0.30	0014 (64 CPUs)

Table 6.4: Switzerland, Basic Simulation, Dual CPU / Node - Results

Figure 6.9 plots the theoretically possible minimum execution time against the actual simulation times measured. The image again shows clearly that some overhead results from using both CPUs of each compute node.

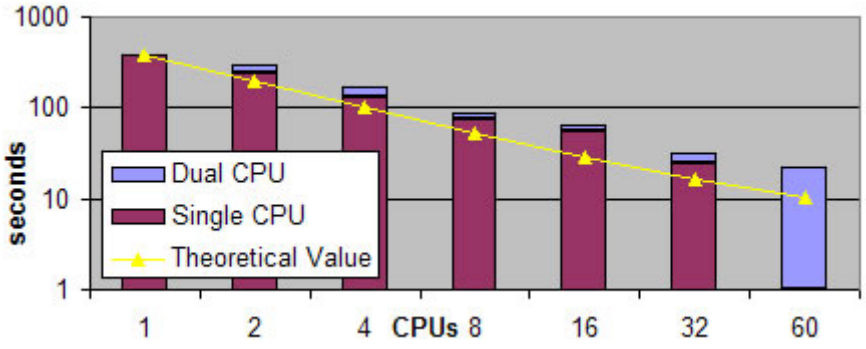


Figure 6.9: Switzerland, Basic Simulation - Speed-Up Chart

6.4.2 Events Communication

This test suite enabled events communication. Every 300 simulation steps, all events which had occurred in the previous interval were distributed to all CPUs and then sent to all interested modules for processing.

The maximum possible speed for this simulation is determined mostly by the new feature added. The time for the communication of events does not decrease if more CPUs are added, it even increases slightly, as a greater percentage of all events is non-local to every CPU. For example, if two CPUs are involved in a computation, the number of events local to each CPU will be around half the number of events which occurred in the last interval (assuming near-perfect load-balancing again), so the other half will have to be received from the other CPU. If four CPUs are involved in the computation, three quarters of all events will have to be received from other CPUs. For $N \to \infty$, the number of events communicated goes up to the total number of events which occurred.

The more expensive part of the events communication phase is the processing by the different modules receiving all events (in the current simulation, only the `Router` is interested in receiving events). This and the actual communication across the network are summed up in $T_{comm}^{events}(N)$, which is *not* equal to 0 for $N = 1$, as the events still need to be processed. As detailed above, the processing of the events is more expensive than the actual network communication. Measurements show that $T_{comm}^{events}(N)$ is around $1.5 \cdot T_{comm}^{events}(1)$.

It is important to note that both $T_{comp}^{links}(N)$ and $T_{comp}^{nodes}(N)$ somewhat increase as events need to be stored whenever they occur, so they can all together be distributed at a later point of time.

The sum of all these components is

$$\begin{aligned} T(N) &\approx \frac{T_{comp}^{links}(1)}{N} + \frac{T_{comp}^{nodes}(1)}{N} + par(N) \cdot \left(T_{comm}^{spaces}(N) + T_{comm}^{agents}(N)\right) \\ &+ (1 + 0.5 \cdot par(N)) \cdot T_{comm}^{events}(1) \end{aligned}$$

where $par(N) = \frac{sign(N-2)+1}{2}$, which evaluates to $\begin{cases} 0, & if\ N = 1 \\ 1, & if\ N > 1 \end{cases}$

In table 6.5, the results for the simulation of the **gotthard scenario** on a single CPU per node are detailed. As few events occur during the simulation of this scenario, only little time is spent distributing events (which happens only every 300 time steps). The additional time spent in computation (which results from storing events as they occur), so the results are very much similar to those for the simulation with events communication disabled.

CPUs	Time	RTR	speed-up	efficiency
01	270	040	01.00	1.00
02	159	068	01.70	0.85
04	093	116	02.90	0.73
08	050	216	05.40	0.68
16	032	338	08.44	0.53
31	018	600	15.00	0.48

Table 6.5: Gotthard, Events Communication, Single CPU / Node - Results

Table 6.8 shows the results for the gotthard simulation executed on two CPUs per node. For the reasons listed above, these results are almost the same as those for the simulation with no events communication.

CPUs	Time	RTR	speed-up	efficiency
02	215	050	01.26	0.63
04	115	094	02.35	0.59
08	062	174	04.35	0.54
16	038	284	07.11	0.44
32	020	540	13.50	0.42
62	012	900	22.50	0.36

Table 6.6: Gotthard, Events Communication, Dual CPU / Node - Results

Figure 6.10 shows how the measured results compare to the theoretically possible values. Where results for both single CPU and double CPU per node are available, the difference between the simulation times for two CPUs per node and one CPU per node is stacked on top of the time the simulation running on a single CPU per node took to complete.

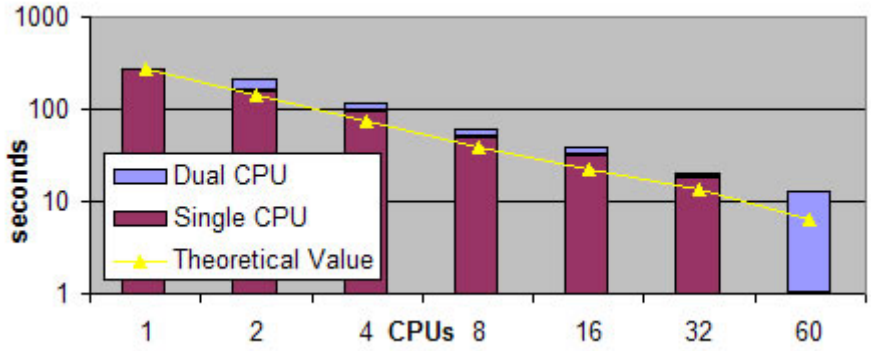


Figure 6.10: Gotthard, Events Communication - Speed-Up Chart

The results for the **switzerland scenario** (see table 6.7) are more interesting than those for gotthard as there are more agents in the simulation (991,471 compared to 49,998) and thus there are many more events to process. The measured efficiency is closer to the theoretically possible one for all simulations because the percentual difference between the time it takes the CPU with the maximum load and the one with the minimum load to complete is lower (both of them have to go through the events communication phase, so if the time they took to execute one step was e.g. 10ms and 40ms, and events communication takes maybe 20ms, it will now take them 30ms and 60ms respectively. Thus the difference between the two was 75% without events communication, but only 50% with it).

The addition of events communication to the simulation leads to an increased execution time for the simulation of the switzerland scenario. This increase is between as little as 3% if the simulation is run with one CPU only and as high as 112% if 30 CPUs are involved in the computation.

CPUs	Time	RTR	speed-up	efficiency
01	398	027	1.00	1.00
02	270	040	1.47	0.74
04	157	069	2.54	0.63
08	102	106	3.90	0.49
16	078	138	5.10	0.32
31	051	212	7.80	0.25

Table 6.7: Switzerland, Events Communication, Single CPU / Node - Results

Results for the simulation of the switzerland scenario on two CPUs per node are listed in table 6.8. Again, efficiencies are closer to the theoretically possible values than in the basic simulation, but also lower than in the case where only a single CPU per compute node is used.

CPUs	Time	RTR	speed-up	efficiency
02	339	032	1.17	0.59
04	203	053	1.96	0.49
08	138	078	2.88	0.36
16	101	107	3.94	0.25
32	071	152	5.61	0.18
62	051	212	7.80	0.13

Table 6.8: Switzerland, Events Communication, Dual CPU / Node - Results

Figure 6.11 plots measured execution times against minimal theoretical values. Measured values are closer to what is theoretically possible than for the basic simulation.

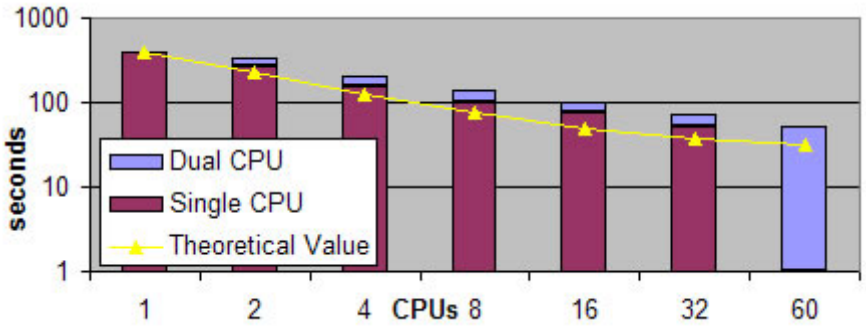


Figure 6.11: Switzerland, Events Communication - Speed-Up Chart

6.4.3 Route Replanning

The last test-suite was to test the performance of the system with 10% route replanning activated. The maximum theoretical speed is still mainly determined by the events communication - these events are needed by the route replanner to compute current travel times for all links in the network (events were still communicated every 300 steps) - but as there is more computation to be done on each CPU again, the influence of communication on the entire simulation speed is diminished a little. The time needed for route replanning, $T_{comp}^{routing}(N)$ scales linearly with the number of CPUs used (if the load balancing between them is perfect), so $T_{comp}^{routing}(N) = \frac{T_{comp}^{routing}(1)}{N}$.

The entire simulation now takes time

$$\begin{aligned} T(N) &\approx \frac{T_{comp}^{links}(1)}{N} + \frac{T_{comp}^{nodes}(1)}{N} + par(N) \cdot \left(T_{comm}^{spaces}(N) + T_{comm}^{agents}(N)\right) \\ &+ (1 + 0.5 \cdot par(N)) \cdot T_{comm}^{events}(1) \\ &+ \frac{T_{comp}^{routing}(1)}{N} \end{aligned} \tag{6.1}$$

where $par(N) = \frac{sign(N-2)+1}{2}$, which evaluates to $\begin{cases} 0, & if\ N = 1 \\ 1, & if\ N > 1 \end{cases}$

The results in table 6.9 again show clearly the effects of the bad load balancing between the different CPUs involved in the computation of the **gotthard scenario** simulation. The effect is even amplified because it is more likely that agents will replan in areas where there are already more agents around (if there are more agents on the same number of links, there will likely be more traffic jams).

CPUs	Time	RTR	speed-up	efficiency
01	2577	04	1.00	1.00
02	2152	05	1.20	0.60
04	1847	06	1.40	0.35
08	1021	11	2.52	0.32
16	0751	14	3.43	0.21
31	0511	21	5.04	0.16

Table 6.9: Gotthard, 10% Route Replanning, Single CPU / Node - Results

Table 6.10 shows results from simulations of the gotthard scenario where both CPUs of each compute node involved were used. As always, the efficiency of this type of setup is worse than if only one CPU per node was used.

CPUs	Time	RTR	speed-up	efficiency
02	2312	05	1.11	0.56
04	1873	06	1.38	0.34
08	1053	10	2.45	0.31
16	0819	13	3.15	0.20
32	0558	19	4.62	0.14
62	0330	33	7.81	0.13

Table 6.10: Gotthard, 10% Route Replanning, Dual CPU / Node - Results

For the simulation of the **gotthard scenario**, using both CPUs of a compute node is for the first time not much less efficient than using only one CPU per node. The efficiency of a parallel execution is rather low for both ways of running a simulation, though. Figure 6.12 shows how the measured execution speeds compare to the maximum speeds theoretically possible.

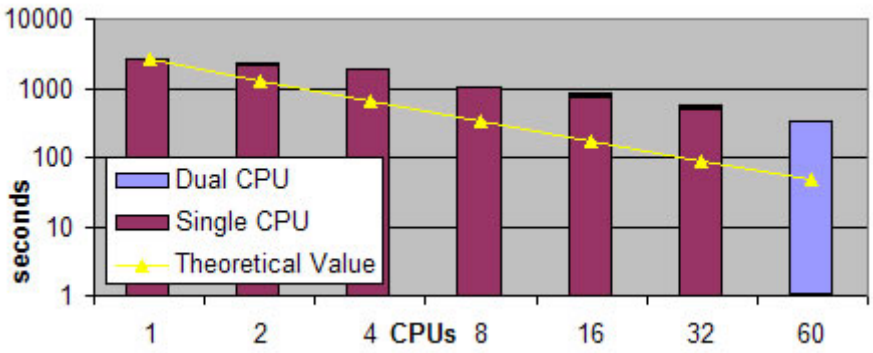


Figure 6.12: Gotthard, 10% Route Replanning - Speed-Up Chart

The speed-up for the simulation of the **switzerland scenario** stays far below what is theoretically possible, which shows how much influence the uneven load balancing between the CPUs has. The effect of the added route replanning on the execution time compared to simulations without route replanning is between 70% for one CPU and 25% for 31 CPUs. If compared to a simulation also without events communication, the effect is lower if fewer CPUs are involved in the computation: the simulation takes around 75% longer to execute if run on one CPU and almost 162% longer if the computation is distributed among 31 CPUs. Detailed results may be found in table 6.11.

CPUs	Time	RTR	speed-up	efficiency
01	659	016	1.00	1.00
02	489	022	1.35	0.67
04	274	039	2.41	0.60
08	190	057	3.47	0.43
16	163	066	4.04	0.25
31	102	106	6.46	0.21

Table 6.11: Switzerland, 10% Route Replanning, Single CPU / Node - Results

The simulation of the switzerland scenario with 10% route replanning enabled is only two seconds faster if 62 CPUs (both CPUs of each compute node used) are involved instead of 31 (where only one CPU is used on its node). This shows once more what a massive effect the bad load balancing has on the simulation speed. Figure 6.12 shows all results for the runs of the **switzerland scenario** on two CPUs per node.

CPUs	Time	RTR	speed-up	efficiency
02	545	020	1.21	0.60
04	356	030	1.85	0.46
08	243	044	2.71	0.34
16	174	062	3.79	0.24
32	149	072	4.42	0.14
62	100	108	6.59	0.11

Table 6.12: Switzerland, 10% Route Replanning, Dual CPU / Node - Results

Figure 6.13 shows the difference between the theoretically possible minimum execution time for the simulation of the switzerland scenario compared to the actual execution times for runs using one or both CPUs of each compute node available to the simulation. Note that using 62 CPUs in dual mode is not noticeably faster than using 31 CPUs where only one CPU per node is employed.

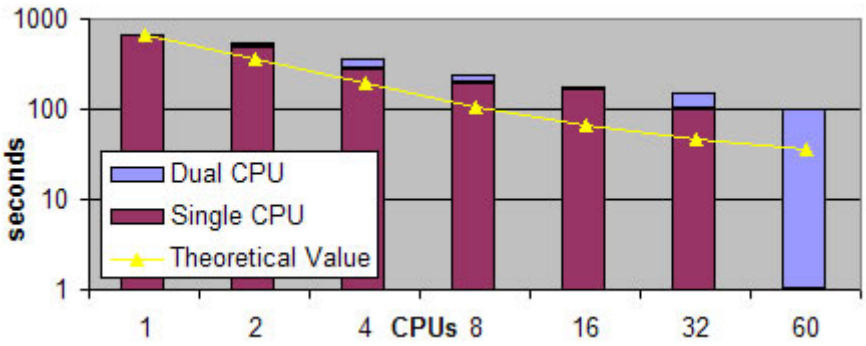


Figure 6.13: Switzerland, 10% Route Replanning - Speed-Up Chart

6.5 Effects on the Simulation

The influence of the added within-day route replanning was tested for the **gotthard scenario**. Simulations were run where 0%, 5%, 10%, and 25% of all agents which realized they were running late were allowed to replan their routes. Statistics on the arrival time and travel time of the agents in the simualation were calculated. Also, the locations where agents chose to replan their days were plotted. Further on, the locations of all agents in the simulation were printed every 60 minutes. Their paths can be traced and the effects of the replanning can be seen in terms of where agents drove through.

6.5.1 Travel Times

Travel times are greatly decreased for the average agent in the the **gotthard scenario** if within-day route replanning is enabled. For each travel time which occurred during the simulation, figure 6.14 shows how many agents took that time to reach their destinations. One can also see from this graph that even though most agents have shorter travel times if more agents are allowed to replan their routes, there are a number of agents which replan right when the main traffic jams start to dissolve, so these agents take longer paths than they would have to and thus arrive later than if they had not replanned at all.

The peak at about four hours of travel time is caused by agents arriving from north, east and west at the same time. Longer travel times are mainly caused by agents arriving from the north (principally Zurich area)

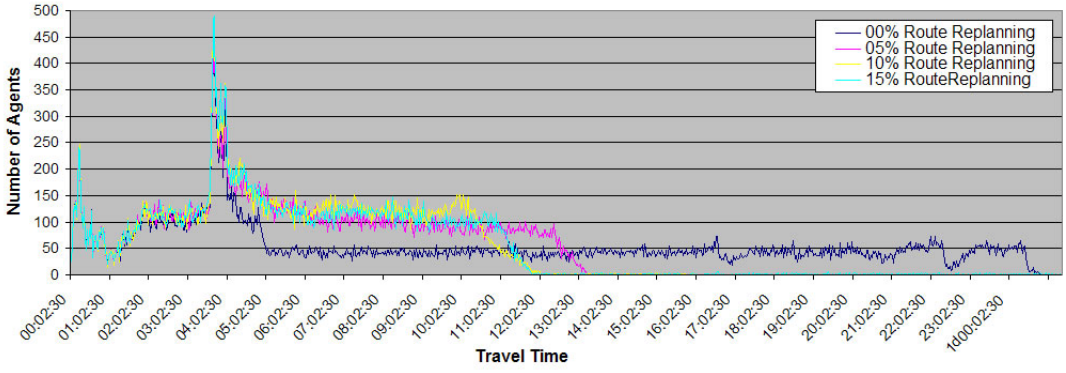


Figure 6.14: Gotthard, Travel Times

Figure 6.15 sums up travel times to show how many agents have a certain maximum travel time. It is visible from this graph that the average agent has much shorter travel times if route replanning is enabled (more agents have short travel times). Note that for 15% route replanning probability agents have longer average travel times than for 10% enabled. Additionally, there are now more than a few (around 1000) agents which have very long travel times, some of them even longer than in the simulation with no route replanning enabled. There are some agents which take quite a long time to arrive if 10% of all agents wishing to replan are allowed to, but their number is much smaller.

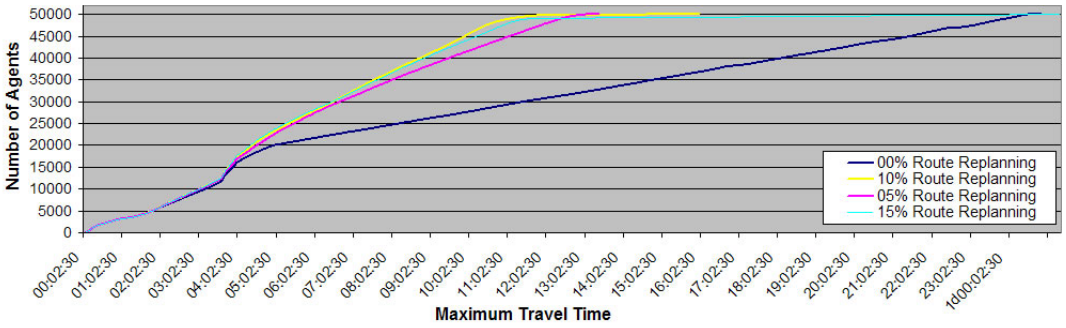


Figure 6.15: Gotthard, Aggregated Travel Times

6.5.2 Arrival Times

Figure 6.16 shows how many agents arrived at their destination activities at which times. The graph shows that for no within-day replanning enabled, some a great number of agents arrive at very late times, where most agents arrive before 8 pm when within-day route replanning is enabled. The figure also shows that a few agents arrive very late as well for higher replanning probabilities. These are agents which chose to replan their routes very late in the simulation and then chose a longer path to arrive at their destination. While they drove along that path which was computed to be the fastest way to their destinations, the traffic jams which had caused them to replan in the first place dissolved.

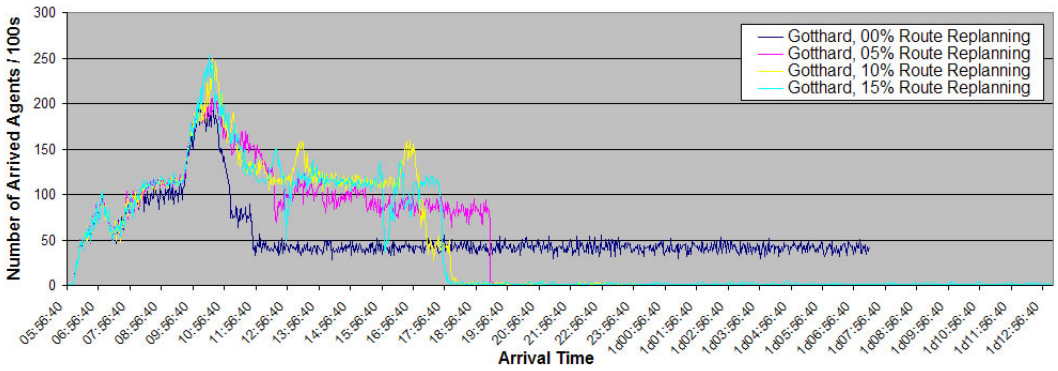


Figure 6.16: Gotthard, Arrival Times

In figure 6.17, the number of agents which arrived up to different points of time is displayed. The time it takes for 50% of all agents to arrive is around 2 hours and 45 minutes shorter if 5% of all agents willing to replan their routes are allowed to versus if no within-day route replanning is enabled. It even takes 8 hours and 18 minutes less for 75% of all agents to arrive if 10% of those willing may get new paths to their destination!

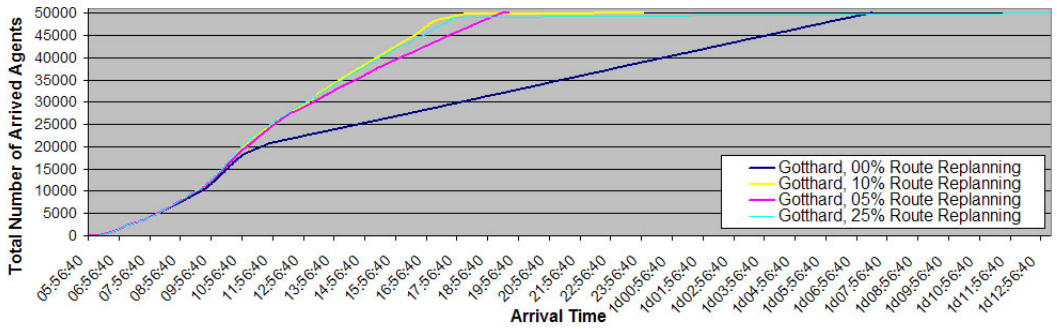
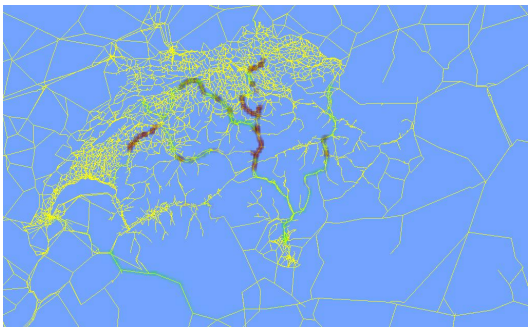


Figure 6.17: Gotthard, Aggregated Arrival Times

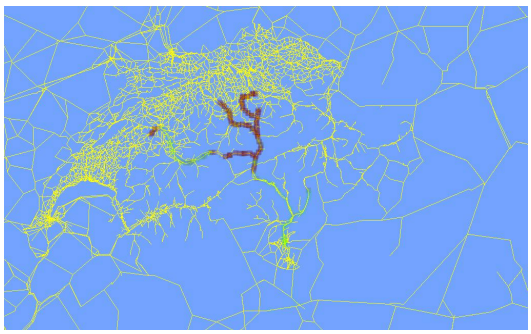
6.5.3 Route Replanning

Agents will decide they are late if they have been trapped in a traffic jam for a while. In the **gotthard scenario**, most agents will find themselves in the middle of a traffic jam at one time or other and thus most of them will decide they are running late. For all simulations with different route replanning probabilities (0%, 5%, 10%, and 15%), the locations where agents were allowed to replan their routes were recorded and visualized. The results for 15% route replanning suggest that probability to be too high to make sense for close to reality results, so these results will not further be taken into account.

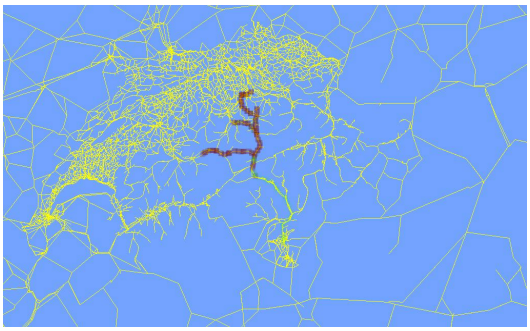
To be able to compare the route replanning locations with the prevailing traffic situations, these were also recorded. In graphics showing route replanning locations, places where more agents were allowed to replan their routes are drawn in darker colors, while places where fewer agents replanned are drawn in lighter colors. Visual data is available for 8 am, 11 am and 3 pm for all simulations. To compare the traffic situations in the simulations with route replanning enabled to the original situations, figure 6.18 shows the traffic situation for the three given times with route replanning disabled.



(a) Traffic Situation - 08:00



(b) Traffic Situation - 11:00



(c) Traffic Situation - 15:00

Figure 6.18: Gotthard Traffic Situation - No Route Replanning

Figure 6.20 shows the places where agents replanned in the **gotthard scenario** (different mark colors mean that a different number of agents replanned their routes - colors further right in figure 6.19 mean a greater number of replanning agents) as well as the traffic situation at the times when they replanned (5% of all agents which thought they were late were allowed to replan their routes). The traffic situation has changed from the way it was when no agents replanned their days and now paths are better distributed.

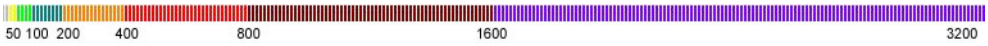
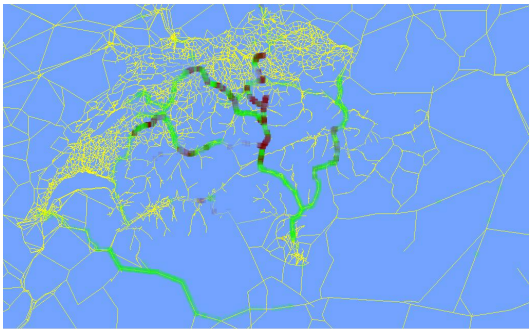
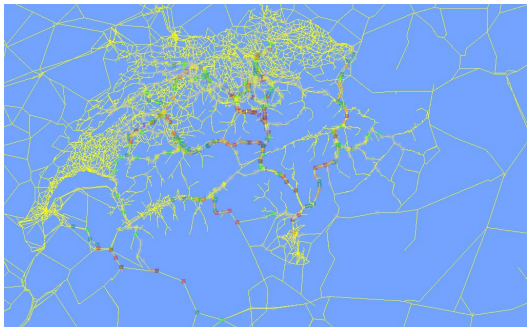


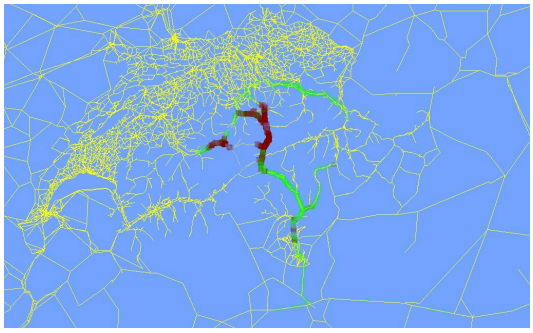
Figure 6.19: Color Index for Route Replanning Numbers



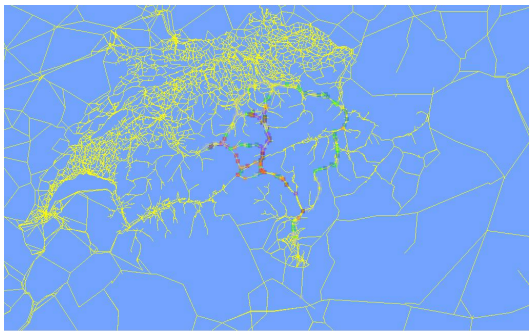
(a) Traffic Situation - 08:00



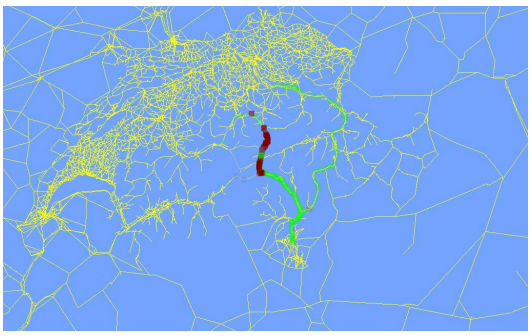
(b) Replanning Places - 07:30 to 08:30



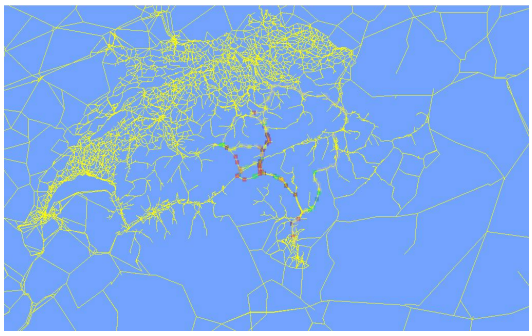
(c) Traffic Situation - 11:00



(d) Replanning Places - 11:30 to 12:30



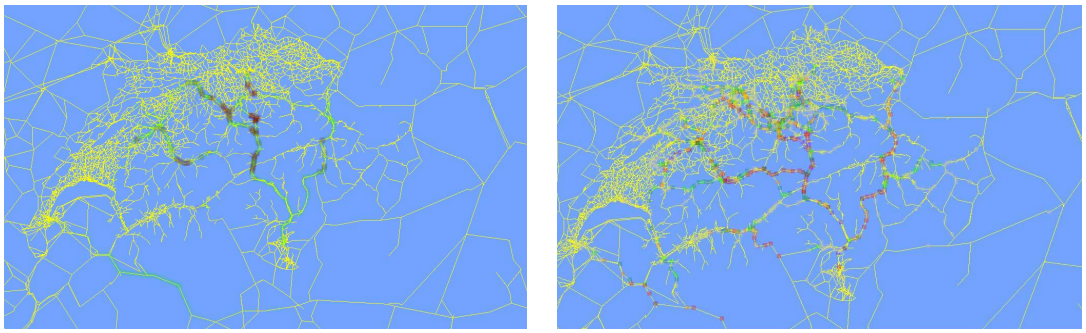
(e) Traffic Situation - 15:00



(f) Replanning Places - 14:30 to 15:30

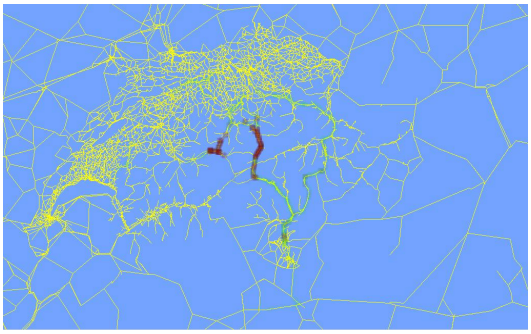
Figure 6.20: Gotthard Traffic Situation - 5% Route Replanning

Figure 6.21 also displays replanning places and traffic simulations, this time for a simulation where 10% of all agents willing were allowed to replan their routes. Note that even though there are again fewer traffic jams than in the case where only 5% of all agents were allowed to replan their routes, more agents are replanning.

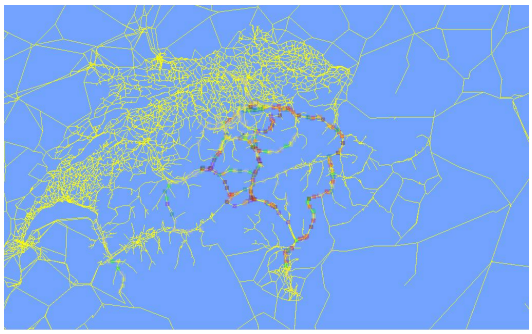


(a) Traffic Situation - 08:00

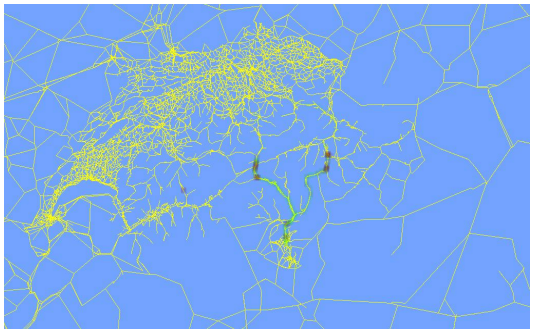
(b) Replanning Places - 07:30 to 08:30



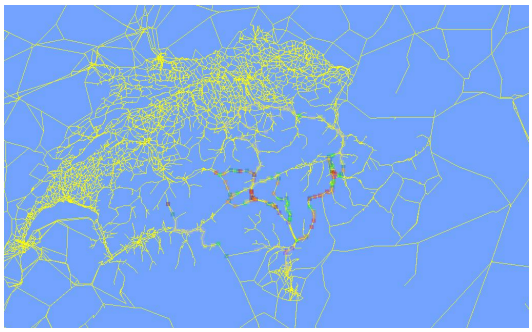
(c) Traffic Situation - 11:00



(d) Replanning Places - 11:30 to 12:30



(e) Traffic Situation - 15:00



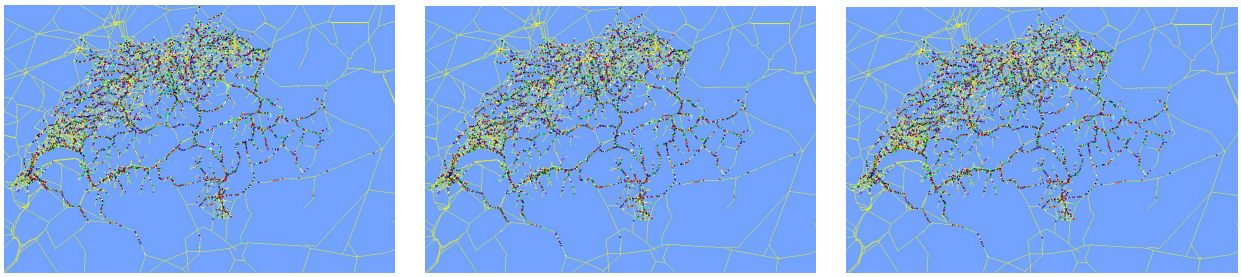
(f) Replanning Places - 14:30 to 15:30

Figure 6.21: Gotthard Traffic Situation - 10% Route Replanning

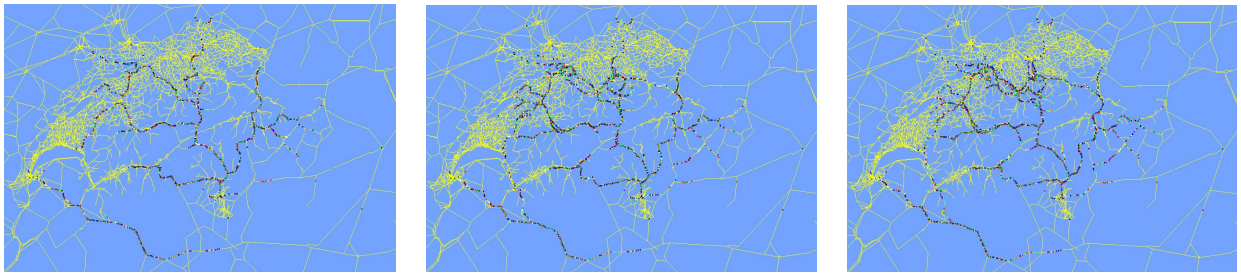
6.5.4 Agents' Paths

The different replanning probabilities lead to different paths for the agents. In the original simulation with no within-day route replanning, most agents drive through the Gotthard tunnel, even if there are traffic jams lasting hours. In the new simulation with route replanning enabled, many agents replan to drive through the San Bernardino tunnel, which saves them a lot of time. Figure 6.22 shows the approximate positions of all agents in the simulation from 7 am to 8 am. In figure 6.23 the traffic situation between 9 am and noon appears, while figure 6.24 shows the agents' positions every hour from 1 pm to 4 pm and figure 6.24 finally displays the agents' positions from 5 pm to 8 pm.

Big differences between agents' paths may be observed at 9 am between the simulation with no replanning and the runs with route replanning enabled. Comparing the agents' positions at 3 pm shows that if 10% of all agents willing to replan are allowed to, they advance faster than if only 5% of them may get a new route.

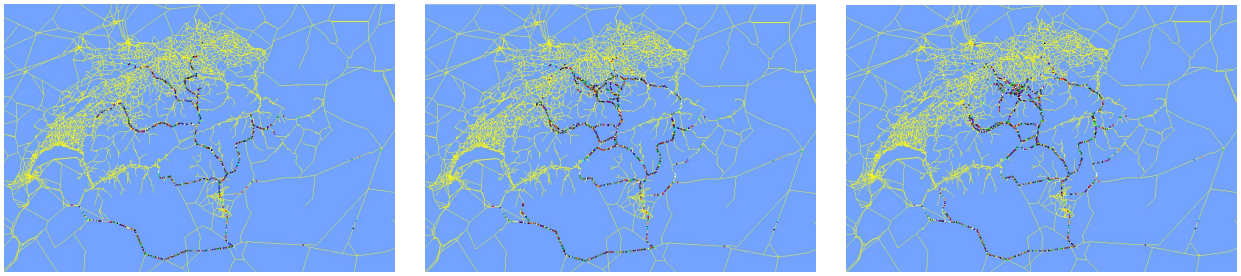


(a) 0% Replanning - 07:00 (b) 5% Replanning - 07:00 (c) 10% Replanning - 07:00

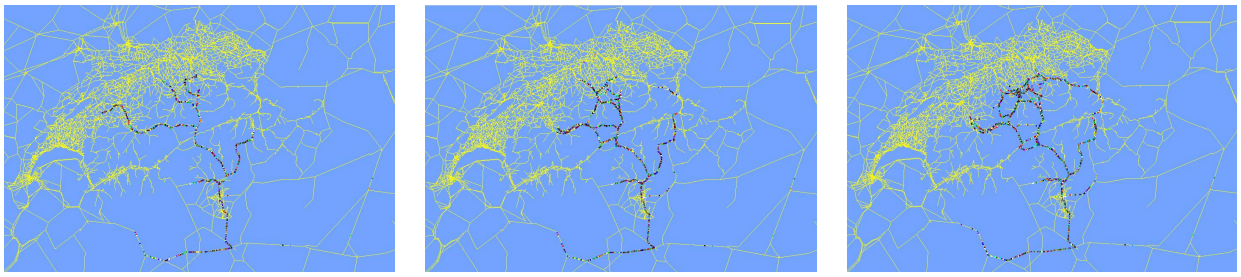


(d) 0% Replanning - 08:00 (e) 5% Replanning - 08:00 (f) 10% Replanning - 08:00

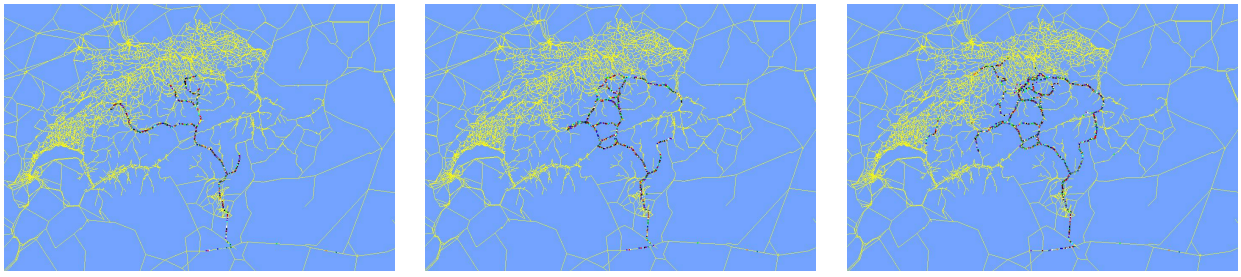
Figure 6.22: Agents' Travel Paths - 07:00 to 08:00



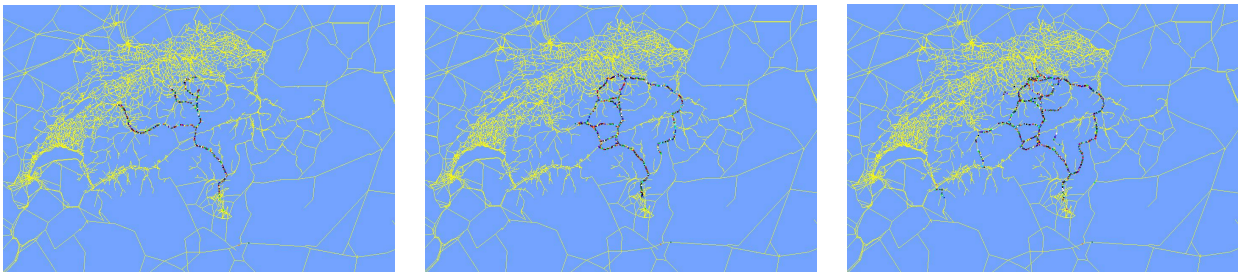
(a) 0% Replanning - 09:00 (b) 5% Replanning - 09:00 (c) 10% Replanning - 09:00



(d) 0% Replanning - 10:00 (e) 5% Replanning - 10:00 (f) 10% Replanning - 10:00

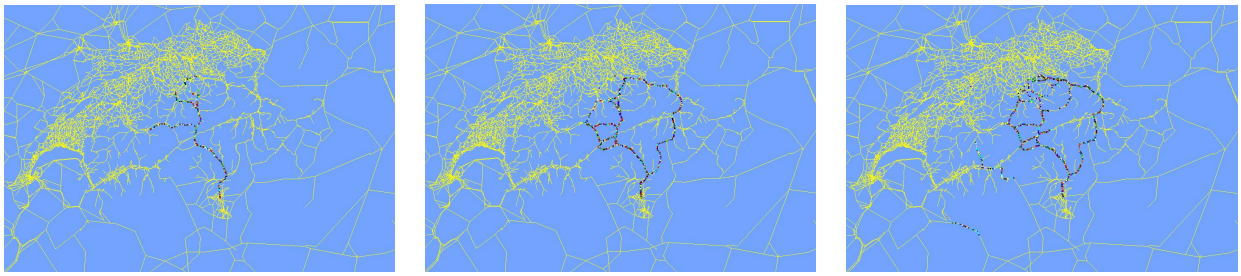


(g) 0% Replanning - 11:00 (h) 5% Replanning - 11:00 (i) 10% Replanning - 11:00

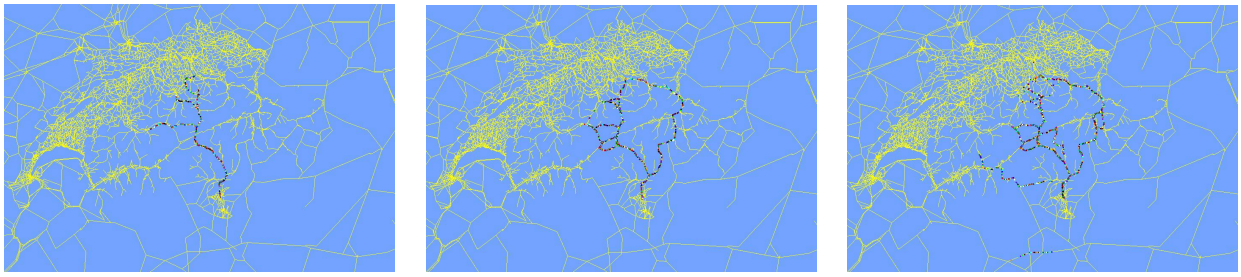


(j) 0% Replanning - 12:00 (k) 5% Replanning - 12:00 (l) 10% Replanning - 12:00

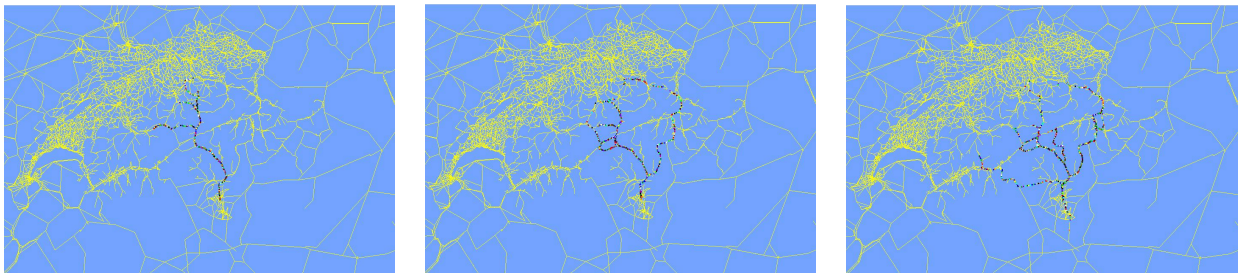
Figure 6.23: Agents' Travel Paths - 09:00 to 12:00



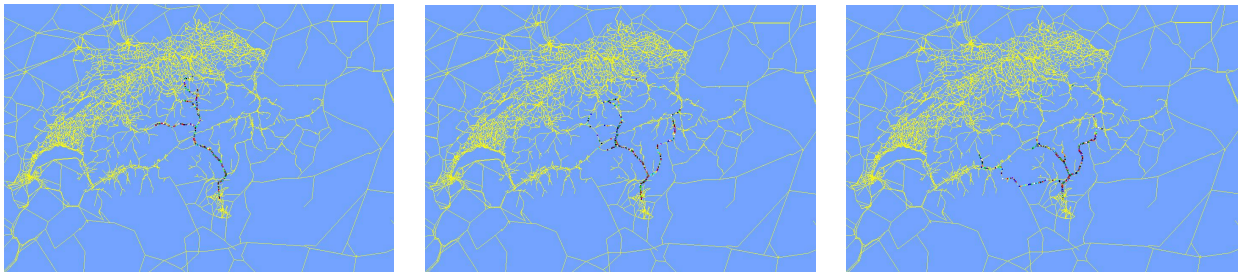
(a) 0% Replanning - 13:00 (b) 5% Replanning - 13:00 (c) 10% Replanning - 13:00



(d) 0% Replanning - 14:00 (e) 5% Replanning - 14:00 (f) 10% Replanning - 14:00

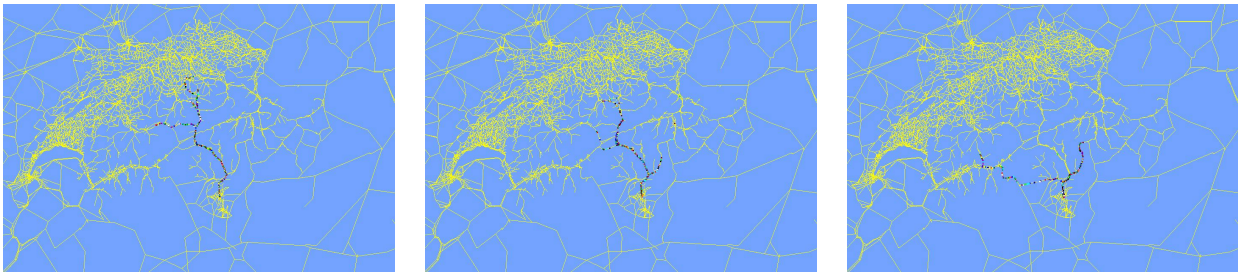


(g) 0% Replanning - 15:00 (h) 5% Replanning - 15:00 (i) 10% Replanning - 15:00

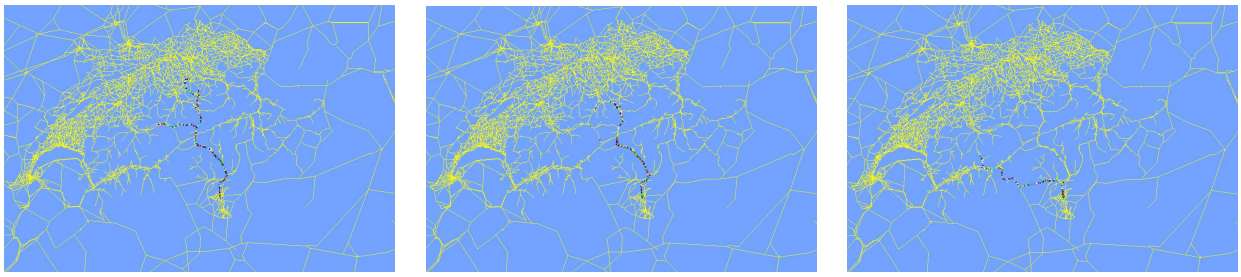


(j) 0% Replanning - 16:00 (k) 5% Replanning - 16:00 (l) 10% Replanning - 16:00

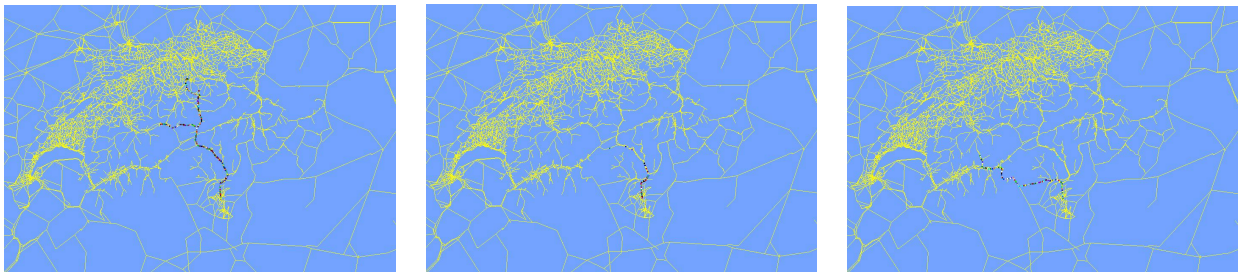
Figure 6.24: Agents' Travel Paths - 13:00 to 16:00



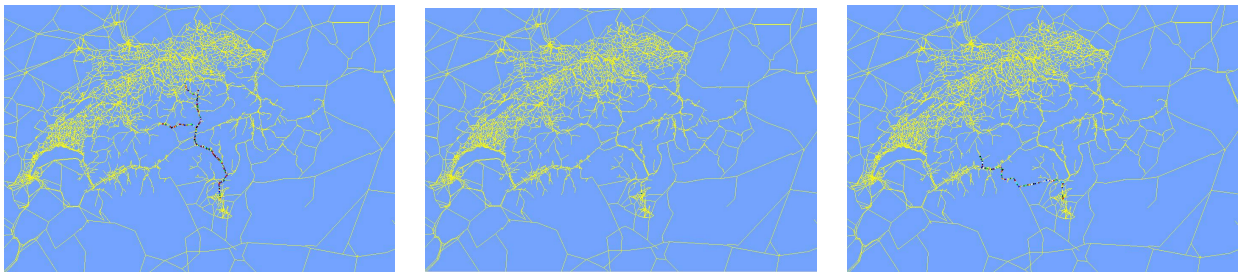
(a) 0% Replanning - 17:00 (b) 5% Replanning - 17:00 (c) 10% Replanning - 17:00



(d) 0% Replanning - 18:00 (e) 5% Replanning - 18:00 (f) 10% Replanning - 18:00



(g) 0% Replanning - 19:00 (h) 5% Replanning - 19:00 (i) 10% Replanning - 19:00



(j) 0% Replanning - 20:00 (k) 5% Replanning - 20:00 (l) 10% Replanning - 20:00

Figure 6.25: Agents' Travel Paths - 17:00 to 20:00

7 Conclusion

The most important results of this diploma thesis are that events communication is possible at a comparatively moderate overhead, and that within-day replanning is feasible. Significant speed improvements are achieved. The results from the execution speed tests also show how important good load balancing is if a simulation is to run as fast as possible.

It has been shown that within-day route replanning decreases agents' average travel times, so this form of replanning in addition to between-day replanning should reduce the number of iterations of a simulation needed to get sensible results (agents already behave fairly rationally in the first iteration).

8 Outlook

As the new simulation offers the possibility to easily integrate new replanning modules, testing different within-day replanning methods to get results even closer to reality follows logically.

One improvement which should dramatically decrease the run time of multi-iteration simulations is to add intelligent load balancing. This might be done by collecting link load statistics for one iteration and using these as edge weights in METIS.

To get results for a specific question in traffic simulation, a scenario is usually rerun several times (usually in excess of 100 iterations). Between the iterations, some agents get new plans based on travel times calculated from events written to a file in the previous iteration. Of course writing the events to a file during the execution of the simulation takes a lot of time, and reading in and parsing the file by another program running in between two simulations (generating new plans) also needs time.

In the new simulation, events are communicated to all processes at run time. It should be possible to have specific processes not involved in the microsimulation also receive all events and then do all the day-to-day replanning when the microsimulation terminates. Even better, it should be possible to have these "controller" processes do some work even while the microsimulation is still running, so the time needed between iterations could be cut short.

The implementation of the controller processes outlined should shorten the total simulation time for any run with more than one iteration. If simulations are run until all agents have reached their final destinations, the gain should be dramatic, as day-to-day replanning for most agents could be done in parallel while the last few agents are still in the microsimulation!

The controller described above has been partly implemented[1] and the concept seems to work. No real test could be conducted yet due to the lack of time, but a first impression suggests that some speed may be gained compared to the traditional aproach of having an independent module replan agents' days between the iterations.

[1] checked into CVS at src/student-projects/chzwicke/src-controller/

9 Summary

During the course of this diploma thesis, the original simulation was taken apart and recreated with stronger object-orientation in mind. Communication of events to all processes was added and parallel within-day replanning was implemented. At various points, bottlenecks limiting execution speed were found and many of them were eliminated, leading to a faster simulation.

Much care was taken to add useful comments to the code so that people doing additional work based on the new simulation should easily be able to understand what is going on.

The new simulation has opened the possibility to easily integrate within-day replanning modules of different kinds. Events which occur during the simulation may be received and processed by any module which needs these for some sort of computation. Two modules were implemented, namely modules for route replanning and activity replanning (activity drop).

The new features were tested thoroughly for correctness. The execution speed for all types of simulations was tested and compared. Also, some insights into the effects of the new features on the simulation were achieved.

A Terminology

- **Wall-clock time** Time passing in the real world, measured e.g. by the clock on your wall
- **Speed-up** is the ratio between the time the execution of a program takes on one CPU and the time it takes to execute on N CPUs: $S_N = \frac{T_1}{T_N}$, where S_N is the speed-up, T_1 and T_N are the execution times on 1 and N CPUs, respectively.
- The **real time ratio (RTR)** shows how much faster than reality the simulation runs. For example, if it takes three minutes to simulate three hours, the RTR is 60.
- The **efficiency** of a parallel program is defined as the speed-up S_N of the program for N CPUs divided by N, the number of CPUs. $E_N = \frac{S_N}{N}$.
- During a simulation run, **events** are created when something interesting happens, for example when an agent starts its journey or leaves a link. These **events** are used by other parts of the simulation, e.g. to calculate link travel times used by the router.

B User Manual

There are two versions of the simulation package, one which is optimized for speed and has most configuration options implemented as sharp defines / configuration switches, and another one which does not need to be recompiled each time a configuration option is changed, but runs more slowly (all speed tests were run with the faster version, of course). In the following sections, the two versions will be named "SPEED"[1] and "EASE"[2] when their options differ.

B.1 Compile Time Switches

The following compile time switches are enabled by adding DEFS += -D[name of the switch] to the Makefile, if not mentioned otherwise.

- **TIME_MEASUREMENT:** If this switch is enabled, wall clock time is measured for different areas of the program and written to stdout at the end of the actual simulation.
- **PARALLEL:** Enables parts of the simulation which are needed for parallel execution. This switch should be enabled indirectly by adding either MPIMYRI=1 (for myrinet) or MPIETH=1 (for ethernet) to the "make"-call (e.g. make MPIMYRI=1).

The following options concern the SPEED version of the simulation package only:

- **BACK_COMPATIBILITY:** This switch puts the simulation into back compatibility mode. This means that events output should be the same as for the original simulation.
- **NO_CHECKS** If this is enabled, all run time checks are disabled, for example, the simulation does not check if enough memory is allocated for events communication. If certain configuration options are set badly, this may lead to segmentation faults.
- **NO_STDOUT:** Disables all output to the console.
- **NO_ACTIVITY_CHAINS:** Disables activity chains, which means that only the first and second activity are imported for each agent.
- **MEMORY_CONSTRAINT:** Saves some memory but sacrifices speed in some cases (for example route replanning will be slower).
- **NO_CURRENT_EVENTS_OUTPUT:** Disables writing events output.
- **NO_CURRENT_EVENTS:** Disables communication of events between CPUs. If this switch is enabled, *NO_ROUTE_REPLANNING* and *NO_DAY_REPLANNING* should be enabled as well.
- **NO_ROUTE_REPLANNING:** Disables route replanning for all agents.

[1] branch "chzwicke-sharp-defined" in CVS
[2] Head in CVS

- **NO_DAY_REPLANNING:** Disables day replanning for all agents.
- **NO_VISUAL_OUTPUT:** If this is enabled, no visual output data will be written.
- **QUANTIFY:** Enables data collection by quantify. Should be enabled indirectly by calling make with QUANTIFY=1. Do *not* combine this switch with PURIFY=1
- **PURIFY:** May only be enabled by adding PURIFY=1 to the call to "make". Do *not* combine with QUANTIFY=1
- **IMPLEMENTATION:** Reserved for future use. If implementations for simulation models other than queue simulation are added, this switch will be used to select the model to use.

B.2 Configuration Options

- **root_dir:** The root directory based on which absolute filenames will be calculated from relative ones. This is used to complete relative file names (e.g. tmp/plans.xml). Filenames starting with "." will not be completed, they stand relative to the current working directory.
- **networkFilename:** The path to the network description file.
- **initialPlansFilename:** Path to the file listing all agents with their initial plans (activities and routes).
- **importInitialRouteProbability:** For the given percentage of agents, the initial route will be imported from *initialPlansFilename*. For all other agents, either free speed travel times or historic travel times (if available) are used to generate a route for each agent.
- **snapshotInterval:** The interval after which visual data is written if *visualizerActivated* is set to 1.
- **historicEventsFile:** Path to the file containing events from a former iteration.
- **workingEventsFilename:** File to which events occurring during the current simulation are written.
- **eventsInterval:** Interval after which events are exchanged between CPUs. Note: current travel times are updated after this interval even in the case where the simulation runs on one CPU only!
- **maxEventsPerStepEstimate:** An estimate of how many events may occur per time step. If this estimate is too low, the EASY version exits with an error message. The SPEED version will generate a segmentation fault if more than *eventsInterval* * *maxEventsPerStepEstimate* events occur during any *currentEventsInterval* steps.

- **routeReplanningProbability:** The probability that an agent which thinks it's late will be allowed to replan its route. This value should be given as a decimal fraction (e.g. 0.1 for 10% replanning probability).
- **routeReplanningInterval:** The interval after which agents are allowed to replan their routes.
- **dayReplanningProbability:** The probability that an agent which thinks it's too late for it's next activity will be allowed to replan its day. This value should be given as a decimal fraction (e.g. 0.1 for 10% replanning probability).
- **dayReplanningInterval:** The interval after which agents are allowed to replan their days.
- **simEndTime:** The latest end time for the simulation. If the simulation has not ended at this point of time because no more agents were around, it will be terminated.
- **checkIfDoneInterval:** The interval after which the simulation checks if all CPUs are done. Setting this to a too low value will slow down the simulation, setting it too high will leave an empty simulation run and thus increase the total run time. This value is only of importance in a simulation executed in parallel, in a one CPU run termination will be checked in each time step (this does not slow down the computation noticeably).
- **seed:** Used to seed the random number generator.
- **maxNumActivities:** The maximum number of activities an agent may have planned. The estimate of an agent's maximum size will be based partly on this value.
- **maxNumRouteNodes:** The maximum number of intersections an agent will pass on its way. The estimate of an agent's maximum size will be based partly on this value.
- **workingSnapshotFilename:** The file to which visual output is written if visual output is enabled (either by setting **visualizerActivated** to one or by *not* defining *NO_VISUAL_OUTPUT*, depending on which version of the simulation is used.

The following option concerns the EASY version of the simulation only:

- **visualizerActivated:** If this is set to 1, visualizer data is written to *workingSnapshotFilename*

B.3 Networks

The network file format is as follows:

```
<network>
  <nodes>
    <node id="[node id]" x="[node x coordinate]" y="[node y coordinate]">
    ...
```

```
    ...
  </nodes>
  <links capdivider="[capacity divider (links' capacities divided by 3600 * this)]">
    <link id="[link id]" from="[origin node id]" to="[destination node id]"
      length="[link length]" capacity="[link capacity]>
    ...
    ...
  </links>
</network>
```

B.4 Agents & Plans

The file specifying the agents in the simulation along with their plans has the following format:

```
<plans>
  <person id="[agent id]">
    <plan>
      <act link="[start link id]" start_time="[activity start time]"
        tolerance="[maximum tine an agent may arrive late (in seconds)]"
        end_time="[latest activity end time]" dur="[duration]">
      <leg dep_time="[departure time (from last activity]">
        <route>[node id] [node id] ... </route>
      </leg>
      ...
      ...
      <act link="[start link id]" start_time="[activity start time]"
        tolerance="[maximum tine an agent may arrive late (in seconds)]"
        end_time="[latest activity end time]" dur="[duration]">
    </plan>
  </person>
  ...
  ...
</plans>
```

- Activities with no start time will be assumed to start as soon as possible.
- If no tolerance is given (the plans format of the original simulation does not include this value), 5 minutes late time is assumed to be OK.
- If neither duration nor end time is given, an activity is assumed to be carried out instantaneously, e.g. a drop-off by a delivery boy.
- If no departure time is given, an agent continues to its next activity as soon as the last activity has been carried out.
- If no route is given, one is calculated at run time, based on either free speed travel times or historic event data.

C Class Overview

- **Activity:** Stores information about a single activity an agent wants carry out
- **Agent:** Represents a person in the simulation. Contains information about the person's plans for the day (activities to go to) and a route from the current location to the next activity to go to. An agent also checks if it's late each time it crosses a node (only if day replanning or activity replanning are not deactivated).
- **Config:** Reads in all configuration options at startup and serves requests asking for them during the simulation.
- **EventListener:** Interface to be extended by all classes which wish to receive events during the simulation.
- **Factory:** Creates the appropriate subclasses for each specific implementation of the simulation (for queue simulation, it creates QAgents, QLinks and QNodes).
- **Link:** Basic class implementing a street, extended by different classes depending on the simulation type (for queue simulation, QLink extends Link).
- **Network:** Contains all links and nodes, forwards requests to these. Imports the network and all agents at startup.
- **Node:** Basic implementation of an intersection. Extended by simulation type specific subclasses (e.g. QNode for queue simulation).
- **Parallelism:** Contains everything necessary for parallel execution of the simulation, such as network decomposition, agent movement, empty space exchange and events communication.
- **QAgent:** Queue simulation specific subclass of Agent.
- **QLink:** Subclass of Link specific for queue simulations.
- **QNode:** Extends Node for queue simulations.
- **RingVector:** A vector with no start or end.
- **Route:** Stores a path from an agent's current point to its next activity.
- **Router:** Generates routes for origin-destination pairs. Also converts routes given as a sequence of node ids to a route usable by the simulation.
- **Simulator:** The main program
- **Tracer:** Writes to stdout. Filters messages depending on the currently set debug level.

D General Hints

D.1 Public-Key Authentication

To run parallel programs, it helps not having to enter a password for each ssh connection opened to a host participating in the computation. To make this possible, one can use public-key authentication with ssh. As instructions to be found on the web are incomplete at best, the following lines list the procedure to enable this form of authentication:

```
[user@client-machine:101]$ mkdir -p ~/.ssh
[user@client-machine:102]$ chmod 740 ~/.ssh
[user@client-machine:103]$ chmod g-w ~
[user@client-machine:104]$ ssh-keygen -t rsa -N "" -f ~/.ssh/id_rsa
[user@client-machine:105]$ ssh user@host-machine
[user@host-machine:106]$ mkdir -p ~/.ssh
[user@host-machine:107]$ chmod 740 ~/.ssh
[user@host-machine:108]$ chmod g-w ~
[user@host-machine:109]$ scp user@client-machine:~/.ssh/id_rsa.pub ~/.ssh/new_key.pub
[user@host-machine:110]$ touch ~/.ssh/authorized_keys
[user@host-machine:111]$ cat ~/.ssh/new_key.pub >> ~/.ssh/authorized_keys
[user@host-machine:112]$ rm ~/.ssh/new_key.pub
```

If $\sim$ is nfs-mounted, steps 105 - 109 and 112 may be left away.

D.2 Core Dumps

When debugging a program, core dumps often come in handy (they can be loaded into the debugger, which often saves a lot of time looking for the reason of a segmentation fault). For sequential programs, it is usually no problem to enable core dumps - `ulimit -c unlimited` for bash or `limit coredumpsize unlimited` for csh.

For parallel programs one may run into the problem that even if the interactive shell for some machine reports the maximum core dump size as unlimited, the same may not be true for a non-interactive shell on the same machine, so that when a parallel program is spawned over ssh by MPI, no core dump may be created after all.

The first possibility to resolve the problem is to have the administrator of the machine remove the limitation for non-interactive shells. The second possibility is to prepend the appropriate command for the remote shell (see above) to the command `mpirun` executes at the remote site.

D.3 Understanding the STL

D.3.1 General Philosophy

It is often the case that certain methods associated with an STL container take longer than expected to execute. For example, list::size() does not execute in constant time, but in time proportional to the number of members the list contains. The reason for that is that the STL designers thought that many programs would not need this method, so updating an extra variable each time a member is added to the list would simply waste time. The idea is that if a program needs information about the size of a list frequently, it should take care to keep the information current itself.

The above implies that anyone using STL containers in their programs should be careful to use the appropriate methods if they are concerned about the run-time of their programs. It often pays off well enough to read through the STL documentation carefully or even look at the source code of a specific container's implementation.

D.3.2 Iterators

The general use of STL iterators is the following:

```
for(X::iterator it = x.begin(); it != x.end(); ++it) {
  ...
}
```

The problem with the above code is that `x.end()` is called every time the exit condition is checked, and that call may be very inefficient for some STL classes. If you are not planning to change the iterator's base container during the iteration, the following should be much more efficient:

```
X::iterator xEnd = x.end();
for(X::iterator it = x.begin(); it != xEnd; ++it) {
  ...
}
```

D.4 Rational Quantify

In pinpointing bottlenecks in program execution, quantify was a big help. The version installed on zuse.inf.ethz.ch seems to have been out of date for quite a while, the now current version seems to be part of "IBM Rational PurifyPlus" ([10]).

The way one works with quantify is to compile the program code with the prefix "quantify", e.g. "quantify g++ program.cpp". When the resulting program is run, quantify automatically kicks in and collects data. The advantage of this tool is that it does not just record the program's point of execution every so many milliseconds, but actually counts the machine cycles each line of code takes to execute.

Finally, quantify presents its results in a nice GUI, showing details for each method as well as annotated source showing how much time the program spent in each line of code. Using these results, it is easy to identify bottlenecks - removing them is usually harder, but often quite possible.

Bibliography

[1] C. Gawron: *An Iterative Algorithm to Determine the Dynamic User Equilibrium in a Traffic Simulation Model*, IJMPC 1998

[2] N. Cetin, A. Burri, K. Nagel: *A Large-Scale Agent-Based Traffic Microsimulation Based On Queue Model*, STRC 2003

[3] B. Raney, N. Cetin, A. Völlmy, M. Vrtic, K. Axhausen, and K. Nagel: *An Agent-Based Microsimulation Model of Swiss Travel: First Results*, Networks and Spatial Economics, 3 (2003) 23–41

[4] http://www.litra.ch, 12.01.2004

[5] http://www.sgi.com/tech/stl/index.html, 19.01.2004

[6] http://expat.sourceforge.net, 19.01.2004

[7] http://www-unix.mcs.anl.gov/mpi/mpich, 19.01.2004

[8] http://www-users.cs.umn.edu/∼karypis/metis, 19.01.2004

[9] http://www.myri.com/myrinet, 22.01.2004

[10] http://www-306.ibm.com/software/awdtools/purifyplus, 06.02.2004