Large scale transportation simulations on Beowulf Clusters

Nurhan Cetin[*] and Kai Nagel[†]
Dept. of Computer Science, ETH Zürich, Switzerland
Postal: ETH Zentrum IFW B27.1, CH-8092 Zürich, Switzerland

May 31, 2001

Abstract

If the individual entities in a system are used as the main components of a traffic simulation, the simulation is called microscopic. When the traffic density is high and the area covered is wide, the individual elements of a microscopic simulation and also the simple rules such as car following, lane changing, gap acceptance, can result in complex behaviors. Such a large scale transportation simulation can consume more time and more computing resources. A parallel computing approach to such a big traffic system might be economical and efficient in terms of money and consumed resources.

This paper describes a parallel approach to a microscopic traffic simulation. The parallelization method is domain decomposition, which means that each CPU of the parallel computer is responsible for a different geographical area of the simulated region. We describe how information between domains is exchanged, and how the transportation network graph is partitioned. An adaptive scheme is used to optimize load balancing.

We demonstrate how computing speeds of a parallel micro-simulations can be systematically predicted once the scenario and the computer architecture are known. This makes it possible, for example, to decide if a certain study is feasible with a certain computing budget, and how to invest that budget. The main ingredients of the prediction are knowledge about the parallel implementation of the micro-simulation, knowledge about the characteristics of the partitioning of the transportation network graph, and knowledge about the interaction of these quantities with the computer system. In particular, we investigate the differences between switched and non-switched topologies, and the effects of 10 Mbit, 100 Mbit, and Gbit Ethernet.

Keywords: Parallel computing, traffic simulation, transportation planning

[*] cetin@inf.ethz.ch
[†] nagel@inf.ethz.ch

1 Introduction

If the individual entities in a system are used as the main components of a traffic simulation, the simulation is called microscopic. Although the micro-simulations have simple rules such as car following, lane changing, gap acceptance etc., these rules can produce complex behaviors if the traffic density is high on a wide area. Such a large scale transportation simulation can consume more time and more computing resources.

Large scale simulations can be run on a cluster of PCs to speed up the computation. Using a cluster of PCs and partitioning the whole task among the computers in this cluster is economical in that such a cluster is affordable by most university engineering departments and by middle size companies. By "a cluster of PCs", we mean that a group of 10-20 PCs connected by a standard LAN technology runs Beowulf Linux. The other solution might be buying a supercomputer such as IBM SP2 or Intel iPSC/860 in order to achieve the parallelism but this solution is not cost-effective.

2 Domain decomposition

Domain decomposition might be defined as partitioning the geographical region into subregions of approximately equal size (Fig. 1). It is one of the crucial issues of parallel computing. After partitioning the domain into subdomains, each CPU in the system is assigned to one of these subdomains and performs the calculation on that subdomain.

Since some of the vehicles in the traffic might leave a subdomain and enter into another subdomain on the way to their destinations, the traffic flow information near the boundary of the neighbor subdomains (or CPUs) needs to be exchanged. This is necessary in order to maintain the consistency between the CPUs.

In the following, we will describe the domain decomposition method for the cellular automata (CA) implementation which is used in TRANSIMS [12]. That particular implementation, however, is used for exposition only; the parallelization approach works on any driving logic which has a similar structure. The domain decomposition for parallelization is straightforward if the state at time $t+1$ depens only on information from time step t, and on neighboring cells. Therefore, an updating process in such a system is in principle composed of two elements, namely, a communication for the boundary information at time step t, and an update from time step t to $t+1$. In the actual implementation, we use two communcations and two sub-updates per time step, see later.

Traffic simulations fulfill two conditions which make this approach efficient:

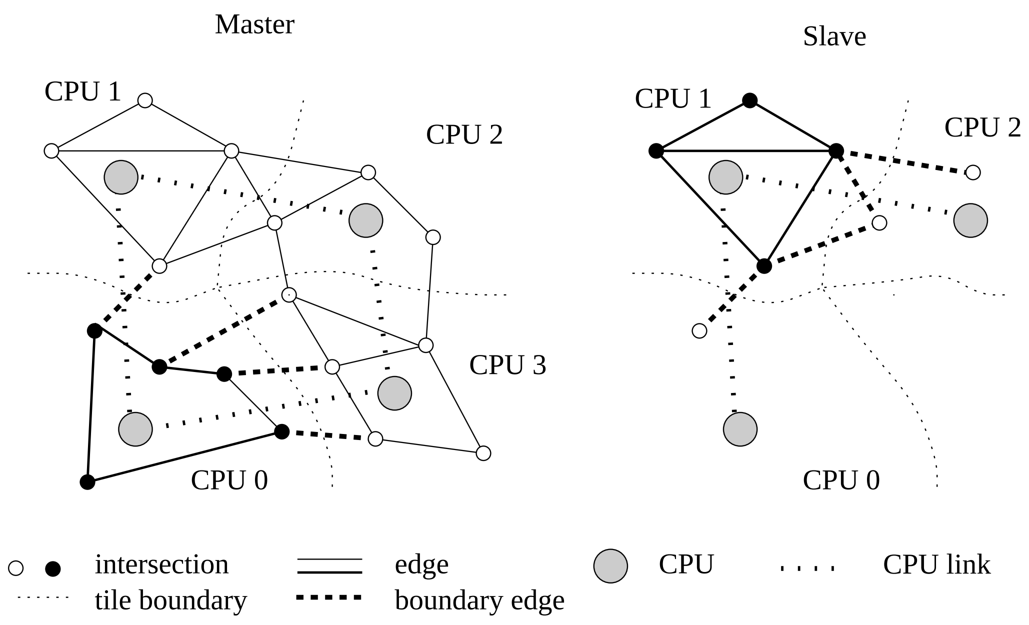


Figure 1: From [6].Domain decomposition of transportation network. *Left:* Global view. *Right:* View of a slave CPU. The slave CPU is only aware of part of the network which is attached to its local nodes. This includes links which are shared with neighbor domains.

- Domains of similar size: The street network can be partitioned into domains of similar size. A realistic measure for size is the accumulated length of all streets associated with a domain.

- Short-range interactions: For driving decisions, the distance of interactions between drivers is limited. In our CA implementation, on links all of the TRANSIMS-1999 [12] rule sets have an interaction range of 37.5 meters (= 5 cells, each of which has a length of 7.5 meters) which is small with respect to the average link length. Therefore, the network easily decomposes into independent components.

We decided to cut the street network in the middle of links rather than at intersections; THOREAU [7] does the same. This separates the traffic complexity at the intersections from the complexity caused by the parallelization and makes optimization of computational speed easier.

In the implementation, each divided link is fully represented in both CPUs. Each CPU is responsible for one half of the link. In order to maintain consistency between CPUs, the CPUs send information about the first five cells of “their” half of the link to the other CPU. Five cells is the interaction range of all CA driving rules on a link. By doing this, the other CPU knows enough about what is happening on the other half of the link in order to compute consistent traffic. Therefore the resulting simplified update sequence on the split links is as follows (Fig. 2):

- Change lanes.
- Exchange boundary information.
- Calculate speed and move vehicles forward.
- Exchange boundary information.

Note, however, that use of the CA can be viewed as a didactic example; any traffic simulation logic where the state at time $t + 1$ uses only information from time t and where interaction is local can be parallelized in this way.

3 Master-Slave Approach

Parallel programs distribute the work between many processors. The load should be distributed evenly so that some of processors are not idle (and/or some of processors are not overloaded). One of the popular techniques for the distribution is called Master-Slave Approach.

As the name implies, one of the processors is designated as master processor which has the knowledge of the overall work to be done. Therefore, the simulation is started up by the master, which spawns slaves, distributes the workload to them, and keeps control of the general scheduling.

Master-slave approaches often do not scale well with increasing numbers of CPUs since the workload of the master remains the same or even increases with increasing numbers of CPUs. For that reason, in TRANSIMS-1999 the master has nearly no tasks except initialization and synchronization. Even the output to file is done in a decentralized fashion. With the numbers of CPUs that we have tested in practice, we have never observed the master being the bottleneck of the parallelization.

4 Message Passing

In a parallel environment, some form of inter-processor communications is required in order to exchange data and information between processors and to provide synchronization of the processors. Generally, there are two main approaches

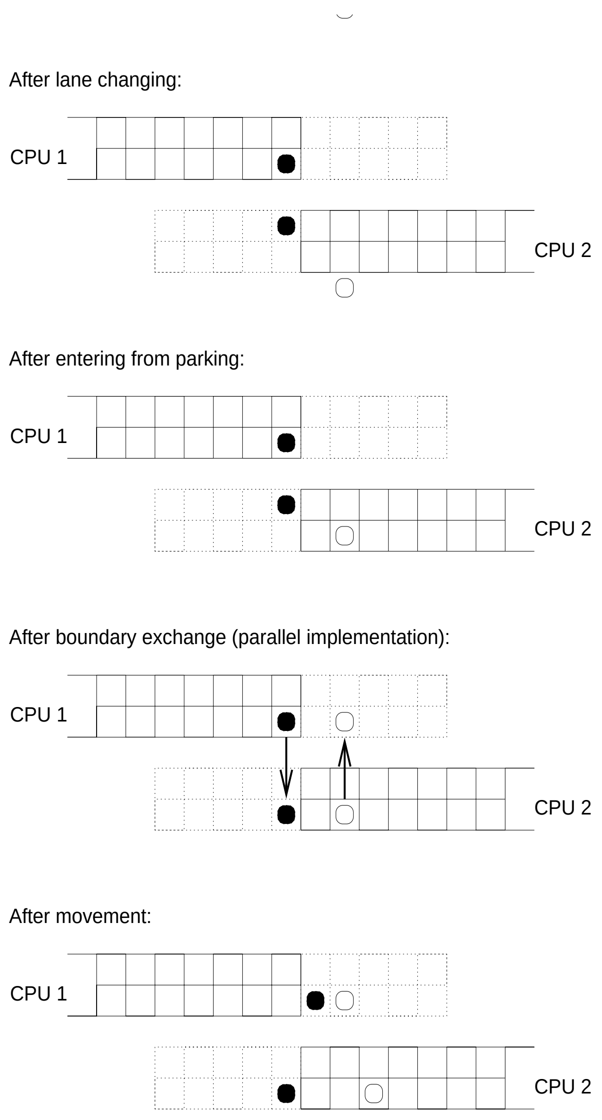

After lane changing:
CPU 1
CPU 2
After entering from parking:
CPU 1
CPU 2
After boundary exchange (parallel implementation):
CPU 1
CPU 2
After movement:
CPU 1
CPU 2

Figure 2: Example of parallel logic of a split link with two lanes.The figure shows the general logic of one time step. Remember that with a split link, one CPU is responsible for one half and another CPU is responsible for the other half. These two halves are shown separately but correctly lined up. The dotted part is the "boundary region", which is where the link stores information from the other CPU. The arrows denote when the information is transferred from one CPU to the other via boundary exchange [6].

to inter-processor communication. One of them is called *message passing* between processors and its alternative is to use *shared-address space* where variables are kept in a common pool therefore they are globally available to all processors. Each paradigm has its own advantages and disadvantages.

In the shared-address space approach, the variables are globally accessible by all processors. Despite multiple processors operating independently, they share the same memory resources. Only one processor can access the shared memory location at a time. Thus, accessing the memory should be provided in a mutually exclusive fashion since accesses to the same variable at the same time by multiple processors might lead to inconsistent data. Shared-address space approach makes it simpler for the user to achieve parallelism but since the memory bandwidth is limited, severe bottlenecks are unavoidable with the increasing number of processors. Also, the user is responsible for providing the synchronization constructs in order to provide concurrent accesses.

In the message passing approach, there are independent cooperating processes (or processors). Each processor has a private local memory in order to keep the variables and data. If an exchange of the information is needed between the processors, the processors communicate and synchronize by passing messages which are simple *send* and receive instructions. With this method, each processor can access its own memory very rapidly. But users have to send and receive data among processors.

The message passing paradigm is usually provided with the library extensions added to the sequential programming languages. PVM([9]), MPI([5]), P4([8]) are the most common message passing libraries and programs.

PVM refers to Parallel Virtual Machine, which is a software package that allows a programmer to create and access a parallel computing system. The components of such a system are the machines connected through the network(s). These machines might be in the same network as well as separated through the internet. Also,they may be homogenenous or heterogeneous in terms of the operating system running on those hosts. The idea is to bring together a variety of architectures under a centralized control. Thus a PVM user divides a problem into subtasks and assigns each subtask to one processor in the system.

PVM is based on the parallel message-passing model. Messages are exchanged between tasks via the connecting networks. If the communication is done between two different types of machines that do not have a common representation for the data, then data conversion is done automatically. Initialization and termination of a process are the user's responsibilities. The user should also use standard interface routines defined in PVM in order to exchange data and to synchronize with the other processes.

MPI stands for Message Passing Interface. It provides a standard for writing message-passing programs. It was designed for high performance on both massively parallel machines and on workstation clusters. It provides more than 100 functions as a library. It also defines an interface for Fortran and C. There are a couple of implementations of MPI on different architectures/systems. It also support heteregenous computing as PVM does. A comparison of MPI and PVM can be found in [3].

There are several other libraries in the literature and they have more or less the same procedures and usage. Some of them are commercial products but one can find free available libraries (such as PVM) too.

5 Graph Partitioning

Graph partitioning is a technique for executing a set of tasks in parallel so as to balance the load and minimize communications among processors. Once we are able to handle split links, we need to partition the whole transportation network graph in an efficient way. Efficient means several competing things: Minimize the number of split links; minimize the number of other domains each CPU shares links with; equilibrate the computational load as much as possible.

There are several algorithms and software for graph partitioning. One approach to domain decomposition is orthogonal recursive bisection. Although less efficient than METIS (explained below), orthogonal bisection is useful for explaining the general approach. In our case, since we cut in the middle of links, the first step is to accumulate computational loads at the nodes: each node gets a weight corresponding to the computational load of all of its attached half-links.

Nodes are located at their geographical coordinates. Then, a vertical straight line is searched so that, as much as possible, half of the computational load is on its right and the other half on its left. Then the larger of the two pieces is picked and cut again, this time by a horizontal line. This is recursively done until as many domains are obtained as there are CPUs available.The orthogonal bisection for Portland 200 000 links network is shown in Fig. 4. It is immediately clear that under normal circumstances this will be most efficient for a number of CPUs that is a power of two. With orthogonal bisection, we obtain compact and localized domains, and the number of neighbor domains is limited.

Another option is to use the METIS library for graph partitioning [4]. METIS uses multilevel partitioning. What that means is that first the graph is coarsened, then the coarsened graph is partitioned, and then it is uncoarsened again, while using an exchange heuristic at every uncoarsening step. The coarsening can for example be done via random matching, which means that first edges are randomly selected so that no two selected links share the same vertex, and then the two nodes at the end of each edge are collapsed into one. Once the graph is sufficiently collapsed, it is easy to find a good or optimal partitioning for the collapsed graph. During uncoarsening, it is systematically tried if exchanges of nodes at the boundaries lead to improvements. “Standard” METIS uses multilevel recursive bisection: The initial graph is partitioned into two pieces, each of the two pieces is partitioned into two pieces each again, etc., until there are enough pieces. Each such split uses its own coarsening/uncoarsening sequence. k-METIS means that all k partitions are found during a single coarsening/uncoarsening sequence, which is considerably faster. It also produces more consistent and better results for large k.

The number of split links from METIS can be approximated as $N_{spl} \approx 140\,p^{0.59} - 140$ for the 20 024-links network mentioned above; for a higher resolution network with 200 000 links we obtain $N_{spl} \approx 250\,p^{0.59}$ [6]. p is the number of CPUs. The orthogonal bisection method, on the other hand, scales N_{spl} as $\sim p^{0.5}$. Therefore, METIS considerably reduces the number of split links.

Such empirical results on graph partitioning can be used to compute the theoretical efficiency. Efficiency is optimal if each CPU gets exactly the same computational load. However, because of the granularity of the entities (nodes plus attached half-links) that we distribute, load imbalances are unavoidable, and they become larger with more CPUs. We define the resulting theoretical efficiency due to the graph partitioning as

$$e_{dmn} = \frac{\text{load on optimal partition}}{\text{load on largest partition}} \,, \qquad (1)$$

where the load on the optimal partition is just the total load divided by the number of CPUs. We then calculated this number for actual partitions of both of our 200 000 links and of our 200 000links Portland networks as shown in Fig. 3 (from [11]). The result shows that, according to this measure alone, our 200 000 links network would still run efficiently on 128 CPUs, and our 200 000links network would run efficiently on up to 1024 CPUs.

6 Adaptive Load Balancing

Load balancing is an important issue for a parallel system. It should be solved in order to enable the efficient use of parallel computer systems such that the loads on different CPU should be as similar as possible and all CPUs should be kept busy as much as possible.

The efficiency measure from the last section gives information about probable load imbalance due to the granularity of the smallest units, which are the nodes

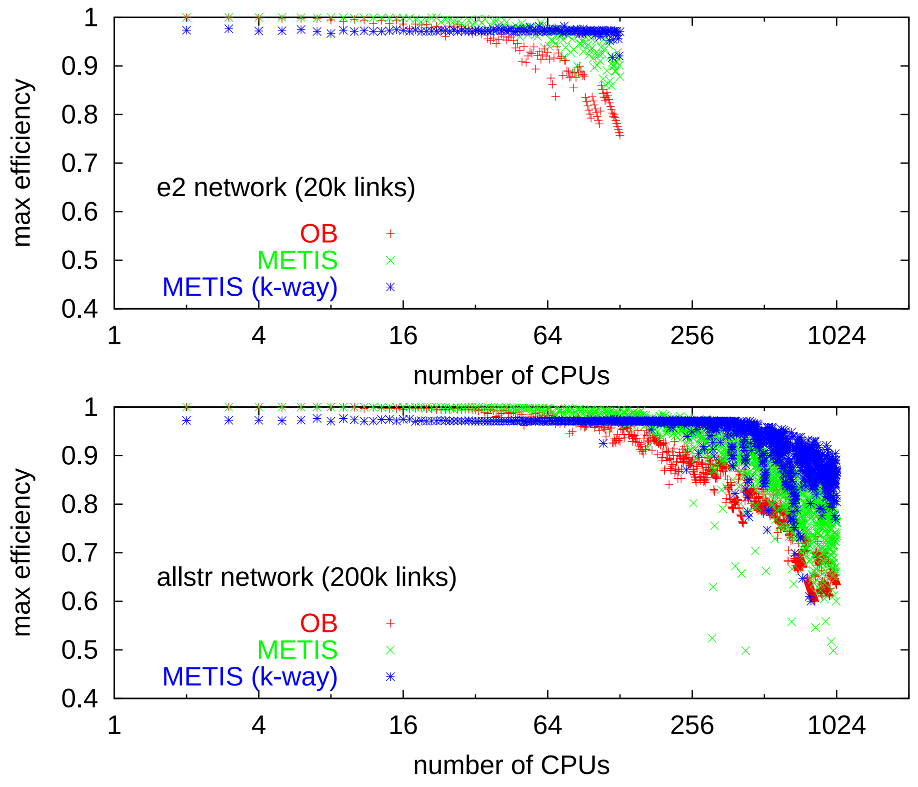


Figure 3: Theoretical efficiencies based on graph partitioning algorithms

with attached half-links. The approach in that section assumes that the computational load of those units depends on the lengths of the attached links only. Some applications, such as traffic simulations, do not have constant computational loads on those units, because the computational load depends on the number of vehicles on those links which in turn depends on traffic. Thus, we should optimize the average response time of both single tasks and the overall application in parallel in order to provide equal load on the CPUs and to minimize delays in data communication between these CPUs.

There are several common approaches to adaptation of the load balancing. One idea is alternating between a few different methods by defining a system as heavily, medium or lightly loaded and issuing different policies for each situation.

Another approach, that is used here, is to learn from the outputs of the previous runs. The loads on CPUs depend on the actual vehicle traffic in the respective domains. Since we are doing iterations, we are running similar traffic scenarios over and over again. We use this feature for an adaptive load balancing: During

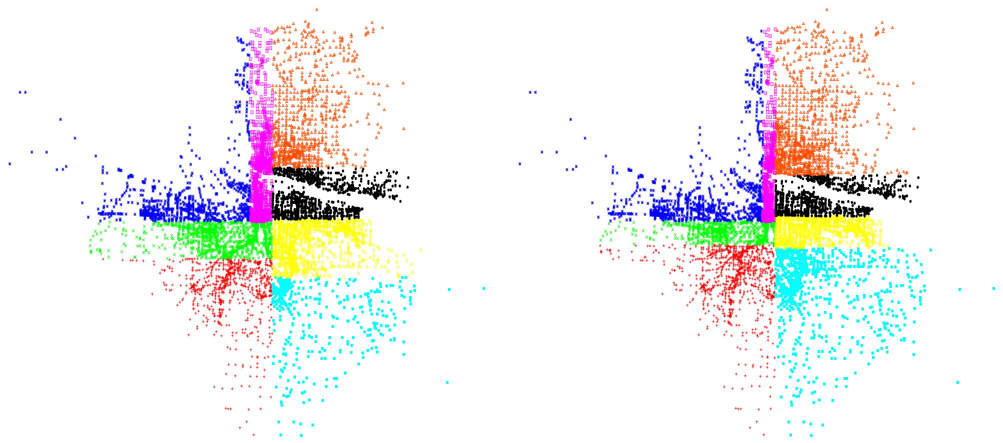


Figure 4: From [11]. Partitioning of the domain. *Left*: After orthogonal bisection. *Right*: After the adaptive load balancing.

run time we collect the execution time of each link and each intersection (node). The statistics are output to file. For the next run of the micro-simulation, the file is fed back to the partitioning algorithm. In that iteration, instead of using the link lengths as load estimate, the actual execution times are used as distribution criterion.

Fig.4 (right) shows the new domains after adaptive load balancing has been employed. One clearly sees that the sizes of the domains are different from the partitioning of the empty network (Fig. 4 left).

To verify the impact of this approach, we monitored the execution times per time-step throughout the simulation period. Figure 5 depicts the results of one of the iteration series. For iteration 1, the load balancer uses the link lengths as criterion. The execution times are low until congestion appears around 7:30 am. Then, the execution times increase fivefold from 0.04 sec to 0.2 sec. In iteration 2 the execution times are almost independent of the simulation time. Note that due to the equilibration, the execution times for early simulation hours increase from 0.04 sec to 0.06 sec, but this effect is more than compensated later on.

The figure also contains plots for later iterations (11, 15, 20, and 40). The improvement of execution times is mainly due to the route adaptation process: congestion is reduced and the average vehicle density is lower. On the machine sizes where we have tried it (up to 16 CPUs), adaptive load balancing led to performance improvements up to a factor of 1.8. It should become more important for larger numbers of CPUs since load imbalances have a stronger effect there.

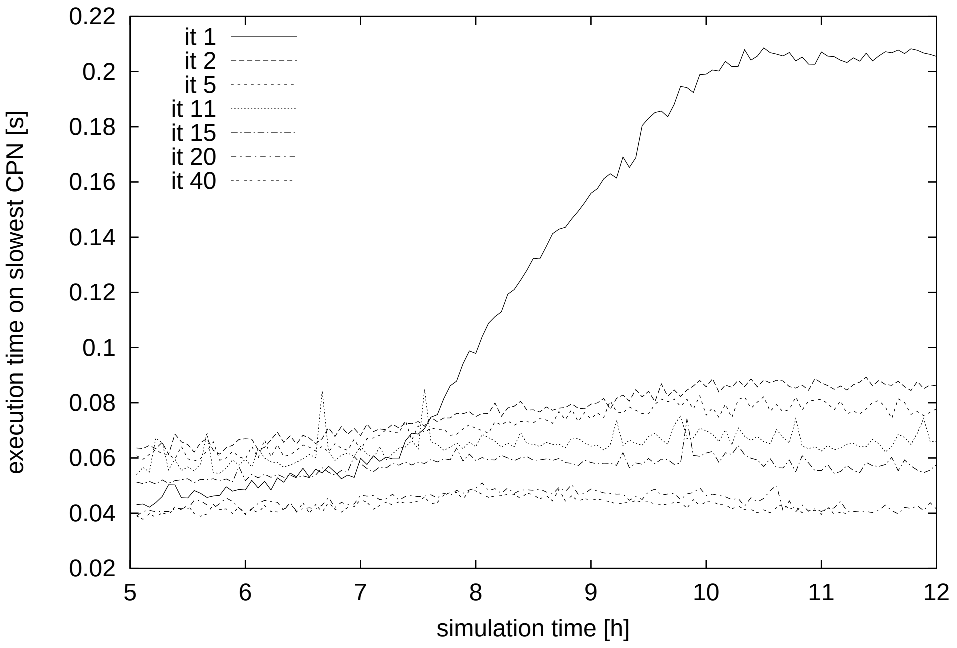


Figure 5: From [11]. Execution times with external load feedback. These results were obtained during the Dallas case study [1, 10].

7 Evaluation of Performance of the parallelized micro-simulation

The size of input usually determines the performance of a sequential algorithm (or program) evaluated in terms of execution time. However, this is not the case for the parallel programs. When evaluating parallel programs, besides the input size, the computer architecture and also the number of the processors should be taken into consideration.

There are various of metrics to evaluate the performance of a parallel program. Execution time, Speedup and Efficiency are the most common metrics to measure the performance of a parallel program. We will discuss these metrics in the following subsections.

7.1 Execution Time

The execution time of a parallel program is defined as the total time elapsed from the time the first processor starts execution to the time the last processor completes the execution. During execution, a processor is either computing or communicating. Therefore,

$$T(p) = T_{cmp}(p) + T_{cmm}(p) \ , \tag{2}$$

where T is the execution time, p is the number of processors, T_{cmp} is the computation time and T_{cmm} is the communication time.

The time required for the computation, namely, T_{cmp} can be calculated roughly in terms of the serial execution time (run time of the algorithm on a single CPU) and the number of processors. Thus,

$$T_{cmp}(p) = \frac{T_1}{p} \cdot \Big(1 + f_{ovr}(p) + f_{dmn}(p)\Big) \ . \tag{3}$$

where $T_1$is the serial execution time, p is the number of CPUs; f_{ovr} includes overhead effects (for example, split links need to be administered by *both* CPUs); f_{dmn} = 1/e_{dmn} - 1 includes the effect of unequal domain sizes as shown in Equation 1 in graph partitioning section.

Time for communication typically has two contributions: Latency and bandwidth. Latency is the time necessary to initiate the communication, and in consequence it is independent of the message size. Bandwidth describes the number of bytes that can be communicated per second. So the time for one message is

$$T_{msg} = T_{lt} + \frac{S_{msg}}{b} \ ,$$

where T_{lt} is the latency, S_{msg} is the message size, and b is the bandwidth.

However, for many of today's computer architectures, bandwidth is given by at least two contributions: node bandwidth, and network bandwidth. Node bandwidth is the bandwidth of the connection from the CPU to the network. If two computers communicate with each other, this is the maximum bandwidth they can reach. For that reason, this is sometimes also called the "point-to-point" bandwidth.

The network bandwidth is given by the technology and topology of the network. Typical technologies are 10Mbit Ethernet, 100Mbit Ethernet, FDDI, etc. Typical topologies are bus topologies, switched topologies, two-dimensional topologies (e.g. grid/torus), hypercube topologies, etc. A traditional Local Area Network (LAN) uses 10Mbit Ethernet, and it has a shared bus topology. In a shared bus topology, all communication goes over the same medium; that is, if several pairs of computers communicate with each other, they have to share the bandwidth.

For example, in our 100Mbit FDDI network (i.e. a network bandwidth of b_{net} = 100Mbit) at Los Alamos National Laboratory, we found node bandwidths of about b_{nd} = 40Mbit. That means that two pairs of computers could communicate at full node bandwidth, i.e. using 80 of the 100 Mbit/sec, while three or more pairs were limited by the network bandwidth. For example, five pairs of computers could maximally get 100/5 = 20 Mbit/sec each.

A switched topology is similar to a bus topology, except that the network bandwidth is given by the backplane of the switch. Often, the backplane bandwidth is high enough to have all nodes communicate with each other at full node bandwidth, and for practical purposes one can thus neglect the network bandwidth effect for switched networks.

If computers become massively parallel, switches with enough backplane bandwidth become too expensive. As a compromise, such supercomputers usually use a communications topology where communication to "nearby" nodes can be done at full node bandwidth, whereas global communication suffers some performance degradation. Since we partition our traffic simulations in a way that communication is local, we can assume that we do communication with full node bandwidth on a supercomputer. That is, on a parallel supercomputer, we can neglect the contribution coming from the b_{net}-term. This assumes, however, that the allocation of street network partitions to computational nodes is done in some intelligent way which maintains locality.

As a result of this discussion, we assume that the communication time per time step is

$$T_{cmm}(p) = N_{sub} \cdot \left(n_{nb}(p)\, T_{lt} + \frac{N_{spl}(p)}{p} \frac{S_{bnd}}{b_{nd}} + N_{spl}(p)\, \frac{S_{bnd}}{b_{net}} \right) , \qquad (4)$$

where N_{sub} is the number of sub-time-steps. Since we do two boundary exchanges per time step, N_{sub} = 2 for the 1999 TRANSIMS micro-simulation implementation.

n_{nb} is the number of neighbor domains each CPU talks to. All information which goes to the same CPU is collected and sent as a single message, thus incurring the latency only once per neighbor domain. For p = 1, n_{nb} is zero since there is no other domain to communicate with. For p = 2, it is one. For $p \to \infty$ and assuming that domains are always connected, Euler's theorem for planar graphs says that the average number of neighbors cannot become more than six. Based on a simple geometric argument, we use

$$n_{nb}(p) = 2\,(3\sqrt{p} - 1)\,(\sqrt{p} - 1)/p \ ,$$

which correctly has $n_{nb}(1) = 0$ and $n_{nb} \to 6$ for $p \to \infty$. Note that the METIS library for graph partitioning does not necessarily generate connected partitions, making this potentially more complicated.

T_{lt} is the latency (or start-up time) of each message. T_{lt} between 0.5 and 2 milliseconds are typical values for PVM on a LAN. Next are the terms that describe our two bandwidth effects. $N_{spl}(p)$ is the number of split links in the whole simulation. Accordingly, $N_{spl}(p)/p$ is the number of split links per computational node. S_{bnd} is the size of the message per split link. b_{nd} and b_{net} are the node and network bandwidths, as discussed above.

In consequence, the combined time for one time step is

$$T(p) = \frac{T_1}{p}\Big(1 + f_{ovr}(p) + f_{dmn}(p)\Big) +$$

$$N_{sub} \cdot \left(n_{nb}(p)\, T_{lt} + \frac{N_{spl}(p)}{p} \frac{S_{bnd}}{b_{nd}} + N_{spl}(p)\, \frac{S_{bnd}}{b_{net}} \right) \ .$$

According to what we have discussed above, for $p \to \infty$ the number of neighbors scales as $n_{nb} \sim const$ and the number of split links in the simulation scales as $N_{spl} \sim \sqrt{p}$. In consequence for f_{ovr} and f_{dmn} small enough, we have:

- for a shared or bus topology, b_{net} is relatively small and constant, and thus
$$T(p) \sim \frac{1}{p} + 1 + \frac{1}{\sqrt{p}} + \sqrt{p} \to \sqrt{p}\ ;$$
- for a switched or a parallel supercomputer topology, we assume $b_{net} = \infty$ and obtain
$$T(p) \sim \frac{1}{p} + 1 + \frac{1}{\sqrt{p}} \to 1\ .$$

Thus, in a shared topology, adding CPUs will eventually increase the simulation time, thus making the simulation *slower*. In a non-shared topology, adding CPUs will eventually not make the simulation any faster, but at least it will not be detrimental to computational speed. The dominant term in a shared topology for $p \to \infty$ is the network bandwidth; the dominant term in a non-shared topology is the latency.

The curves in Fig. 6 are results from this prediction for a switched 100 Mbit Ethernet LAN; dots and triangles show actual performance results [6]. The top graph shows the time for one time step, i.e.*T(p),* and the individual contributions to this value. One can clearly see that for more than 64 CPUs, the dominant time contribution comes from the latency.The bottom graph shows the real time ratio (RTR)

$$rtr(p) := \frac{\Delta t}{T(p)} = \frac{1 sec}{T(p)}\ ,$$

which says how much faster than reality the simulation is running. Δt is the duration a simulation time step, which is $1sec$ in TRANSIMS-1999. This figure shows that even something as relatively profane as a combination of regular Pentium CPUs using a switched 100Mbit Ethernet technology is quite capable in reaching good computational speeds. For example, with 16 CPUs the simulation runs 40 times faster than real time; the simulation of a 24 hour time period would thus take 0.6 hours. These numbers refer to the Portland 200 000 links network. Included in the plot (black dots) are measurements with a compute cluster that corresponds to this architecture. The triangles with lower performance for the same number of CPUs come from using dual instead of single CPUs on the computational nodes. Note that the curve levels out at about forty times faster than real time, no matter what the number of CPUs. As one can see in the top figure, the reason is the latency term, which eventually consumes nearly all the time for a time step. This is one of the important elements where parallel supercomputers are different: For example the Cray T3D has a more than a factor of ten lower latency under PVM [2].

Fig. 7 shows some predicted real time ratios for other computing architectures. For simplicity, we assume that all of them except for one special case explained below use the same 500MHz Pentium compute nodes. The difference is in the networks: We assume 10Mbit non-switched, 10Mbit switched, 1Gbit non-switched, and 1Gbit switched. The curves for 100Mbit are in between and were left out for clarity; values for switched 100Mbit Ethernet were already in Fig. 6. One clearly sees that for this problem and with today's computers, it is nearly impossible to reach *any* speed-up on a 10Mbit Ethernet, even when switched. Gbit Ethernet is somewhat more efficient than 100Mbit Ethernet for small numbers of CPUs, but for larger numbers of CPUs, switched Gbit Ethernet saturates at exactly the same computational speed as the switched 100Mbit Ethernet. This is due to the fact that we assume that latency remains the same – after all, there was no improvement in latency when moving from 10 to 100Mbit Ethernet. FDDI is supposedly even worse [2].

The thick line in Fig. 7 corresponds to the ASCI Blue Mountain parallel supercomputer at Los Alamos National Laboratory. On a per-CPU basis, this machine is slower than a 500 MHz Pentium. The higher bandwidth and in particular the lower latency make it possible to use higher numbers of CPUs efficiently, and in fact one should be able to reach a real time ratio of 128 according to this plot. By then, however, the granularity effect of the unequal domains (Fig. 3) would have set in, limiting the computational speed probably to about 100 times real time with 128 CPUs. We actually have some speed measurements on that machine for up to 96 CPUs, but with a considerably slower code from summer 1998. We omit those values from the plot in order to avoid confusion.

Fig. 7 (bottom) shows predictions for the higher fidelity Portland 200 000links

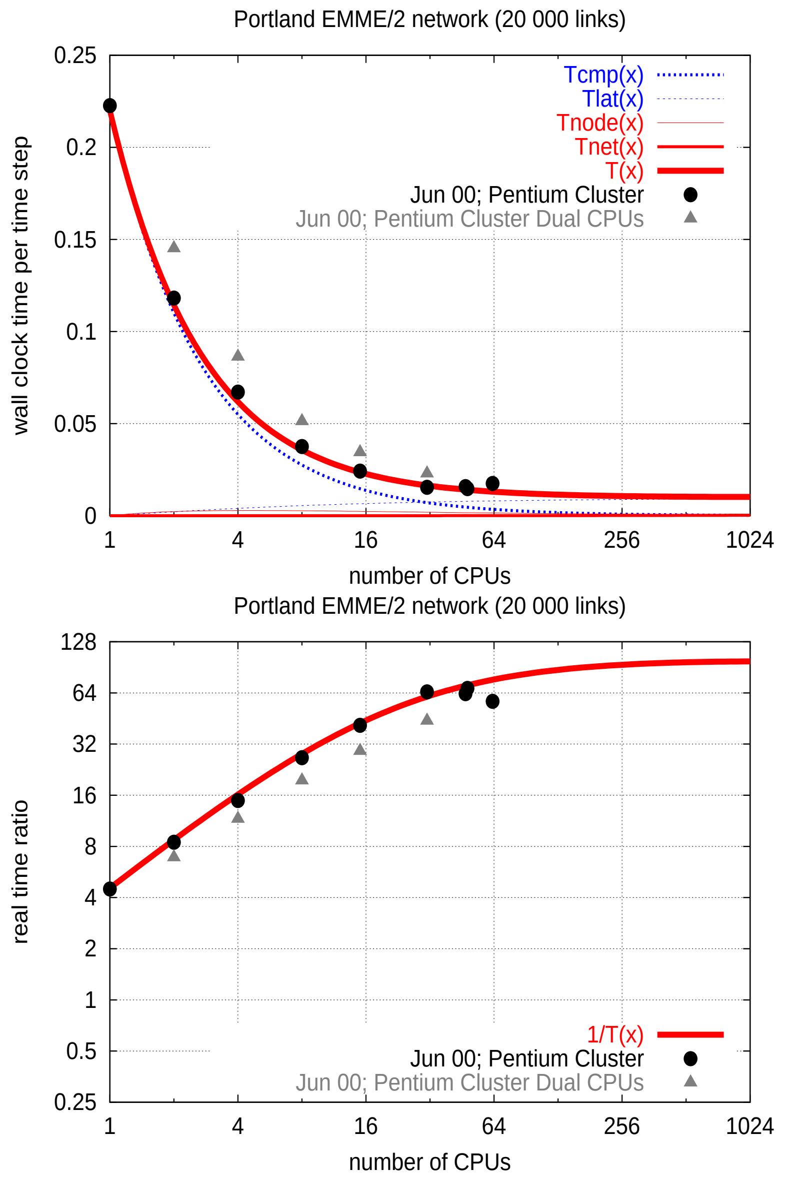
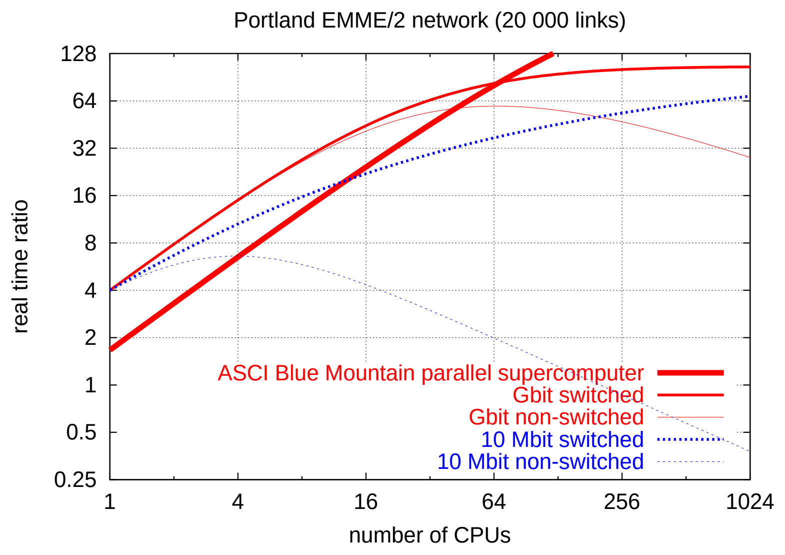


Figure 6: 100 Mbit switched Ethernet LAN. *Top:*From [6]. Individual time contributions. *Bottom:* Corresponding Real Time Ratios. The black dots refer to actually measured performance when using one CPU per cluster node; the crosses refer to actually measured performance when using dual CPUs per node (the *y*-axis still denotes the number of CPUs used). The thick curve is the prediction according to the model. The thin lines show the individual time contributions to the thick curve.



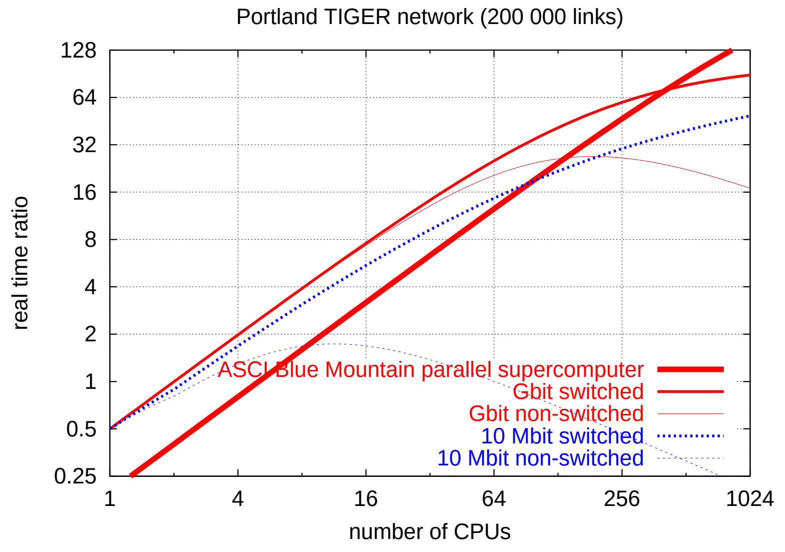


Figure 7: From [6]. Predictions of real time ratio for other computer configurations. *Top:* With Portland 20 024 links network. *Bottom:* With Portland 200 000 links network. Note that for the switched configurations and for the supercomputer, the saturating real time ratio is the same for both network sizes, but it is reached with different numbers of CPUs. This behavior is typical for parallel computers: They are particularly good at running larger and larger problems within the same computing time. All curves in both graphs are predictions from our model. We have some performance measurements for the ASCI machine, but since they were done with an older and slower version of the code, they are omitted in order to avoid confusion.

network with the same computer architectures. The assumption was that the time for one time step, i.e. T_1 of Eq. (3), increases by a factor of eight due to the increased load. This has not been verified yet. However, the general message does not depend on the particular details: When problems become larger, then larger numbers of CPUs become more efficient. Note that we again saturate, with the switched Ethernet architecture, at 80 times faster than real time, but this time we need about 64 CPUs with switched Gbit Ethernet in order to get 40 times faster than real time — for the smaller Portland 200 000 links network with switched Gbit Ethernet we would need 8 of the same CPUs to reach the same real time ratio. In short and somewhat simplified: As long as we have enough CPUs, we can micro-simulate road networks of *arbitrarily large size*, with hundreds of thousands of links and more, 40 times faster than real time, even without supercomputer hardware. — Based on our experience, we are confident that these predictions will be lower bounds on performance: In the past, we have always found ways to make the code more efficient.

7.2 Speed-Up and Efficiency

We have cast our results in terms of the real time ratio, since this is the most important quantity when one wants to get a practical study done. In this section, we will translate our results into numbers of speed-up, efficiency which allow easier comparison for computing people.

Speedup achieved by a parallel algorithm is defined as the ratio of the time required by the best sequential algorithm to solve a problem, $T(1)$, to the time required by parallel algorithm using *p* processors to solve the same problem,$T(p)$. For simplicity, $T(1)$ is calculated by running the parallel program on one processor.

We can define the Speedup as in the following formula

$$S(p) := \frac{T(1)}{T(p)} \ ,$$

where *p* is again the number of CPUs. Depending on the viewpoint, for $T(1)$ one uses either the running time of the parallel algorithm on a single CPU, or the fastest existing sequential algorithm. Since TRANSIMS has been designed for parallel computing and since there is no sequential simulation with exactly the same properties, $T(1)$ will be the running time of the parallel algorithm on a single CPU. For time-stepped simulations such as used here, the difference is expected to be small.

Speedup is limited by a couple of factors. First, the software overhead appears in the parallel implementation since code length of a parallel implementation is more than the one of sequential program. Second, speedup is generally limited by the speed of the slowest node or processor. Thus, we need to make sure that each node performs the same amount of work. i.e. the system is load balanced. Third, if the communication and computation cannot be overlapped, then the communication will reduce the speed of the overall application. To avoid this, the parallel program should keep the processors busy as much as possible.

A final limitation of the Speedup is known as Amdahl's Law - Serial Fraction. This states that the speedup of a parallel algorithm is effectively limited by the number of operations which must be performed sequentially. Thus, let's define S as the amount of the time spent by one processor on sequential parts of the program and P as the amount of the time spent by one processor on parts of the program that can be parallelized. Then, we can formulate the serial run-time as $T(1)$:= S + P and the parallel run-time as $T(p)$:= S + P/N. Therefore, the serial fraction F will be

$$F := \frac{S}{T(1)} \ ,$$

and the speedup $S(p)$ is expressed as

$$S(p) := \frac{S+P}{S+\frac{P}{N}} \ ,$$

or in terms of serial fraction, it would be

$$S(p) := \frac{1}{F+\frac{1-F}{N}} \ ,$$

.

As an illustration, let us say, we have a program containing 100 operations each of which take 1 time unit. If 80 operations can be done in parallel i.e. P = 80 and 20 operations must be done sequentially i.e. S = 20. then by using 80 processors, the Speedup would be 100 / 21 ¡ 5 i.e. a speedup of only 5 is possible no matter how many processors are available.

Now note again that the real time ratio is $rtr(p) = 1\ sec/T(p)$. Thus, in order to obtain the speed-up from the real time ratio, one has to multiply all real time ratios by $T(1)/(1sec)$. On a logarithmic scale, a multiplication corresponds to a linear shift. In consequence, speed-up curves can be obtained from our real time ratio curves by shifting the curves up or down so that they start at one.

This also makes it easy to judge if our speed-up is linear or not. For example in Fig. 7 bottom, the curve which starts at 0.5 for 1 CPU should have an RTR of 2 at 4 CPU, an RTR of 8 at 16 CPUs, etc. Downward deviations from this mean sub-linear speed-up. Such deviations are commonly described by another number, called efficiency, and defined as

$$E(p) := \frac{T(1)/p}{T(p)}$$

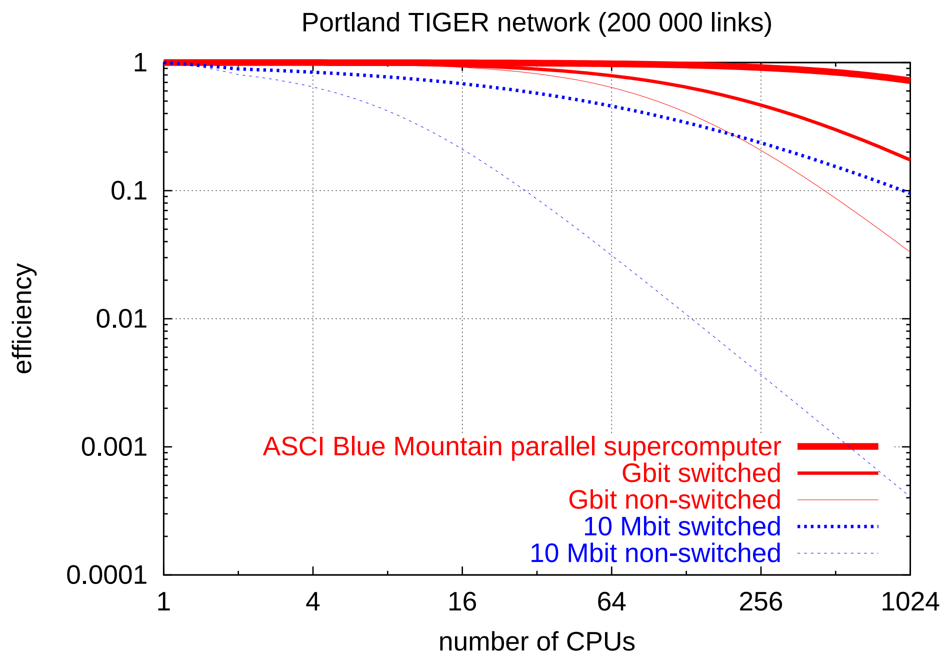


Figure 8: From [6]. Efficiency for the same configurations as in Fig. 7 bottom. Note that the curves contain exactly the same information.

.

It is obvious that an ideal system with p processors has a speedup equal to p. However, this is not the case in practice since in a parallel program, a processor cannot use 100% of its time for the computation. It should also consume some of its time for the communication. Therefore, we can interpret the efficiency formula above as a measure of the percentage of time for which a processor is utilized effectively. Ideally, efficiency equals to 1 but in practice it is between 0 and 1 depending on how a processor is employed.

Fig. 8 contains curves of the efficiency E as a function of the number of CPUs p for some of the cases discussed above. Note again that these plots contain no new information, they are just re-interpretations of the data used for Fig. 8 bottom. Also note that in our logarithmic plots, $E(p)$ will just be the difference to the diagonal $pT(1)$. Efficiency can point out where improvements would be useful.

8 Summary

This paper explains a parallelization method for the wide area micro traffic simulations. These kind of simulations should be parallelized in order to achieve an efficient use in terms of computing resources. Our approach here is to run such a simulation on a cluster of PCs which is much more affordable than to buy a parallel computer.

Parallel computing comes with some important issues such as domain decomposition, data sharing/exchanging and communication between processors. We represent our approaches on these issues which will affect the performance of a parallel system.

A well-behaved parallel system is load balanced. In order to achieve load balancing, one should be careful with the domain decomposition. If the parallel application does not have constant loads on the processors, it is better to use a dynamic/adaptive method to disaggregate the domain onto processors.

Data sharing among processors can be employed by using either shared-address space method or message passing approach. Message passing is more efficient in terms of bandwidth and memory usage. Each processor is independent but at the same time in a cooperation with the other processors when necessary. As the name implies, the communication is done through the messages exchanged among processors.

We finally demonstrate how computing time for a parallel traffic micro-simulation can be systematically predicted. An important result is that a typical city with 20 024 links network runs about 40 times faster than real time on 16 500 MHz Pentium computers connected via switched 100 Mbit Ethernet. These are regular desktop/LAN technologies. When using the next generation of communications technology, i.e. Gbit Ethernet, we predict the same computing speed for a much larger network of 200 000 links with 64 CPUs.

References

[1] R.J. Beckman et al. TRANSIMS–Release 1.0 – The Dallas-Fort Worth case study. Los Alamos Unclassified Report (LA-UR) 97-4502, see transims.tsasa.lanl.gov, 1997.

[2] J.J. Dongarra, I.S. Duff, D.C. Sorensen, and H.A. van der Vorst. *Numerical linear algebra for high-performance computers*. Software, Environments, and Tools. SIAM Society for Industrial and Applied Mathematics, Philadelphia, 1998.

[3] W. Gropp and E. Lusk. Why are pvm and mpi so different?

[4] METIS library. www-users.cs.umn.edu/˜karypis/metis/metis.html.

[5] MPI: Message Passing Interface. See www-unix.mcs.anl.gov/mpi/mpich.

[6] K. Nagel and M. Rickert. Parallel implementation of the TRANSIMS micro-simulation, submitted. See www.inf.ethz.ch/˜nagel/papers.

[7] W. Niedringhaus, J. Opper, L. Rhodes, and B. Hughes. IVHS traffic modeling using parallel computing: Performance results. In *Proceedings of the International Conference on Parallel Processing*, pages 688–693. IEEE, 1994.

[8] The p4 parallel programming system. See http://www-fp.mcs.anl.gov/ lusk/p4/.

[9] PVM: Parallel Virtual Machine. See www.epm.ornl.gov/pvm/pvm_home.html.

[10] M. Rickert. *Traffic simulation on distributed memory computers*. PhD thesis, University of Cologne, Germany, 1998. See www.zpr.uni-koeln.de/˜mr/dissertation.

[11] M. Rickert and K. Nagel. Dynamic traffic assignment on parallel computers. *Future generation computer systems*, in press. See www.inf.ethz.ch/˜nagel/papers.

[12] TRANSIMS, TRansportation ANalysis and SIMulation System, since 1992. See transims.tsasa.lanl.gov.