Model-based Parallelization of Discrete Traffic Simulation Models

Oliver Ullrich[1*], Daniel Lückerath[1], Ewald Speckenmeyer[2]

[1] National Science Foundation's Industry-University Cooperative Research Center, School of Computing and Information Sciences, Florida International University, ECS 243C, 11200 SW 8th St, Miami FL-33199; *oullrich@fiu.edu

[2] Institut für Informatik, Universität zu Köln, Albertus-Magnus-Platz, 50923 Köln, Germany

Simulation Notes Europe SNE 24(3-4), 2014, 115 - 122
DOI: 10.11128/sne.24.tn.10251
Received: September 28, 2014 (Selected ASIM SST 2014 Postconf. Publ.); Accepted: November 3, 2014;

Abstract. To re-establish regular operations in a tram traffic network after a large disturbance, e.g. resulting from vehicle breakdown or station closure, the viability of several rescheduling and rerouting strategies has to be evaluated prior to their implementation. Here, a multi-modal traffic simulation system can help to enhance the decision quality. Such a system obviously faces tight time constraints, so simulation data has to be acquired fast.

In this paper we propose a method for the parallel execution of discrete traffic simulation models, which would accelerate data generation in comparison to a sequential model. To assess this method's dynamic behavior in real-world applications, some experiments conducted on a software system modeling schedule based tram traffic are presented.

After giving an introduction to the scope and aim, we show some background on the parallelization of discrete simulation models. The main part of the paper begins with the proposal of a method to parallelize the execution of simulation models with problem specific properties. Some estimations of the method's efficiency are shared, followed by several experiments to highlight its dynamic behavior in real-world applications.

The paper ends with a short summary and some thoughts on further research.

Introduction

When severe disturbances occur in tram networks, e.g. originating from broken down trams, closed stations, or other blocked resources, traffic operators have to apply rescheduling and rerouting strategies (see [10] and [12]) to reestablish regular operations. To be effective, these strategies are inevitably multimodal: trams are rescheduled to compensate for cancellations, regional and local buses are rerouted to relieve the tram network, some transit operators even co-operate with taxi companies (see [26]). To evaluate the applicability of a given rescheduling or rerouting strategy prior to its implementation in the realworld system, a multi-modal simulation software is needed. Operators obviously face tight time constraints for their decisions, so simulation data has to be acquired fast.

In this paper we propose a method for the parallel execution of discrete traffic simulation models. We do not aim for a general approach, which would be equally well applicable for all discrete models, but for a method that utilizes some specific properties of a subclass of models, including traffic simulation models. The traffic planners' laptop or desktop computers constitute the target platform of the resulting simulation tools; the method thus should utilize their capacity for small scale parallel processing. To employ the available resources effectively, the method applies a dynamic and adaptive load balancing scheme. From the model's point of view, the mechanics of parallelization and load balancing are transparent, so that these internals can be changed without compromising the subsequent use of the model. A sequential model of tram traffic is already in place (see [11]); the accompanying representation of bus traffic is only partly implemented yet (see [24]).

This paper continues with a presentation of some background on general methods of parallel simulation and their customization in practical applications (section 1). We then present an approach on parallel execution of simulation models that share some characteristics like being spatially explicit, having mostly local dependencies, and having computational load generated by transient entities moving through a network (section 2), followed by some efficiency estimations (section 3). Based on an implementation of the approach, some experiments are conducted, focusing on the parallel computation of artificial loads, and the parallel simulation of tram traffic (section 4). The paper closes with a short summary of the lessons learned and some thoughts on future work (section 5).

1 Background

A discrete simulation model comprises of entities which communicate with each other via simulation events or messages, and which change their states at discrete points in model time. The main task when parallelizing a discrete simulation model is to avoid the occurrence of parallelization artifacts, i.e. to assure that parallel execution yields the same results as sequential execution. This is not a trivial task because a model usually contains both spatial dependencies (which are often local, i.e. between neighboring entities) and temporal dependencies (which are also often local, between close points in model time).

Several approaches to the parallelization of discrete simulation models are known: The simplest approach is the concurrent execution of several sequential experiments (see [15]). Outsourcing of maintenance functions (e.g. random number generation or database access) to secondary processors is also relatively simple but usually does not scale very well. A special case of this is the parallel administration of central data structures, usually applied to the future event list (for an overview of suitable data structures, see [18]). Time based parallelization techniques, i.e. computing different simulation time intervals in parallel, are scaling very well in principal but are only suited for specific models (a few applications are known, as described in [6] or [7]).

A well examined, and also well scaling technique is model based (also called spatial) parallelization, which can be found in both literature (see [5], pp. 39) and real-world applications (e.g. see [1], [13] or [19]).

1.1 Model based parallelization

Model based parallelization utilizes the model's inherent parallelism, i.e. that many state changes can be executed independently from each other. To achieve this, the model is decomposed into partial models, which are assigned to the involved processors. With this method, entities of different partial models communicate via messages sent over the network or a common cache memory.

Obviously, careful synchronization of the model's execution is necessary. The local causality constraint (see [5], pp. 52) demands that each entity executes its concerning simulation events in a non-decreasing order regarding their scheduled time of occurrence. Non-adherence to this constraint may result in causality errors which invalidate the simulation results. Two categories of synchronization methods are known: Conservative synchronization methods prevent the out-of-order execution of simulation events by technical measures, thus guaranteeing adherence to the local causality constraint. Important conservative synchronization mechanisms include synchronization via null messages (described in [2] and [3]), deadlock detection and recovery (described in [4]), and synchronized execution (see [17] and [21]). Optimistic synchronization methods execute the partial models as fast as possible, and thereby allow violations of the local causality constraint. The methods detect and subsequently repair these violations by rejecting and re-computing the invalid regions of the simulation run. The best known optimistic approach is the aptly named “time warp”, first described in [8].

1.2 Load balancing for parallel discrete simulation systems

Resulting from the typical dependencies in simulation models, the simulation's execution speed is generally dependent on the processor which advances slowest in simulation time. It is therefore necessary to incorporate a load balancing system into the simulation engine. This system does not aim at high utilization of the processors' capacity alone, but also has to consider an uniform advance in simulation time. Discrete simulation systems therefore often apply special load balancing schemes. Those methods can be characterized as dynamic, static, adaptive, non-adaptive, local, centered, or hierarchical (see [14], pp. 12).

A dynamic load balancing method continuously considers imbalances which develop from shifts in the model's computational load, and re-assigns partial models to appropriate processors while executing the simulation run. In contrast, static methods estimate the load and assign partial models to processors in a preprocessing step before the start of the simulation run, and thus don't consider dynamic changes in the model's activity. Adaptive methods consider fluctuations in the available processor power originating from the demand of dynamic processes belonging to third parties. In inhomogeneous computer networks adaptive methods also consider the dissimilar performance power of the respective processors. A non-adaptive approach ignores those fluctuations. In local methods, the processors only exchange data within limited neighborhoods and act on this local information alone, while centered methods utilize a marked controller process to which the other processors report. Hierarchical methods usually organize communication in a tree topology.

A load balancing method viable for traffic simulation models on a PC or laptop computer should be both dynamic and adaptive, and thus has to consider both the varying computational load of the model, and the changing availability of resources on a nonexclusively used machine. Such a dynamic and adaptive method requires a load measure which considers both the available processing power and the engaged processors' advance in simulation time. A centered method is usually easier to implement and quite adequate for a PC system with only eight to sixteen processor cores; if a method is targeted at a massively parallel system it should avoid a potential bottleneck by utilizing a hierarchical or local scheme.

2 An Approach to the Parallelization of Discrete Traffic Simulation Models

To be viable for the proposed method, a model has to comply to some prerequisites: It has to be spatially explicit and representable as a sparsely populated graph; dependencies have to be typically local, so that neighborhood relationships can therefore be exploited; the model's activity, and thus the computational load, has to be produced by transient entities which steadily move through the model, so the load typically shifts slowly and is caused by several simulation steps.

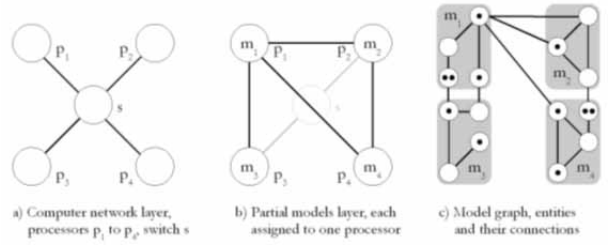

Figure 1: Three layers of abstraction.

Many traffic simulation models comply with these prerequisites (see e.g. [11], [16], and [24]). The proposed parallelization method is flexible regarding simulation paradigms: the implemented models can be event based, process based, agent based, or be based on the activity scanning approach. The method builds upon three layers of abstraction (see Figure 1): On the computer network layer processors and processor cores form a star-shaped graph, connected by a local area network or a shared cache memory; the partial models are assigned to these processors, and connected by the communication occurring during the simulation run; the model graph builds up on that and consists of model nodes, which represent entities, and their connecting edges. Transient entities (here represented by tokens) map the dynamically changing model activity; they move through the model graph along its edges.

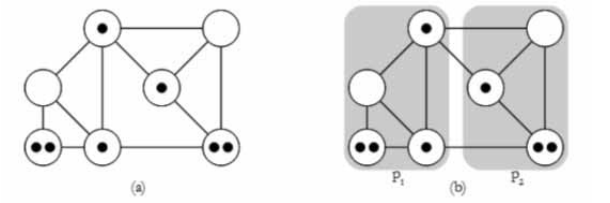

Figure 2: Static load balancing.

Before the start of a simulation run, the model is partitioned, and the resulting partial models are assigned to the participating processors (see Figure 2). A heuristic method (see [9]) is applied in this static load balancing step to reduce the number of edges between partitions, and thus to reduce the communication load. During the simulation run, tokens move from node to node and thus generate load imbalances, which have to be handled dynamically (see Figure 3).

To accomplish this, a processor p_i which is overcharged, but has a neighbor p_j which is not fully loaded, selects a number of model nodes and shifts them over to p_j (see Figure 4). This adjustment is done iteratively in the course of several simulation steps, until a stable state is reached.

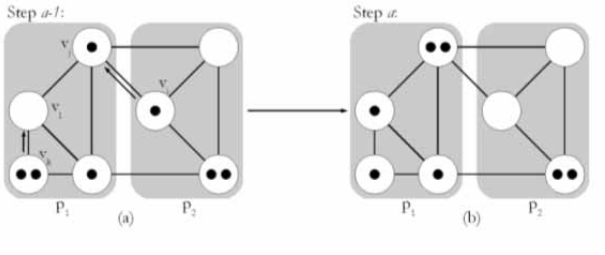

Figure 3: Dynamic load shifting.

The method uses a dynamically calculated load measure, considering both the changes in the model's activity and the time elapsing while computing, and therefore is dynamic and adaptive. It also exploits knowledge of regional dependencies to keep down communication load; existing neighborhood relations are not affected by the mechanism. Load balancing is carried out when all processors have entered the synchronization barrier (see [5], pp. 65-96), and are thus done with processing step t, but did not yet start with processing step $t+1$. The method prefers to shift model nodes from a slow processor p_s to a fast processor p_f in a way that iteratively further reduces the communication load during the simulation run. To accomplish this, it classifies each model node v in the set V_{ps} of all nodes hosted by processor p_s in one of four priority classes:

4. All $v_i \in V_{ps}$:
3. each node v_i in 4 which has an edge to a node $v_j \in V_{\acute{p}}$, with any $\acute{p} \neq p_s$;
2. each node v_i in 3 which has an edge to a node $v_j \in V_{pf}$, which is hosted by a processor p_f which is not operating at full capacity; and
1. each node v_i in 2 with a greater number of edges to nodes hosted by processor p_f than to nodes administrated by p_s.

The method prefers to shift nodes from class 1, followed by class 2 and 3. Nodes which are only members of class 4 are not shifted

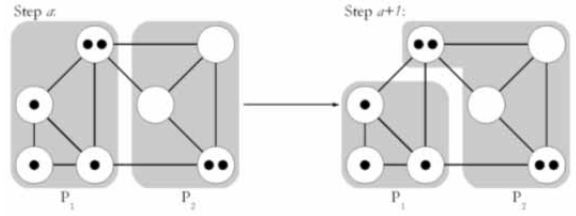

Figure 4: Dynamic load balancing.

Each simulation step consists of three phases: The computation of the model, the duration of which is dependent on the partial model's activity, and which the load balancing method tries to distribute equally over all processors; the synchronization phase, in which processors already finished with computing wait for the others to finish their tasks; and the communication and load balancing phase, in which the method shifts model nodes from fully loaded processors to underloaded ones. The load balancing step itself consists of three phases (see Figure 5): load measurement, load assessment, and load shifting. For a more detailed description of the method see [22], pp. 61-76.

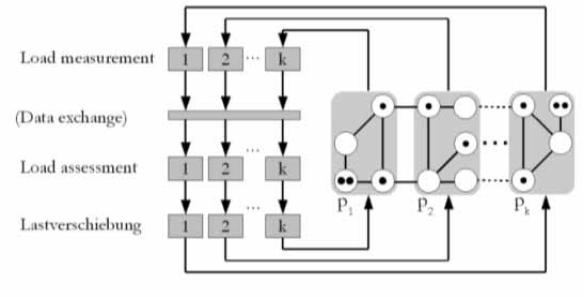

Figure 5: Phase of the load balancing process.

3 Scalability and Efficiency

As described, each simulation step i consists of three phases: computation of the model, synchronization, and communication, whose computational complexity for each processor p is denoted by $t_m(p,i)$, $t_s(p,i)$, and $t_c(p,i)$ (see Figure 6). These values can be estimated, which in turn yields estimations for the scalability and efficiency of the proposed method.

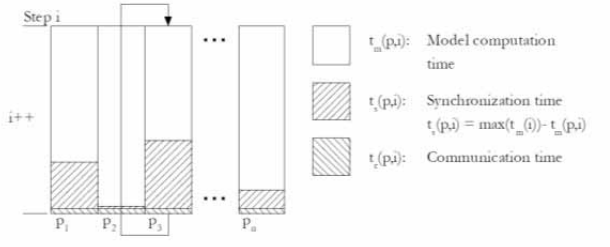

Figure 6: Phase of the simulation step.

The combined complexity $t_g(i)$ of each simulation step i in a system with k processors for a bad case (load balancing mechanism is switched off and computational loads for each processor p in each step i are drawn randomly, for a detailed description see [22], pp. 76-84) is shown in equation 1:

$$t_g(i) = \underbrace{c_m * \left(1 - \frac{1}{k+1}\right)}_{t_m(p,i)+t\_s(p,i)} + \underbrace{6 * (k-1) + c_n * \frac{k+1}{k\hat{}2}}_{t_c(p,i)} \quad (1)$$

Here, c_m denotes the computational load generated by the model, and c_n denotes the number of transient entities to be moved between model nodes.

The average time complexity $t_g(i)$ of a single step i (load balancing mechanism is switched on, computational loads are shifting smoothly through the model) is shown in equation 2:

$$t_g(i) = \underbrace{\frac{c_m}{k}}_{t_m(p,i)} + \underbrace{6 * (k-1) + c_n * \frac{k+1}{k^2} + 2 * c_l * (k-1)}_{t_c(p,i)} \quad (2)$$

Here, c_l denotes the number of resident entities which have to be transferred in the context of load balancing. The scalability of the method is thus mainly dependent on the values c_m and c_n, and in the average case also on c_l. These values are all properties of the model, and are thus not influenced by the method itself. In the average case, with the partial model's loads shifting in a benign way, the method shows linear scaling. The manifesting scaling factor for an individual model is directly dependent on the ratio of its computation load $t_m(p,i)$ to its communication load $t_c(p,i)$. Or, to put it simple: Bigger models scale better.

The expected efficiency of the method for a model with unfavorable arbitrary load imbalances can be shown to be always greater than 0.5 (see equation 3).

$$E\left(t_{busy}(k)\right) > E\left(t_{idle(k)}\right) \quad \forall k > 1 \quad (3)$$

For a detailed analysis of computational complexity, scalability and efficiency, see [22], pp. 76-86.

4 Experiments

The proposed method was implemented as a C++ framework (described in [22], chapter 4), and is utilized in two different scenarios. To keep influences of a complex real-world model with often irregular properties at a minimum, the framework is first applied to the computation of artificial loads moving through a randomly generated graph. This is followed by a real-world application in the simulation of timetable based tram traffic. Experiments are run using notebook computers with 6 gigabytes memory and an Intel Core i7-740QM processor with four cores running at 1.73 gigahertz. The turbo boost technology for accelerating single thread applications is turned off for the experiments. For experiments with more than four processor cores, several notebooks are connected by a 100 megabit ethernet switch.

4.1 Computation of artificial loads

The first set of experiments is conducted on the parallel computation of artificial load generated by token movements on a randomly generated graph. Tokens stay at a node for a certain number of simulation steps, and then move over an edge to a randomly picked neighboring node. During each simulation step, these tokens generate computational load. This is generated by executing the Sieve of Eratosthenes algorithm (see [20], pg. 85) to identify prime numbers up to an upper bound. This upper bound q_{max} is set to the sum of the base load of the node v_i and the token's weights: $q_{max} = l_{base} + l_{token} * |T_i|$.

To generate the graph, n nodes with a base load of $l_{base} = 10.000$ are generated. For each node v, two nodes $u_1 \neq v$ and $u_2 \neq u_1 \neq v$ not yet connected to v are chosen randomly. Then, two edges $v \leftrightarrow u_1$ and $v \leftrightarrow u_2$ are added to connect v to those nodes. A token is generated for every fifth node v_i with $i \bmod 5 = 0$. Each of those tokens has a weight of $l_{token} = 10.000$, a maximum retention period of $t_{max} = 100$ simulation steps, and a current retention time t_t drawn from a uniform distribution between 0 and t_{max}. This value t_t is decreased by one during each simulation step. When it reaches zero, an edge outgoing from its current host node is selected randomly; the token is then moved over this edge and is reinitialized by its new host with $t_t = t_{max}$. In the conducted experiments, each simulation run consists of 500 steps, the load balancing scheme is active.

#Proc.	Runtime (sec)	Speedup	Marg. utility
1	2,010.0	1.00	1.00
2	1,010.3	1.99	0.99
3	682.9	2.95	0.96
4	516.5	3.89	0.94
5	421.0	4.77	0.88
6	372.1	5.40	0.63
7	340.9	5.90	0.49
8	325.8	6.17	0.27

Table 1: Runtime and speedup for the computation of artificial loads.

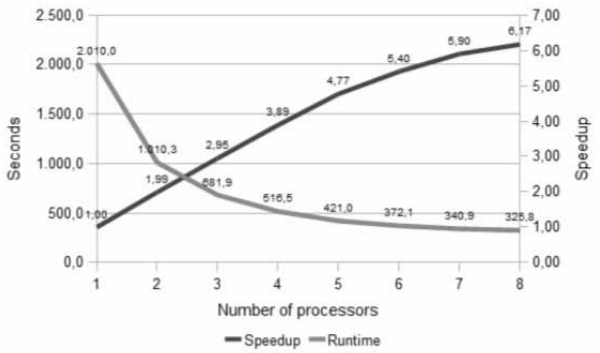

Figure 7: Runtime and speedup for the computation of artificial loads.

We begin by running a mid size instance of $n = 400$ nodes on up to eight processor cores. Average runtime, speedup values, and marginal utility are shown in Table 1 and Figure 7. A second series of asymptotic experiments begins with a graph consisting of $n = 100$ nodes and 200 edges on a single processor core, going up to $n = 800$ nodes on eight processors. Average runtime, scaling factor and marginal utility are shown in Table 2 and Figure 8.

#Proc.	Graph		Scalability		
	|V|	|E|	Runtime (sec)	Scaling factor	Marg. utility
1	100	200	495.8	1.00	1.00
2	200	400	499.1	1.99	0.99
3	300	600	523.4	2.84	0.86
4	400	800	521.4	3.80	0.96
5	500	1,000	521.7	4.75	0.95
6	600	1,200	550.5	5.40	0.65
7	700	1,400	551.8	6.29	0.89
8	800	1,600	568.4	6.98	0.69

Table 2: Runtime and scaling factor for the computation of artificial loads.

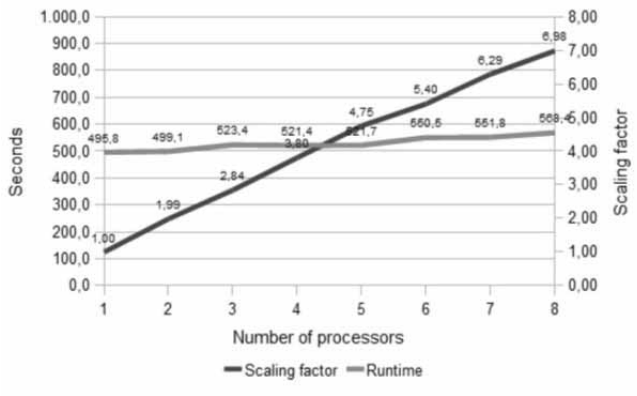

Figure 8: Runtime and scaling factor for the computation of artificial loads.

4.2 Simulation of time-table based tram traffic

The proposed method was then utilized to parallelize a sequential simulation engine of time-table based tram traffic (described in [11]). The resulting software tools were applied to the KVB network of Cologne, Germany (see [23]), and the TAM Tramway network of Montpellier, France (see [25]).

The described experiments are conducted on a model of Cologne's network. It consists of 528 platforms and 58 track switches connected via 584 tracks. These tracks cover a total length of 407.4 kilometers, resulting in an average track length of 697.6 meters. 15 lines with 182 line routes are served by 178 vehicles which execute 2,814 trips per operational day. The simulation engine was run on up to eight processors of the described type, under the parameter set described in section 4.1. Average runtime, speedup and marginal utility are shown in Table 3 and Figure 9.

4.3 Results and discussion

The simulation models are executed 10 times per measuring point. For the computation of artificial loads (see Figure 7 and Table 1), the method yields a high gain in speedup for up to five processors (speedup 4.72), which flattens when more processors are added. With partial models getting smaller, the ratio of synchronization and communication time to model computation time rises, so efficiency is declining. For the asymptotic experiments (see Figure 8 and Table 2) a linear regression yields a function $s(k) = 0.86 * k + 0.27$ for scaling, and $T(k) = 10.16 * k + 483.3$ for run time. Under the described conditions the method thus shows a linear scalability with a scaling factor of around $0.86 * k$.

The simulation of time-table based tram traffic shows mixed results (see Figure 9 and Table 3): For up to four processor cores - based on a single parallel computer - the speedup rises to 2.83, and then caves in to 1.89 when the fifth processor - connected via LAN - is added. The reason for this behavior is the significantly higher communication cost between LAN connected computers in relation to communication between parallel processor cores. The ratio of high communication cost to a relatively low computation cost for the distributed partial models forbids an effective execution on LAN connected computers. A linear regression for the first four measuring points yield a function of $z(k) = 0.62 * k + 0.5$ for speedup, and $T(k) = -55.03 * k + 289.5$ for run time. The last four points yield functions of $z(k) = 0.12 * k + 1.3$ and $T(k) = -7.71 * k + 177.3$. The model instance is therefore large enough to be efficiently run on a parallel computer, but too small to be executed expediently on LAN connected computers. For a more in-depth discussion of the experiments see [22], chapters 5 and 6.

#Proc.	Runtime (sec)	Speedup	Marg. utility
1	263.8	1.00	1.00
2	144.5	1.82	0.82
3	106.4	2.44	0.62
4	93.1	2.83	0.39
5	139.8	1.89	-0.94
6	130.6	2.02	0.13
7	121.0	2.12	0.10
8	117.3	2.25	0.13

Table 3: Runtime and speedup for the simulation of time-table based tram traffic.

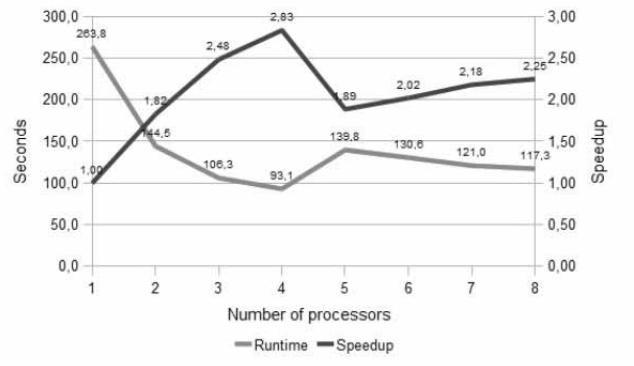

Figure 9: Runtime and speedup for the simulation of time-table based tram traffic.

5 Summary and Further Research

We presented an approach to the parallel execution oftraffic simulation models, which includes a dynamicand adaptive load balancing scheme. Some thoughtson scalability and efficiency were shared: even underadverse circumstances the efficiency does not get lower than 0.5. Experiments conducted on the computation of artificial loads yield a speedup of 3.89 on four processors and 6.17 on eight processors. Parallel execution of a tram traffic model shows a speedup of 2.83 on four processor cores. A higher speedup might be reached when executing a larger model. In a next step, the multimodal model will be extended by a representation of bus transit.

Acknowledgements

This material is partially based upon work supported by the National Science Foundation under grants I/UCRC IIP-1338922, AIR IIP-1237818, SBIR IIP- 1330943, III-Large IIS-1213026, MRI CNS-0821345, MRI CNS-1126619, CREST HRD-0833093, I/UCRC IIP-0829576, MRI CNS-0959985, FRP IIP-1230661, and U.S Department of Transportation under a 2013 TIGER grant.

This contribution is a post-conference publication from ASIM SST 2014 (22nd Symposium Simulationstechnik, Berlin, September 3-5, 2014, HTW Berlin). The original contribution was published in Tagungsband ASIM 2014, 22. Symposium Simulationstechnik, J. Wittmann and Ch. Deatcu (Eds.), ASIM Mitteilung 151, ARGESIM Report 43, ISBN 978-3-901608-44-5, ARGESIM Verlag, Wien, 2014.

References

[1] Avril H, Tropper C. The Dynamic Load Balancing of Clustered Time Warp for Logic Simulation. In: *Proceedings of the tenth workshop on Parallel and distributed simulation*; 1996 May; P. 20-27

[2] Bryant RE. Simulation of Packet Communication Architecture Computer Systems. Computer Science Laboratory. Cambridge, Massachusetts, Massachusetts Institute of Technology, 1977. Report

[3] Chandy KM, Misra J. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering. 1979;* SE- 5(5): 440-452.

[4] Chandy KM, Misra J. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*. 1981; 24(4): 198-205

[5] Fujimoto RM. *Parallel and Distributed Simulation*. New York: John Wiley & Sons, 2000

[6] Greenberg AG, Lubachevsky BD, Mitrani I. Algorithms for Unboundedly Parallel Simulations. *ACM Transactions on Computer Systems*: 1991; 9(3): 201-221

[7] Heidelberger P, Stone H. Parallel Trace-Driven Cache Simulation by Time Partitioning. In: *Proceedings of the 1990 Winter Simulation Conference*. 1990; 734-737

[8] Jefferson DR. Virtual Time. *ACM Transactions on Programming Languages and Systems.* 1985; 7(3): 404-425

[9] Kernighan BW, Lin S. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell Syst. Tech Journal*.1970; 49(2): 291-307

[10] Lückerath D. Thoughts on restauration of regular tram operation. In: *Proceedings of Sommertreffen Verkehrssimulation 2012*, AM 143, ARGESIM/ASIM Pub., TU Vienna, pp. 4-6, 2012.

[11] Lückerath D, Ullrich O, Speckenmeyer E. Modeling time table based tram traffic. *Simulation Notes Europe.* 2012; 22(2): 61-68

[12] Lückerath D, Ullrich O, Speckenmeyer E. Applicability of rescheduling strategies in tram networks. In: *Proceedings of ASIMWorkshop STS/GMMS 2013*. ARGESIM Report 41, AM 145, ARGESIM/ASIM Pub., TU Vienna, 2013.

[13] Meisgen F. Dynamic Load Balancing for Simulations of Biological Aging. *International Journal of Modern Physics C*. 1997; 8(3): 575-582

[14] Meisgen F. *Dynamische Lastausgleichsverfahren in heterogenen Netzwerken*. Aachen: Shaker Verlag, 1998.

[15] Merz M, Bröcker E.: Einsatz von Open Source Frameworks zur Parallelisierung von Dymola Simulationen. In: Proc. ASIM/GI Workshop STS/GMMS, Ed.: W. Commerell, ISBN 978-3-9810998-3-6, Ulm, März 04-05, S. 283-289, 2010.

[16] Nagel K, Schreckenberg M. A cellular automaton model for freeway traffic. *Journalde Physique I*. 1992; 2(12): 2221-2229

[17] Nicol DM. The cost of conservative synchronization in parallel discrete event simulations. *Journal of the Association of Computing Machinery*. 1993; 40(2): 304-333

[18] Rönngren R, Ayani R. A Comparative Study of Parallel and Sequential Priority Queue Algorithms. *ACM Transactions on Modeling and Computer Simulation*. 1997; 7(2): 157-209

[19] Schlagenhaft R. Dynamischer Lastausgleich optimistisch synchronisierter, verteilter Simulation. In: *Proc. ASIM-Workshop VSPP*. 1999

[20] Sedgewick R. *Algorithms in C++, Vol. 1.* Boston: Addison-Wesley, 1998.

[21] Steinman J. SPEEDES: *Synchronous parallel environment for emulation and discrete event simulation.* Advances in Parallel and Distributed Simulation, SCS Simulation Series.1991; Vol. 23: 95-103

[22] Ullrich O. *Modellbasierte Parallelisierung von Anwendungen zur Verkehrssimulation - Ein dynamischer und adaptiver Ansatz.* Dissertation, Univ. Köln, 2014.

[23] Ullrich O, Lückerath D, Franz S, Speckenmeyer E. Simulation and optimization of Cologne's tram schedule. *Simulation Notes Europe* . 2012; 22(2): 69-76

[24] Ullrich O, Proff I, Lückerath D, Kuckertz P, Speckenmeyer E. Agent-based modelling and simulation of individual traffic as an environment for bus schedule simulation. In: *mobil.TUM 2013*. Proceedings of mobil.TUM 2013. 2013 Jun

[25] Ullrich O, Lückerath D, Speckenmeyer E. A robust schedule for Montpellier's Tramway network. In: *ASIM 2014*.Proceedings of ASIM 2014 - 22nd Symposium on Simulation Technique; 2014 Sept, Berlin

[26] Zeng AZ, Durach CF, Fang Y. Collaboration decisions on disruption recovery service in urban public tram systems.*Transportation Research Part E*.2012; 48(3): 578-590