# An Introduction to Parallel Discrete Event Simulation

Oliver Ullrich*, Daniel Lückerath

Fraunhofer Institute for Intelligent Analysis and Information Systems IAIS, Schloss Birlinghoven, 53757 Sankt Augustin, Germany; *oliver.ullrich@iais.fraunhofer.de

SNE 29(2), 2019, 63-72, DOI: 10.11128/sne.29.on.10471
Received: May 10, 2019
Accepted; May 31, 2019
SNE - Simulation Notes Europe, ARGESIM Publisher Vienna, ISSN Print 2305-9974, Online 2306-0271, www.sne-journal.org

**Abstract.** Some discrete simulation models are too large to be executed on a single processor; in other cases, results might be required faster than a sequential execution can provide them. Such models are candidates for parallelization. Here, models are distributed among several processors, and are then executed with careful synchronization.

This paper provides an introduction to the fundamentals and methods of the parallel execution of simulation models, with a focus on model-based parallelization. The paper describes the two main classes of parallel simulation methods, conservative and optimistic simulation, their respective advantages and shortcomings. A second focus is put on static and dynamic load balancing, with a dynamic load balancing method first developed to accelerate the simulation of transportation systems being introduced in some detail. In addition, the paper describes some typical applications of model-based parallelization.

## Introduction

Many discrete simulation models contain a certain degree of inherent concurrency. For example, in the simulation of a light rail network the braking manoeuvres of one vehicle in one region of the network would not directly influence the passenger exchange of a different vehicle in another region. The two vehicles can thus be simulated independently of each other in the majority of cases.

The goal of model-based parallelization is to exploit that existing concurrency through parallel execution of events that take place in different regions of the model on a number of participating processors or processor cores. The basic assumption is that these events can often be executed independently of each other without inducing communication between partial models. During the course of the execution, synchronization issues may arise between these partial models. For example, if a vehicle entity leaves the partial model of one processor it has to be sent to another processor and there has to be integrated with the partial model already being executed.

This paper presents an introduction to background, approaches, and techniques for the parallel execution of simulation models, with a focus on model-based parallelization. It introduces a dynamic load balancing method first developed for the efficient execution of multimodal transportation models. The paper is especially addressed to students of the craft, and to practitioners who might want to look beyond the GUI of their usual modeling tools. While in many cases parallelization methods are hidden in the execution engine of a simulation tool, some applications call for a more hands-on approach. The fundamentals of parallel simulation are easily understood, and its methods are also very powerful. Researchers, students, or practitioners can utilize well-researched parallelization methods to create fast simulation applications executing large models.

The paper continues by sharing some background on the concepts and general approaches to parallel simulation (see Section 1), and then goes on to describe the two main classes of model-based parallelization techniques: conservative and optimistic methods (see Section 2). Subsequently, static and dynamic load balancing approaches are described (see Section 3), followed by an examination of some typical applications (see Section 4). The paper closes with a summary of the lessons learned and recommendations for further reading (see Section 5).

## 1 Background

Discrete simulation models consist of a set of entities that represent physical or logical components of the examined system, including their behavior and relationships to each other and their state changes over time.

In discrete simulation, a model changes its state at discrete points in simulation time. Here, simulation time – or model time – is the time that elapses from the point of view of the simulated entities (see [1]). Simulation time has to be distinguished from wallclock time, the time elapsing in the real world while the simulation run is executed. In many cases simulation models are executed as fast as possible. In certain applications, however, it is desirable to tie model execution to wallclock time, for example if a human has to react to the changes in the model. This is referred to as real-time execution or scaled real-time execution.

Simulation time can progress in fixed or variable increments. In models with fixed time increments, the model is executed by starting out from simulation time $t_{start}$, iterating through steps $i$ with a fixed model time increment $\Delta t$ – the model state can change at any of these points $t_{start} + i * \Delta t$ in simulation time. The entities communicate with each other via messages that might be scheduled with a timestamp in the (model time) future.

With models with a variable time step, simulation time is incremented while processing simulation events. These methods are often called event-based simulation (see [1] or [25]). Each of these events has a timestamp that marks the scheduled time of its occurrence, and often also an attribute that describes the type of the event and various other fields such as a list of the intended receivers and the identity of the sending entity. The events are managed in a Future Event List (FEL), a priority queue that keeps all scheduled events sorted in ascending order of their timestamp. To execute the model, the event with the least timestamp is pulled from the FEL, the simulation time is advanced to its timestamp, and the event is processed – which usually changes the model state. New events can be scheduled during processing; they are then inserted into the FEL.

In order to accelerate model execution, computation can be distributed over parallel processes, for example on several processors or, more and more often, several cores of the same processor. Here, usual goals are to execute the model as fast as possible or in (scaled) real time. An execution that is too fast for a desired real-time binding can be slowed down to the desired speed without any problems.

A central condition for such a parallel execution is that a simulation run in parallel has to deliver identical results as a sequential execution of the same model; the simulation technique must not influence the model behavior (see [6] or [19]).

The central measure for the efficiency of a parallelization method is the speedup. That value determines the ratio of the runtime of the sequential execution of a model to the time needed for parallel execution. The aim of parallel execution is to achieve the highest possible speedup with a given number of processors, or, more precise, given computational resources.

A number of vastly different approaches to parallel simulation exist. For an in-depth discussion of the methods described below – and more – see [6]. *Model-based parallelization* methods, also called space-based parallelization, aim to exploit the parallelism inherent in the model. For this purpose, the model is decomposed into partial models, which are then distributed on the available processors for execution. The different partial models communicate via messages encapsulating simulation events or serialized entities that are sent over the shared cache or the connecting network. The processors $p_1$ to $p_k$ from the set of processors $P$ each execute a specific partial model – they can be seen as the nodes of a graph, with the messages sent between processors inducing edges.

Any model-based parallelization method has to keep the execution of partial models carefully synchronized. Here, the *local causality constraint* prescribes that each model entity has to process simulation events in a non-descending order regarding their timestamps. If the local causality constraint is not met, the simulation results might be invalidated by causality errors. For example, lets assume that in a light rail simulation a processor $p_1$ has processed an operational day up until a simulation time of 12:30, while a processor $p_2$ has only arrived at 12:05. Now a vehicle entity leaves the partial model of $p_2$. That processor sends a message to $p_1$ and transfers the vehicle data for further simulation from 12:06. From the point of view of $p_1$ that message comes from 24 simulation minutes in the past. During these 24 simulated minutes $p_1$ might have already allocated the resources "rightfully" occupied by the vehicle to other entities. The transferred message can not be processed sensibly; the simulation has to terminate with an error message.

A central concept is the lookahead (see [5] or [13]): If an entity or a partial model is currently processing events at a simulation time $t$, then a lookahead of $L$ guarantees that no additional simulation events will be generated with a timestamp lesser than $t+L$ (see [6]). For models with a fixed time increment the lookahead corresponds of that increment and is therefore always greater than zero. For event-based models the lookahead usually changes in the course of the simulation; under certain circumstances a lookahead value of zero is possible (see Section 2.1).

## 2 Model-based Parallelization

Model-based parallelization methods can be categorized based on the way the causality constraint is kept: With conservative methods, causality is guaranteed at all times by only processing simulation events that are explicitly considered safe. With optimistic methods, each entity indiscriminately executes events as quickly as possible. In case a processor receives an event with a timestamp that lies in the past from its local point of view, it rejects corresponding parts of the already executed simulation and restores causality by recalculating them from the timestamp of that event on.

In the following, a selection of important conservative and optimistic parallelization methods are described.

### 2.1 Conservative Parallelization Methods

Two of the most important conservative parallelization methods are synchronization with null messages and synchronous execution. Both methods – and more – are described in great detail in [6].

**Synchronization with null messages.** Synchronization with null messages was independently developed as the first conservative parallelization method for event-based simulations by Bryant (see [2]) and Chandy and Misra (see [3]) and explained in detail by Fujimoto (see [6]). Here, the processors $p_1$ to $p_k$ are regarded as nodes of a graph. In that graph, if a processor $p_i$ sends messages to a processor $p_j$ in the course of the simulation run, a directed edge exists between these nodes.

The method assumes that a processor $p_i$ sends messages to a processor $p_j$ in order of non-decreasing timestamps. A processor stores incoming messages in a series of FIFO queues, each assigned to an incoming edge of the processor graph. It follows that messages are present in each of these queues in non-decreasing order. In a model with variable time progress, messages or events that stay local on one processor are managed in a separate priority queue.

A message with timestamp $t$ is declared secure if there is at least one message with a timestamp not lower than $t$ at the head of each inbound queue. The presence of these messages means that no processor can send messages that lie before $t$ in simulation time. Now the processor selects the message $N_{min}$ with the lowest timestamp from all incoming queues and, if applicable, the local event list. Since no message with a lower timestamp can subsequently occur, the causality condition is maintained when $N_{min}$ is processed.

During event processing, further events with the same or a greater timestamp are sent to neighboring processors if necessary. At that point deadlocks can occur: If for every participating processor not all queues at the incoming edges are filled with at least one message, each processor waits for messages from the other processors to arrive (see Figure 1). Therefore, no events can be declared safe – the simulation is blocked.

To solve this problem, Bryant (see [2]) and Chandy and Misra (see [3]) suggest that each processor, after processing a message, sends so-called null messages to all neighboring processors. These messages are timestamped with the current simulation time plus the lookahead value $L$ of the processor.

The handling of null messages by the receiving processor is the same as that of regular messages. However, when processing a null message, no change is made to the model state apart from advancing the simulation time to its timestamp. Sending null messages during each event processing ensures that messages are always available in the FIFO queues – the development of a deadlock is thus precluded.

The efficiency of the method largely depends on the lookahead value: A small lookahead means that many null messages have to be sent and processed. In addition, the model cannot contain any circles in the graph with a lookahead of $L=0$, otherwise deadlock situations become possible. Here, the processors involved process only null messages and send (because of $L=0$) further null messages with the same timestamp to each other. The simulation time never advances, the application is caught in an endless loop.

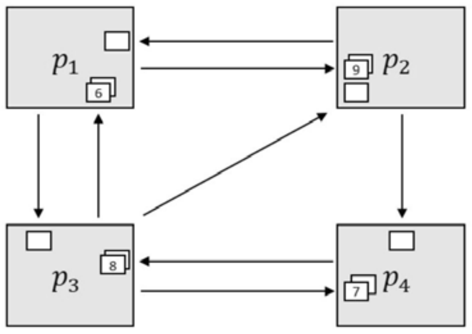

**Figure 1:** A deadlock occurs when each processor is waiting for its neighbors to send messages; sending null messages can solve this issue.

**Synchronous execution.** In synchronous execution (see [6]), each processor executes the events or simulation steps of a simulation time interval recognized as safe and then enters a synchronization barrier. Here, each processor waits for all other processors to complete calculation. Then the next interval, now declared safe, is processed. Thus, there is a defined point in wall-clock time when all processors have finished calculating a certain simulation time interval and before any of them starts calculating the next time interval.

To determine what events are safe to be executed, the lookahead $L(i)$ for a step $i$ is used. While $t(i)$ is defined as the simulation time of the next unprocessed message at processor $p$, $L(i)$ is its local lookahead value. The global lookahead $t_L$ is the minimum value of $t(i)+L(i)$ for all processors. All messages with timestamps of up to $t_L$ are then declared safe.

The synchronization barrier can be implemented in different ways. When synchronizing with tree barriers, the processors are regarded as a balanced span tree, with one processor being designated as a controller. A leaf processor that has completed the calculation step and now wants to enter the barrier sends a barrier message to its parent node in the tree and then waits for a response. An inner node that wants to enter the barrier waits for messages from its daughter nodes. If these are complete, it sends a barrier message to its parent node and then waits for the response. When the controller has finished calculating the interval and has received barrier messages from all daughters, all the processors have reached the barrier phase. To then leave the barrier and initiate the next calculation phase, the controller sends release messages to its daughters, who in turn send them on to their daughters.

A special case of tree barriers is the so-called central barrier. Here all processors are synchronized directly by a controller. The disadvantage of the otherwise very efficient central barrier is the linear growth of the number of messages that have to be processed by the controller, leading to a bottleneck when a large number of processors is involved.

The synchronization messages can be utilized for sending piggybacked data values, such as local lookahead values. The method calls for no other prerequisites than the presence of a positive lookahead for determining the size of the simulation time increments. In particular, there are no requirements for the connections between the individual partial models, since the presence or fill level of FIFO queues need not be taken into account.

## 2.2 Optimistic Parallelization Methods

The optimistic method Time Warp was first proposed by Jefferson (see [11]) and is described in detail by Fujimoto (see [6]). During the 1990s the method matured with modifications that improve memory consumption (see [22]) as well as reduce costly rollbacks (see [4] and [23]). In Time Warp the parallelization tasks are divided into a local and a global control mechanism. The work carried out by the local mechanism takes place locally on each processor – the processors can work largely independently of each other. The global mechanism performs activities such as input, output, and garbage collection, and requires coordination between the processors.

**Local control mechanism.** As with other event-based methods, processors execute events from the local Future Event List (FEL), and the state variables of the model are changed if necessary. However, the events are not simply discarded after processing, but stored in another list, the Processed Event List (PEL).

If a message arrives from another processor whose timestamp is greater than or equal to the current simulation time of the local partial model, it is inserted into the FEL and processed normally. If a straggler message $N$ arrives with a timestamp $t$ lesser than the local simulation time, the model has to be rolled back to its state at time $t$ – all state changes from this point on have to be undone. Furthermore, the already processed events with a timestamp greater than $t$ have to be retrieved from the PEL and inserted back into the FEL for reprocessing. Message $N$ is also inserted in the FEL.

There are two general ways to perform this rollback: In copy state saving, the values of all state variables are saved before each event processing. If a straggler arrives, the saved state with the corresponding timestamp is copied back to the state variables – later changes are discarded. With incremental state saving, a log entry notes each change of a state variable. That log entry is then put on a stack keeping a record of all changes.

It may not be enough to reset the local model state: If invalidated messages were sent to other processors, these messages have to be retrieved – or “unsent” – and their effects have to be undone by the receiving processor. Time Warp uses so-called anti-messages for this purpose. Each anti-message $N_A$ corresponds to exactly one regular message $N$. When an anti-message arrives at a processor, the corresponding regular message is automatically deleted from the corresponding data structure (FEL or PEL). The anti-message is also destroyed.

That mechanism elegantly undoes all invalidated changes and restores causality, but also might lead to a cascade of rollbacks and anti-messages.

Rollbacks do not affect the model state at a time less than or equal to $t$, i.e. simulation results up to the simulation time of the straggler are retained. It follows that at least the processing of the event with the system-wide least timestamp will not be cancelled. There is therefore a minimum simulation time that might be affected by potential rollbacks – the state of the model before that simulation time will never be invalidated.

**Global control mechanism.** The Global Virtual Time $GVT_t$ at a time $t$ denotes the minimum timestamp of all unprocessed or partially processed events across all processors involved at a time $t$. As already described, no rollbacks can take place to times times lesser than $GVT_t$.

When calculating $GVT_t$, messages must be taken into account that have already been sent but not yet received by the recipient. Since these transient messages can potentially trigger a rollback and thus reduce the local simulation time, the minimum of local simulation times cannot simply be determined. As a remedy, a simple protocol can be used in which each recipient of a message $N$ confirms the reception to the sender. Until this acknowledgement is received, the sender is responsible for the message $N$ and has to include it in the calculation of the local minimum – afterwards $N$ becomes the responsibility of its receiver. This guarantees that at any point in time the simulation time of $N$ is included in the calculation of $GVT_t$.

The value of $GVT_t$ is used for a number of administrative tasks, for example the collection of fossil states: backup copies older than it can safely be deleted. As input/output operations generally cannot be undone, simulation events can only order outputs when the current $GVT_t$ has advanced to at least the simulation time of the event. A special case is the processing of program and calculation errors: These can occur due to causality errors, for example a negative number of trains in a depot. The program cannot simply be terminated, as such errors may be reversed by rollbacks.

## 2.3 Comparison

The best method for the parallel execution of an individual model is largely dependent on its specific properties; no single method is optimal for all applications (as analyzed in detail in [6]).

Generally, conservative methods tend to be less complex in structure (see [9] and [15]). They work with only a single set of state variables, without the need to manage backups. Since conservative methods only execute events or time increments that are explicitly declared safe – they are based on worst-case scenarios –, they do not fully exploit the parallelization potential of a model. Conservative methods can therefore be excessively pessimistic. In general, the greater the lookahead value, the more events can be processed in parallel, so that a higher degree of model-inherent parallelism can be exploited.

An advantage of optimistic methods is that even models with a lookahead of zero can be efficiently executed without further restrictions. Parallel execution is not hindered by all potential dependencies between partial models, as is the case with conservative methods, but only by dependencies that actually occur in the course of a run.

If these dependencies are high, or if the computational loads shift over time, for example resulting from dynamically changing activities in the model, these methods might behave too optimistically, so that a cascade of miscalculations is carried out, that then have to be taken back by complex rollback operations (see [14]). To ensure causality, backups of the model state are necessary for each occuring change. For activities such as input/output, error handling, or memory management, for which the usual library functions can be used in conservative methods, optimistic procedures require specifically implemented rollback-safe functions.

In summary, optimistic methods tend to be more complex than conservative methods. The lower overhead of conservative methods has a positive effect on performance, especially when the lookahead is known – and ideally large in comparison to the event density. However, if a lookahead value is not known or is very low compared to the event density, optimistic methods generally have performance advantages.

## 3 Load Balancing

Resulting from the typical dependencies in simulation models, the speed of the execution is generally dependent on the partial model that advances most slowly in simulation time. It is therefore beneficial to incorporate a load balancing system into the simulation engine. Such a system does not exclusively aim at high utilization of the processor capacity, but also has to consider a uniform advance in simulation time.

Load balancing schemes employed by parallel simulation methods can be characterized as dynamic, static, adaptive, non-adaptive, local, centralized, or hierarchical (see [16]): A *static* method estimates the load and assigns partial models to processors in a preprocessing step before the start of the simulation run, and thus does not consider dynamic changes in the model activity. In contrast, a *dynamic* load balancing method continuously considers imbalances that develop from shifts in the computational load and re-assigns partial models to appropriate processors while executing the simulation run. *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* system ignores those fluctuations and differences. In *local* methods, the processors only exchange data with determined neighborhoods and act on this local information, while *centralized* methods utilize a marked controller process to whom the other processors report. *Hierarchical* methods usually organize communication in a tree topology.

A load balancing method used on a PC or laptop computer should have static and dynamic components, with the latter being also adaptive, and thus consider both the changing computational load of the models, and the changing availability of resources on a non-exclusively used machine. A centralized method is usually simpler to implement and quite adequate for a system with only eight to sixteen processor cores (see [27]); if a method is targeted at a massive parallel system it should avoid a potential bottleneck by utilizing a hierarchical or local scheme.

### 3.1 Static Load Balancing

At the start of a simulation run, the model entities should be assigned to the participating processors in a way that ensures a balanced load. As a second objective to optimally using the computational potential of the processors, the communication load, resulting from sending and receiving messages from one processor to another, shall be as low as possible. Without further knowledge of model specifics, the static load balancing mechanism uses the number of edges between model partitions as an indicator for communication load. It therefore aims to distribute the model in a way that keeps the number of inter-partition edges at a minumum.

In literature, the decomposition of a graph $G(V,E)$ with $n = |V|$ nodes into $k$ components of simular size is known as the GRAPH PARTITION problem. GRAPH PARTITION is $NP$ complete (see [10]), and can thus – in case $P \neq NP$ holds – not be solved efficiently. For the parallelization of simulation models an exact solution is not necessary, especially since a dynamic change of the load in the course of a simulation run would quickly destroy any optimum static load balance (see [24]).

Kernighan and Lin (see [12]) describe a simple heuristic method suitable for static load balancing. The method starts out from a given partition where all partial models have the same number of nodes (give or take one) – for many models that may be a simple geographical breakdown. The method then iteratively improves the communication load using a hill climbing algorithm (see [18]).

Kernighan and Lin first describe a method to decompose a graph into two partitions $K_1$ and $K_2$. Starting out from a given initial partition the method computes for each pair of nodes $(v_i, v_j)$ with $v_i \in K_1$ and $v_j \in K_2$, the impact of a potential movement of $v_i$ to $K_2$ and $v_j$ to $K_1$ on the number of inter-partition edges. In case an improvement is possible, the nodes are moved accordingly. The method iterates as long as additional improvements are possible, and thus until a local optimum is reached. It has a computational complexity of $N(n^2)$.

The described method is then extended to disect a graph into $k > 2$ partitions: To that effect each pair $K_i$

and $K_j$ out of the $k$ partitions are locally optimized using the $k = 2$ method. Usually several iterations are executed, so that the simple method is run $e * (k-1) * (k-2) = O(k^2)$ times. That results in a computational complexity of $O(k^2 * (n/k)^2) = O(n^2)$ for $k$ partitions with $n/k$ nodes – the method is thus independent of the number of participating processors. Kernighan and Lin empirically determine that after two iterations of their method approx. 95% of the potential gain has been reached.

## 3.2 Dynamic Load Balancing

In many models the majority of inter-entity dependencies are regional in nature: most of the time entities interact with their neighbors, only rarely do they send messages to far away regions of the model. Based on that thought dynamic load balancing generally has two aims:

- ensuring that all participating processors progress uniformly in simulation time by computing loads adequate to their respective performance, and
- keeping the communication load between the processors as low as possible by exploiting regional dependencies in the model.

Generally, the processors perform the load balancing in three steps: *load measuring*: each processor $p_i$ determines its own load and communicates the results to the other processors; *load evaluation*: each processor $p_i$ determines whether any model nodes shall be migrated to adjacent processors and, if so, what nodes shall be migrated to what processor $p_j$; and *load migration*: the model nodes are encapsulated as messages and sent to adjacent processors.

The dynamic load balancing mechanism described here has been first developed for the parallel simulation of transit systems as part of a conservative, synchronous execution engine (see [24] and [27]).

**Measuring loads.** To be able to employ effective countermeasures against overload or underload the load of individual processors has to be measured in regular intervals. In conservative parallelization that can be achieved for example as part of the synchronization barrier, in optimistic methods as part of the local control mechanism. The following describes a comparatively simple way to measure load as part of the synchronization barrier that also integrates the available individual performance of a processor – including its change over time, for example through external user or system processes (see [27]).

Each processor $p \in P$ measures its load $l_p(i)$ at time $t(i)$ when all processors have completed their computations regarding simulation step $i$ (see Figure 2). By utilizing the timer functions of the operating system each processor $p$ measures the model-dependent processing time $t_m(p,i)$ it needs to execute the simulation step, and the duration of synchronization time $t_s(p,i)$ that elapses between the completion of the execution and $t(i)$. The load $l_p(i)$ of the processor at step $i$ is now defined as

$$l_p(i) = \frac{t_m(p,i)}{t_m(p,i) + t_s(p,i)} \tag{1}$$

The still available capacity that was wasted as idle time can be determined as

$$f_p(i) = \frac{t_s(p,i)}{t_m(p,i) + t_s(p,i)} \tag{2}$$

The total time $t_g(i)$ used to execute simulation step $i$ is now composed of the processing time, the synchronization time and the communcation time $t_c(p,i)$ used to load balancing and other administrative work (see Equation 3). The values of $t_g(i)$ are equal for all $p \in P$.

$$t_g(i) = t_m(p,i) + t_s(p,i) + t_c(p,i) \tag{3}$$

Based on local load data alone a value of $l_p(i)$ near 1 – and thus a synchronization time $l_s(p,i)$ near 0 – can signal either an optimum load near capacity or a bottleneck caused by overload. To be able to discern, the load data of the other processors in $P$ has to be included. That data basically consists of two numbers that can be exchanged as part of the synchronization process.

The load measurement based on $t_s(p,i)$ takes into account internal and external disturbances, it considers both the progress of simulation time (which in the described, simple case is fixed) and the change of available computing power over time. Based on that load measurement method a dynamic and adaptive load evaluation can be performed.

**Evaluating loads.** During the load evaluation step, each processor $p$ has to decide whether load balancing has to be performed at all, and if so, how many and which nodes are to be migrated.

Moving model nodes requires computing time and network resources. To avoid over-reaction caused by

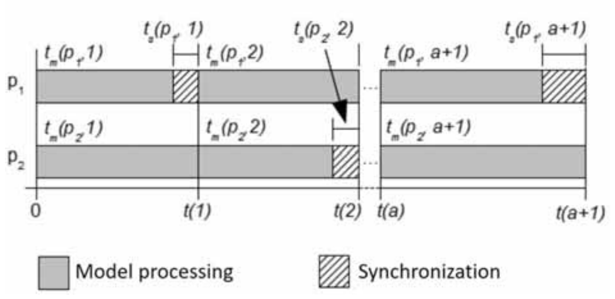

**Figure 2:** Load measurement on a system with two processors.

only short-term load imbalances, a smoothed value $s_i$ of the synchronization time $t_s(p,i)$ is considered (see Equation 4) when deciding whether load balancing should take place. Nodes are only migrated if $s_i$ is below a threshold value $\beta_i$.

$$s_i = \alpha * t_s(p,i) + (1-\alpha) * s_{i-1} \tag{4}$$

An effective method has to prevent overcompensation occuring due to long network runtimes or from attempting to balance even very small imbalances. To avoid thrashing, i.e. nodes being repeatedly sent back and forth between two processors, the value $\beta_i$ is not constant, but changes over the course of the simulation run between limits $\beta_{min}$ and $\beta_{max}$: If load movements have been performed in step $i$, the threshold value is decreased: $\beta_{i+1} = max(\beta_i/\gamma; \beta_{min})$, with $\gamma \geq 1$. Further movements are therefore only performed if processor $p$ is heavily overloaded. If no load balancing has been performed for a while, the threshold value is increased: $\beta_{i+1} = min(\beta_i * \gamma; \beta_{max})$.

The number $\delta$ of to be migrated nodes from the set of all nodes $V_{p_s}$ managed by the sending processor $p_s$ is determined as $\delta = \max(1, \lfloor |V_{p_s}| * \varphi \rfloor)$, with $\varphi$ being the ratio of nodes to be moved. Candidates are those nodes that have at least one edge to a node $v_j$ managed by any other processor $p(v_j) \neq p_s$.

The method preferredly (priority 1) selects those nodes $v_i$ for movement to a target processor $p_f(v_i)$ where the number of edges $(v_i, v_j)$ to nodes $v_j$ with $p(v_j) = p_f(v_i)$ is greater than the number of edges to nodes $v_k$ with $p_s = p(v_k)$. That processor $p_f(v_i) = p(v_j)$ then is the target of a potential migration. In addition, any node $v_i$ that has at least one edge to a node $v_j$ managed by a processor $p(v_j) \neq p_s$ not currently running at full capacity can also be moved (priority 2).

Moving a priority 1 node $v_i$ to processor $p_f(v_i)$ reduces the number of edges between model partitions. The load balancing method therefore does not only distribute the computing load evenly, but also reduces the expected communication load.

**Moving loads.** The load movement itself takes place during a defined time when all processors pause model computation. For optimistic methods that would be during the control mechanism, while conservative methods using barriers typically utilize the synchronization step. At that time changes can be made to the model graph without having to regard ongoing simulation calculations.

Here, each processor $p_s(v)$ encodes each model node $v$ to be relocated as a message $N_v$, then sends it through the common cache or over the network to the corresponding target processor $p_f(v)$ and, if necessary, informs third processors that contain nodes with edges to $v$ of its relocation. Each received message $N_v$ is decoded and converted to a new node $v$, which is integrated into the partial model administrated by $p_f$.

## 4 Applications

Since their inception, a large number of applications of model-based parallelization and load balancing methods have been developed. A few typical applications reported on during the years are presented below.

**Simulation of Electronic Circuits.** Schlagenhaft et al. (see [21]) and Schlagenhaft (see [20]) describe a method to parallelize the simulation of the dynamic behavior of logical circuits. Their event-based model is parallelized using the optimistic Time Warp method. The executing processors are not exclusively available to the application, but are also used by third-party processes. The modeled logical circuits consist of switching elements between which dependencies in the form of binary signals exist. In the model, each switching element is mapped as an entity; these are combined into clusters by statically partitioning the model at the start of the simulation run; the clusters are then joined together to form partitions that are then assigned to the individual processors. These clusters are managed individually, with each cluster having its own FEL. Thus, clusters can be moved during load balancing. A further advantage of dividing the partitions into individual clusters comes into play in case of a rollback: Here, the simulation does not have to be wholly discarded and recalculated for the entire partition, but only for a few clusters – or even only for a single one.

Schlagenhaft, et al. describe a dynamic and adaptive load balancing method to utilize the available resources in the best possible way. The global control mechanism is extended by a load balancing method that can move individual clusters between partitions. Load measurement, load evaluation and load shift are performed as part of the Global Virtual Time $GVT_t$ calculation mechanism. When used on two processors and with a load from external processes (see [21]), the load balancing procedure improves the runtime by approx. 24%. Schlagenhaft (see [20]) reports on improvements of up to 60% when using six processors in networks with external loads.

**Simulation of Social Interactions.** Permalla (see [17]) presents a parallel discrete event model of the Naming Game, a sociological model of social interactions and consensus building without a central coordinating instance. They utilize the concurrency inherent in the model to implement an efficient application based on a parallel discrete event simulation framework. Analogous to the work of Schlagenhaft (see [20]) the individual entities are bundled into clusters, depending on the indidivual structure of the social network. These clusters are hosted by a processor core each that also administrates one FEL per cluster.

While the parallelization overhead resulting from the step from one to two involved processors significantly increases the runtime, Permalla reports a speedup of 3.43 using 16 processor cores on a single machine.

**Simulation of Transit Networks.** Ullrich et al. (see [24], [27], and [26]) utilize synchronized execution to parallelize transit simulation models. Their aim is to support the decision-making of operator personnel in the case of major network disturbances. Often the operators only have a short time window at their disposal, as decisions have to be taken in a matter of minutes or even seconds. To be effective, a simulation application enabling the online examination of the impact of potential counter-measures has to run fast, enabling the quick rejection of strategies unsuitable for specific situations. As the traffic operator's desktop computers that also run third-party user processes are the target platform of the resulting simulation tool, the method is specifically aimed at utilizing their small scale parallel processing capacity while being able to quickly shift load to idle processor cores in case external user processes claim resources. To address these issues, the method applies a dynamic and adaptive load balancing scheme analogous to the one described in Section 3.

Ullrich et al. report a speedup of 2.83 for four parallel processor cores with a common cache. Connecting machines over the network with its longer message delays reduces the speedup to 2.25. While the dynamic and adaptive load balancing mechanism only improves run time by a few percent on machines exclusively available to the transit simulation, it has a significantly larger impact when used to compensate for ressources assigned to third-party processes on machines concurrently used by other user processes (see [24]). Experiments with artificial loads demonstrate that effect of load balancing increases with the size of the model.

## 5 Conclusion

This paper presented an overview of basic concepts and methods of the parallel execution of discrete simulation models, with a focus on model-based parallelization. Following a short description of the background of parallel simulation, the two main classes of methods – conservative and optimistic execution – were presented, complemented by a description of typical static and dynamic load balancing mechanisms. Finally, some typical applications of model-based parallelization we introduced.

Model-based parallelization is a comparatively simple, easy to understand, but also very powerful approach. It is especially useful to accelerate the execution of large models that have to yield results fast. A wide array of applications has been reported on during the last two decades, including communications and electronics, disaster mitigation, health care, logistics, supply management, and transportation.

For further, more detailed study a number of sources authored by Richard Fujimoto, the unrivaled chronicler of the field, can be recommended: His introductionary book "Parallel and Distributed Simulation Systems" (see [6]) covers most concepts described in this article in great detail; it has aged exceedingly well. More recent developments are shared in his tutorial papers for the Winter Simulation Conference (see [7]). For students of the development of parallel (and distributed) simulation since the 1970s the historical overview "Parallel Discrete Event Simulation: the Making of a Field" by Fujimoto et al. (see [8]) is warmly commended.

## References

[1] Banks, J., Carson, J. S., Nelson B. L., Nicol D. M.: *Discrete-Event System Simulation*. Pearson, 2010.

[2] Bryant, R. E.: *Simulation of Packet Communication Architecture Computer Systems*. Computer Science Laboratory. Technical Report, Cambridge, Massachusetts, Massachusetts Institute of Technology, 1977.

[3] Chandy, K. M., Misra, J.: Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. In: *IEEE Transactions on Software Engineering*, SE-5(5), 1965, pp. 250-255.

[4] Dickens, P., Reynolds, P.: *SRADS with local rollback*. Institute for Parallel Computation, School of Engineering and Applied Science, University of Virginia, 1990.

[5] Fujimoto, R.: Lookahead in Parallel Discrete Event Simulation. In: Proc. *1988 International Conference on Parallel Processing*, 1988.

[6] Fujimoto, R.: *Parallel and Distributed Simulation Systems*. John Wiley & Son, 2000.

[7] Fujimoto, R.: Parallel and Distributed Simulation. In: Proc. *2015 Winter Simulation Conference*, 2015, pp. 45-59.

[8] Fujimoto, R., Bagrodia, R., Bryant, R. E., Chandy, K. M., Jefferson, D., Misra, J., Nicol, D., Unger, B.: Parallel Discrete Event Simulation: the Making of a Field. In: Proc. *2017 Winter Simulation Conference*, 2017, pp. 262-291.

[9] Fujimoto, R., McLean, T., Perumalla, K., Tacic, I.: Design of high performance rti software. In: Proc. *Fourth IEEE International Workshop on Distributed Simulation and Real-Time Applications*, 2000, pp. 89-96.

[10] Garey, M., Johnson, D., Stockmeyer, L.: Some simplied NP-complete graph problems. In: *Theoretical Computer Science*, 1976, pp. 237-267.

[11] Jefferson, D. R.: Virtual Time. In: *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, 1985, pp. 404-425.

[12] Kernighan, B. W., Lin, S.: An Efficient Heuristic Procedure for Partitioning Graphs.*Bell Syst. Tech Journal*, Vol. 49, No. 2, 1970, pp. 291-307.

[13] Lin, Y. B., Lazowska, E. D.: Exploiting lookahead in parallel simulation. In: *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 4, 1990, pp. 457-469.

[14] Lubachevsky, B., Schwartz, A., Weiss, A.: Rollback Sometimes Works ... If Filtered. In: Proc. 1989 Winter Simulation Conference, 1989, pp. 630-639.

[15] Mattern, F.: Efficient algorithms for distributed snapshots and global virtual time approximation. In: *Journal of Parallel and Distributed Computing*, Vol. 18, No. 4, 1993.

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

[17] Permalla, K. S.: Concurrent Conversation Modeling and Parallel Simulation of the Naming Game in Social Networks. In: Proc. *2017 Winter Simulation Conference*, 2017, pp. 1037-1048.

[18] Russell, S. J., Norvig, P.: *Artificial Intelligence: A Modern Approach (2nd ed.)*, Upper Saddle River, New Jersey: Prentice Hall, 2003, pp. 111-114.

[19] Sargent, R. G.: Verification and validation of simulation models. In: Proc. *2010 Winter Simulation Conference*, 2010, pp. 166-183.

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

[21] Schlagenhaft, R., Ruhwandel, M., Sporrer, C., Bauer, H.: Dynamic Load Balancing of a Multi-Cluster Simulator on a Network of Workstations. In: Proc. *PADS95*, 1995, pp. 175-180.

[22] Sokol, L. M., Stucky, B. K.: Experimental results for a constrained optimistic scheduling paradigm. In: *Distributed Simulation*, Vol. 22, 1990, pp. 169-173.

[23] Steinman, J. S.: Breathing time warp. In: Proc. *Seventh Workshop on Parallel and Distributed Simulation*, 1993, pp. 109-118.

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

[25] Ullrich, O., Lückerath, D.: An Introduction to Discrete-Event Modeling and Simulation. In: *Simulation Notes Europe (SNE)*, Vol. 27, No. 1, 2017, pp. 9-16.

[26] Ullrich, O., Lückerath, D., Franz, S., Speckenmeyer, E.: Simulation and optimization of Cologne's tram schedule. In: *Simulation Notes Europe (SNE)*, Vol. 22, No. 2, August 2012, pp. 69-76.

[27] Ullrich, O., Lückerath, D., Speckenmeyer, E.: Model-based parallelization of discrete traffic simulation models. In: *Simulation Notes Europe (SNE)*, Vol. 24, No. 3-4, 2014, pp. 115-122.