8. Parallel Discrete Event Simulation

Georg Kunz (RWTH Aachen University)

8.1 Introduction

Ever since discrete event simulation has been adopted by a large research community, simulation developers have attempted to draw benefits from executing a simulation on multiple processing units in parallel. Hence, a wide range of research has been conducted on Parallel Discrete Event Simulation (PDES). In this chapter we give an overview of the challenges and approaches of *parallel simulation*. Furthermore, we present a survey of the parallelization capabilities of the network simulators OMNeT++, ns-2, DSIM and JiST.

8.1.1 Why do we need Parallel Discrete Event Simulation?

Communication systems are becoming increasingly complex – and so do the corresponding evaluation tools. In general, two orthogonal trends in terms of complexity can be identified: an increase in structural complexity on the one hand and in computational complexity on the other. Both impose high demands on the simulation architecture and the hardware executing the simulations.

We denote the size of a simulated network as an indicator of the structural complexity of a simulation model. Recent developments in the Internet, in particular fast growing systems like peer-to-peer networks, caused an enormous increase in the size of communication systems. Such large systems typically posses complex behavioral characteristics which cannot be observed in networks of smaller size (e.g., testbeds) or captured by analytical models. Thus, in order to study those characteristics, simulation models comprise huge numbers of simulated network nodes. Since every network node is represented in memory and triggers events in the simulation model, memory consumption and computation time increase significantly.

Even if the investigated network is relatively small, computational complexity becomes an important factor if the simulation model is highly detailed and involves extensive calculations. In particular wireless networks which make use of advanced radio technologies such as OFDM(A) [258] and Turbo Codes [58] fall in this category. Sophisticated radio propagation models, interference modeling, and signal coding models further escalate the overall complexity.

Simulation frameworks aim to compensate these issues by enabling simulations to be executed in parallel on multiple processing units. By combining memory and computation resources of multiple processing units, simulation time can be restricted to a reasonable amount while at the same time extremely high memory requirements can be met. Although this approach is known for more than two decades [161, 163, 365], recent technological advances greatly reduce prices for parallel computing hardware – thus making such hardware available to a large research community and moving back into the focus of simulation developers.

8.1.2 Challenges of Parallel Discrete Event Simulation

The approach taken by PDES is to divide a simulation model in multiple parts which execute on independent processing units in parallel. The central challenge of PDES is thereby to maintain the correctness of the simulation results as we will see in the following.

We first briefly recapitulate the concept of discrete event simulation. Any discrete event simulation, i.e., the simulation framework and a particular simulation model, exhibits three central data structures: i) state variables of the simulation model, ii) a timestamped list of events, and iii) a global clock. During a simulation run, the scheduler continuously removes the event with the smallest timestamp

$$e_{min} = \min\{T(e)|\forall e \in E\}$$

from the event list and executes the associated handler function. T denotes the timestamp function which assigns a time value to each event and E is the set of all events in the event list. While the handler function is running, events may be added to or removed from the event list. Choosing e_{min} is crucial as otherwise the handler function of an event e_x with $T(e_{min}) < T(e_x)$ could change state variables which are later accessed when e_{min} is handled. In this case the future (e_x) would have changed the past (e_{min}) which we call a *causal violation*.

By complying with this execution model, a sequential network simulator prevents causal violations. However, this model cannot easily be extended to support parallel execution since causal violations may occur frequently. The following example presents a naive approach and illustrates its flaws.

Assume that n processing units, e.g., CPUs, can be utilized by a parallel simulation framework. In order to keep all available CPUs busy, the central scheduler continuously removes as many events in timestamp order from the event queue as there are idle CPUs. Hence, at any time, n events are being processed concurrently. Now consider two events e_1 and e_2 with $T(e_1) < T(e_2)$ that have been assigned to different CPUs in timestamp order. The processing of e_1 creates a new event e_3 with $T(e_1) < T(e_3)$ and $T(e_3) < T(e_2)$.

Since e_2 has already been scheduled and may have changed variables that e_3 depends on, a causal violation has occurred.

Thus, we formulate the central challenge of PDES as follows:

Given two events e_1 and e_2, decide if both events do not interfere, hence allowing a concurrent execution, or not, hence requiring a sequential execution.

Parallel simulation frameworks employ a wide variety of synchronization algorithms to decide this question. The next section presents a selection of fundamental algorithms and discusses their properties.

8.2 Parallel Simulation Architecture

In this section, we introduce the general architecture of parallel discrete event simulation. This architecture forms the substrate for algorithms and methodologies to achieve high simulation performance while maintaining correctness of the simulation results.

A parallel simulation model is composed of a finite number of partitions which are created in accordance to a specific partitioning scheme. Three exemplary partitioning schemes are i) space parallel partitioning scheme, ii) channel parallel partitioning, and iii) time parallel partitioning.

The *space parallel partitioning* scheme divides the simulation model along the connections between simulated nodes. Hence, the resulting partitions constitute clusters of nodes. The *channel parallel partitioning* scheme bases on the assumption that transmissions that utilize different (radio) channels, mediums, codings etc. do not interfere. Thus, events on non-interfering nodes are considered independent. As a result, the simulation model is decomposed in groups of non-interfering nodes. However, channel parallel partitioning is not generally applicable to every simulation model, thus leaving it for specialized simulation scenarios [288]. Finally, *time parallel partitioning* schemes [290] subdivide the simulation time of a simulation run in time-intervals of equal size. The simulation of each interval is considered independent from the others under the premise that the state of the simulation model is known at the beginning of each interval. However, the state of a network simulation usually comprises a significant complexity and is not known in advance. Thus, this partitioning scheme is also not applicable to network simulation in general. Consequently, the remainder of this chapter focuses on space parallel partitioning.

A *Logical Process (LP)* constitutes the run-time component which handles the simulation of partitions. In this context, each partition is typically mapped to exactly one LP. Furthermore, every LP resembles a normal sequential simulation as each LP maintains state variables, a timestamped list of events and a local clock. Additionally, inter-LP communication is conducted by sending timestamped messages via *FIFO channels* which preserve

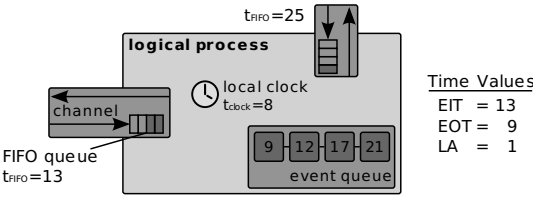

Fig. 8.1: A logical process (LP) maintains a local clock, an event queue and is connected to other LPs via FIFO channels.

a local FIFO characteristic. This means that all messages arrive at the receiving LP in exactly the same order as they were send in. Based on the notion of LPs, the *local causality constraint* defines an execution model for LPs that prevents causal violation:

Local Causality Constraint. A discrete-event simulation, consisting of logical processes that interact exclusively by exchanging time stamped messages obeys the local causality constraint if and only if each LP processes events in non-decreasing time stamp order.

In practice, the number or LPs (i.e., partitions) is equal to the number of CPUs provided by the simulation hardware. Consequently LPs directly map to physical processes. Furthermore, the timestamped and message-based communication scheme constitutes two important properties. First, they allow a transparent execution of LPs either locally on a multi-CPU computer or distributed on a cluster of independent computers. Second, and more importantly, timestamps provide the fundamental information used by synchronization algorithms to decide which events to execute and to detect causal violations. We now present two classes of synchronization algorithms: conservative and optimistic algorithms. While conservative algorithms aim to strictly avoid any causal violation at time of the simulation run, optimistic algorithms allow causal violations to occur, but provide means for recovering.

8.2.1 Conservative Synchronization Algorithms

Conservative synchronization algorithms strive to strictly avoid causal violations during a simulation run. Hence, their central task is to determine the set of events which are *safe* for execution. In order to decide on this question, conservative algorithms rely on a set of simulation properties [43].

The *Lookahead* of a LP is the difference between the current simulation time and the timestamp of the earliest event it will cause at any other LP. The *Earliest Input Time (EIT)* denotes the smallest timestamp of all messages that will arrive at a given LP via any channel in the future. Accordingly, the

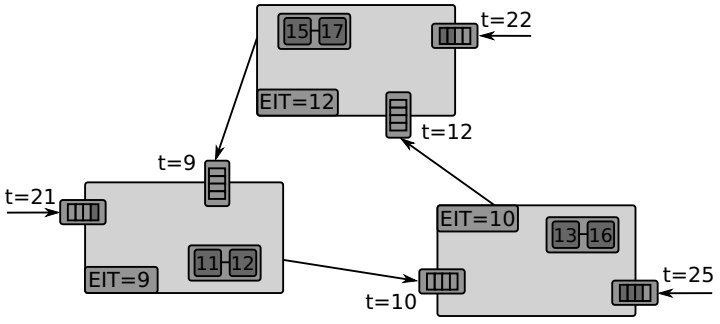

Fig. 8.2: Three LPs are deadlocked in a circular dependency: Every LP waits for its neighbor to send a message in order to increase the EIT.

Earliest Output Time (EOT) denotes the smallest timestamp of all messages that a given LP will send in the future to any other LP.

Based on these definitions, a LP can safely execute all events which have a smaller timestamp than its current EIT since it is guaranteed that no messages with a smaller timestamp will arrive later. Figure 8.1 shows an exemplary LP and the corresponding time values.

Null-Message Algorithm

The simple approach presented above does not solve the synchronization problem entirely as it can cause the simulation to deadlock. Figure 8.2 illustrates this behavior. The LPs can neither execute an event from their local event queue nor any incoming message since the local EIT is too small. Hence, each LP waits for a message from its direct neighbor in order to increase the EIT thereby creating a circular dependency.

This problem is addressed by the *Null-Message Algorithm (NMA)* which was first introduced by Misra and Chandra [317]. The algorithm uses null-messages, i.e., messages which do not contain simulation model related information, to continuously increase the EIT of all neighboring LPs. For this purpose, null-messages carry the LP's current EOT timestamp, which is determined by adding the lookahead to its current local time. Hence, null-messages can be considered as a promise of an LP not to send any message with a smaller timestamp than EOT in the future. Upon receiving a null-message, each LP updates its EIT to a potentially greater value. If the updated EIT has advanced beyond events in the event queue, those are now considered safe for execution. This algorithm guarantees to prevent deadlocks if the simulation model does not contain zero-lookahead cycles.

Performance Considerations

The performance of conservative synchronization algorithms is highly influenced by the size of the lookahead. If the lookahead is small, a potentially excessive number of null-messages is exchanged by the simulation without actually making progress. This system behavior is often called the "time-creeping" problem: Assume two LPs that are blocked at a simulation time of 100 seconds and whose next (simulation model) events are scheduled at a simulation time of 200s. Due to a small lookahead of only 1 second, 100 null-messages have to be transmitted in order to reach the next event.

However, the actual size of the lookahead is an inherent property of the simulation model – not of the synchronization algorithm. In network simulations the lookahead is usually determined by the link delay between the nodes. This works well for simulation models of fixed and wired networks such as the Internet by considering long distance backbone links. Unfortunately, link delays are extremely small in wireless networks, thus decreasing the lookahead significantly. As a result, extensive research work has been conducted on the development of techniques to extract the maximum lookahead from a simulation model [291, 315]. The general idea of these approaches is to exploit standardized protocol-specific properties such as timeouts or waiting periods (e.g., SIFS and DIFS in IEEE 802.11) to increase the available lookahead.

In consideration of these facts, it is important to determine the potential of a simulation model for achieving a satisfactory parallel performance. Given a specific simulation model and particular simulation hardware, Inequality (8.1) allows for roughly answering this question for the Null-Message Algorithm:

$$E_{seq} \geq \frac{n\tau P\lambda}{L} \tag{8.1}$$

If (8.1) holds, then the model is expected to perform well under NMA. E_{seq} denotes the event density of the simulation model under sequential execution, i.e., the number of events per simulated second, and n is the number of LPs under parallel execution. Furthermore, τ measures the messaging latency of the physical simulation hardware, while P characterizes its computing power in processed events per second. λ represents the coupling factor of the LPs, describing how fluctuations of E and P effect blocking of LPs, and finally, L denotes the lookahead. An in-depth discussion of these approximations can be found in [476].

Ideal-Simulation-Protocol

Researchers have proposed a wide range of synchronization algorithms providing different properties and characteristics. The *Ideal Simulation Protocol*

(ISP) [43] is a means for determining the overhead of any conservative synchronization algorithm, thereby allowing an objective performance comparison.

ISP bases on the observation that each synchronization algorithm imposes two types of overhead: messaging overhead which is caused by sending (simulation model) messages to LPs on remote machines and the actual synchronization overhead which is caused by blocking and additional (synchronization) messages that are only used by the algorithm (e.g., null-messages). While messaging overhead is a property of the simulation model, the synchronization overhead is a property of a particular algorithm. The idea of ISP is to eliminate the synchronization overhead while preserving the messaging overhead, hence achieving an optimal simulation performance which acts as a baseline for any other synchronization algorithm.

ISP is a two-phase synchronization protocol: in the first phase, ISP employs an arbitrary conservative algorithm for the parallel simulation of a simulation model of choice. During this phase it collects meta information about all messages and events that occurred in the simulation run and writes those to a trace file. In the second phase the simulation is re-run with respect to the information from the trace file. By utilizing this information, an optimal synchronization is achieved without the need for inter-LP synchronization. Thus the efficiency of an algorithm S with respect to a specific simulation architecture A is given by

$$\text{Efficiency}(S, A) = \frac{\text{Execution time using ISP on } A}{\text{Execution time using } S \text{ on } A}$$

8.2.2 Optimistic Synchronization Algorithms

In contrast to conservative algorithms, *optimistic synchronization algorithms* allow LPs to simply execute all events (in time stamp order) as they come in, but without ensuring that causal violations will not occur. This probably counter-intuitive behavior is motivated by the observation that conservative algorithms sometimes block LPs unnecessarily: Often not enough information is available to mark a certain event safe, although it actually is. Hence, the simulation performance is reduced significantly.

Thus, optimistic algorithms assume that an event will not cause a causal violation. This approach has two primary advantages: First, it allows exploiting a higher degree of parallelism of a simulation model. If a simulation model contains two largely independent partitions, which interact only seldom, only infrequent synchronization is actually needed. Second, the overall performance of the parallel simulation depends less on the lookahead. Thus making it attractive to models with small lookahead such as in wireless networks.

Clearly, the downside is that a causal violation leaves the simulation in an incorrect state. As a result, optimistic algorithms provide recovery

mechanisms: during a simulation run, the PDES engine continuously stores the simulation state. Upon a causal violation, the simulation is rolled-back to the last state known to be correct.

Time-Warp Algorithm

The *Time-Warp Algorithm* [241] is a well-known optimistic algorithm. A causal violation is detected when an LP lp_i receives a message m with a smaller time stamp than its own local clock: $T(m) < T(LP)$. As a result, lp_i restores the latest checkpoint known to be valid. Since the incorrect simulation state not only affects lp_i, but also any other LP that recently received a message from lp_i, the subsequent roll-back must also include those LPs. Hence, lp_i initiates the roll-back by sending an *anti-message* to all neighboring LPs for every message sent since the restored checkpoint. If an anti-message is enqueued in a channel queue together with the corresponding original message, i.e., this message was not yet processed by the receiving LP, both annihilate each other. If, however, the original message was processed, the receiving LP also initiates a roll-back. By using this process, the algorithm recursively drives all incorrect messages and states out of the system.

Performance Considerations

A major drawback of this class of algorithm is the significant amount of hardware resources needed for storing the simulation state checkpoints. Additionally, I/O operations pose a special problem as they cannot be rolled-back in general.

One approach to these challenges bases on the notion of *Global Virtual Time (GVT)* which is given by the smallest time stamp of all unprocessed messages in the simulation. Since GVT denotes the oldest message in the system, it is guaranteed not to be rolled-back. A garbage collector subsequently reclaims all resources occupied by older events and commits pending I/O operations.

Another performance issue of optimistic algorithms is thrashing: a significant decrease in simulation performance caused by frequent roll-backs. This behavior is often triggered by fast LPs that rush ahead and induce roll-backs in the majority of slower LPs. To counteract this behavior, a time window W is introduced which limits the amount of optimism. LPs may then only execute events within $GVT+W$, thus restricting fast LPs from rushing ahead.

8.3 Parallelization in Practice

In this section we analyze a selection of contemporary network simulators in terms of their PDES capabilities.

8.3.1 OMNeT++

OMNeT++ natively supports PDES by implementing the conservative Null-Message Algorithm and the Ideal-Simulation-Protocol [108]. By utilizing different communication libraries, a parallel simulation can furthermore make use of multi-CPU machines as well as a distributed setup.

In order to distribute a simulation model to a set of LPs, OMNeT++ employs a placeholder approach: A simple placeholder module is automatically created for each module which is assigned to a remote LP. When a message arrives at a placeholder module, it transparently marshals the message and sends it to the real module in the particular LP, which unmarshals and processes it.

Due to this (almost) transparent integration of PDES into the simulator architecture, OMNeT++ imposes only very few restrictions on the design of a simulation model for parallel execution. According to the definition of the LP-based architecture of PDES (c.f. Section 8.2), parallel OMNeT++ models must not use global variables or direct method calls to other modules. However, since the lookahead is given by the link delays, currently only static topologies are supported, making it difficult to parallelize mobile scenarios.

The properties of a parallel simulation setup are entirely specified in the global `omnet.ini` configuration file. This file defines the partitioning, which assigns every module and compound to a specific LP, as well as the synchronization and communication algorithms to use.

8.3.2 ns-2

PDNS [394] constitutes the parallel simulation architecture of ns-2. It is based on a conservative synchronization algorithm, which coordinates distributed instances of ns-2 which execute the partitions of the parallel simulation model. In PDNS terminology these instances are called federates.

PDNS was designed to integrate seamlessly into ns-2 without effecting existing simulation models. However, parallelization is not transparent to the simulation models. Instead, the parallelization process requires modifications of the simulation models in order to make use of the functionality provided by PDNS. In particular, links between nodes in different federates have to be replaced by dedicated “remote links", which implement PDES functionality. For the actual inter-federate communication, PDNS itself builds upon two

communication libraries (libSynk and RTIKIT), which provide a variety of communication substrates such as TCP/IP, Myrinet and shared memory for testing and debugging.

8.3.3 DSIM

DSIM [96] is a parallel simulator which relies on optimistic synchronization. It is designed to run on large-scale simulation clusters comprised of hundreds to thousands of independent CPUs.

The core of DSIM is formed by a modified optimistic time warp synchronization algorithm carefully designed for scalability. For instance, the obligatory calculation of the GVT does not require message acknowledgments and utilizes short messages of constant length in order to reduce latency and messaging overhead. Furthermore, resource management is performed by a local fossil collection algorithm (garbage collection), which intends to increase locality by immediately reusing freed memory regions.

A performance evaluation using the synthetic PHOLD benchmark [162] indicates that DSIM achieves a linear speedup with an increasing number of CPUs. However, no evaluations using detailed network simulations are known to the author.

8.3.4 JiST

JiST [52, 51] is a general purpose discrete-event simulation engine that employs a virtual machine-based simulation approach. Implemented in Java, it executes discrete event simulations by embedding simulation time semantics directly into the execution model of the Java virtual machine. Simulation models are programmed using standard object-oriented language constructs. The JiST system then transparently encapsulates each object within a JiST entity, which forms the basic building block of a simulation model. While program execution within an entity follows the normal Java semantics, method invocations across entities act as synchronization points and are hence handled by the JiST run-time system. Invocations are queued in simulation time order and delivered to the entity when its internal clock has progressed to the correct point in time.

By utilizing this approach, entities can independently progress though simulation time between interactions - thus natively allowing a concurrent execution. Furthermore, an optimistic execution model can be supported via check-pointing.

8.3.5 IKR SimLib

The IKR SimLib [216] (see Chapter 4) is a simulation library that uses the common *batch means* method. This method seeks to obtain sufficient exactness and small confidence intervals [278]. For this purpose, a simulation run is divided over time into batches that are statistically independent. The results of each batch are interpreted as samples and are used to compute mean values and confidence intervals. The IKR SimLib provides support to execute these independent batches on multiple CPUs in parallel. Moreover, as soon as all batches are finished, the library supports aggregation of the statistical data calculated in each batch. Consequently, the parallel execution of batches efficiently reduces the time in comparison of sequential batch execution.

Thanks to the assumption of independent batches and their parallel execution, the IKR SimLib does not need to employ synchronization algorithms. This avoids the overhead typically imposed by synchronization and increases the simulation speed. Furthermore, it allows good scalability in terms of processing units. However, this approach is less suited for running extremely complex simulations, which impose huge demands on the hardware as within one batch the complete model needs to be simulated on one machine. As a result, very large simulations may not fit in the memory of a single machine or may take longer in comparison to other parallelization approaches.

8.4 Conclusion

Parallel discrete event simulation has been the field of intensive research in the last two to three decades. As a result, a plethora of different algorithms, frameworks and approaches exist today. We could hence sketch selected approaches only briefly in this chapter.

We first introduced the primary challenges that PDES faces: given a simulation model, a partitioning of the model needs to be found and distributed among the parallel simulation hardware. Next, an efficient synchronization algorithm is needed to maintain the correctness of the distributed simulation and to avoid causal violations. We presented two major classes of synchronization algorithms: conservative and optimistic synchronization. Along their discussion, performance considerations were presented. Finally, a selection of contemporary parallel simulation frameworks was introduced.

Although a wide range of simulation frameworks natively support PDES today, PDES still hasn't achieved a final breakthrough in network simulation due to a lack of performance and challenging programming models. Instead, often multiple sequential simulations are run on multi-core hardware in parallel. Hence, PDES remains a hard problem today. However, due to the ongoing development in the hardware sector, which favors an increasing number of processing units over an increasing speed of a single unit, PDES will remain an important and active field of research.