Karlsruhe Institute of Technology

Parallel Simulation of Queueing Petri Net Models

Diploma Thesis of

Jürgen Christian Walter

At the Department of Informatics
Institute for Program Structures
and Data Organization (IPD)

Reviewer:	Prof. Dr. Ralf Reussner
Second reviewer:	Prof. Dr. Walter F. Tichy
Advisor:	M.Sc. Simon Spinner
Second advisor:	Dipl. Inf. Jörg Henß

Duration: April 15th, 2013 – October 14th, 2013

Abstract

For years the CPU clock frequency was the key to improve processor performance. Nowadays, modern processors enable performance improvements by increasing the number of cores. However, existing software needs to be adapted to be able to utilize multiple cores. Such an adaptation poses many challenges in the field of discrete-event software simulation. Decades of intensive research have been spent to find a general solution for parallel discrete event simulation. In this context, Queueing Networks (QNs) and Petri Nets (PNs) have been extensively studied. However, to the best of our knowledge, there is only one previous work that considers the concurrent simulation of Queueing Petri Nets (QPNs) [Jür97]. This work focuses on comparing different synchronization algorithms and excludes a majority of lookahead calculation and net decomposition. In this thesis, we build upon and extend this work. For this purpose, we adapted and extended findings from QNs, PNs and parallel simulation in general.

We apply our findings to SimQPN, which is a sequential simulation engine for QPNs. Among other application areas, SimQPN is currently applied to online performance prediction for which a speedup due to parallelization is desirable. We present a parallel SimQPN implementation that employs application level and event level parallelism. A validation ensures the functional correctness of the new parallel implementations. The parallelization of multiple runs enables almost linear speedup. We parallelized the execution of a single run by the use of a conservative barrier-based synchronization algorithm. The speedup for a single run depends on the capability of the model. Hence, a number of experiments on different net characteristics were conducted showing that for certain models a superlinear speedup is possible.

Zusammenfassung

Lange Zeit war die Frequenz des Prozessors ausschlaggebend für die Geschwindigkeit von Prozessoren. Heutzutage erlauben moderne Prozessoren Leistungssteigerungen durch eine Erhöhung der Anzahl von Prozessorkernen. Allerdings muss die bestehende Software angepasst werden, um die Kerne eines Multikernprozessors auch nutzen zu können. Auf dem Gebiet der ereignisgesteuerten Simulation birgt eine derartige Anpassung viele Herausforderungen. Jahrzehnte intensiver Forschung wurden darauf verwandt, eine allgemeine Lösung für parallele ereignisgesteuerte Simulationen zu finden. In diesem Zusammenhang wurden Warteschlagen-Netzwerke und Petri-Netze intensiv untersucht. Jedoch existiert, nach unserer Kenntnis, nur eine einzige Arbeit zur Parallelisierung der Simulation von Warteschlangen-Petri-Netzen [Jür97]. Diese Arbeit konzentriert sich auf den Vergleich von Synchronisationsalgorithmen und schließt die Betrachtung vorausschauender Berechnungen und Netzwerkzerlegung weitgehend aus. Die vorliegende Diplomarbeit baut auf dieser Arbeit auf und erweitert sie. Hierfür haben wir bestehende Erkenntnisse von Warteschlangen-Netzwerken, Petri-Netzen und paralleler Simulation im Allgemeinen für Petri-Warteschlangen-Netzwerke angepasst und erweitert.

Wir wenden unsere Erkenntnisse auf den sequentiellen Warteschlangen-Petri-Netz Simulator SimQPN an. Neben anderen Anwendungsgebieten wird SimQPN aktuell dazu verwendet, Geschwindigkeiten von Software in Echtzeit vorherzusagen. Insbesondere für die Echtzeitvorhersage ist eine Beschleunigung durch Parallelisierung wünschenswert. Wir präsentieren eine neue SimQPN Implementierung bei der Parallelisierungen auf Anwendungsebene und auf Ereignisebene zum Einsatz kommen. Eine Validierung stellt die funktionale Korrektheit der parallelen Implementierung sicher. Das Parallelisieren mehrerer Durchläufe ermöglicht eine nahezu lineare Beschleunigung. Die Durchführung eines einfachen Durchlaufs haben wir unter Nutzung eines konservativen Barrieren-basierten Synchronisationsalgorithmus parallelisiert. Die erreichbare Beschleunigung hängt hierbei stark von den Eigenschaften des getesteten Modells ab. Daher wurde eine Vielzahl von Experimenten zu unterschiedlichen Netzwerkcharakteristika durchgeführt, die zeigt, dass für einige Modelle sogar superlineare Beschleunigung möglich ist.

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, October 14^{th}, 2013

. . . .

(Jürgen Christian Walter)

Contents

1. Introduction

For years the CPU clock frequency was the key to improve microprocessor performance. As Pancratius states in [PT08] ”We have reached a major turning point: microprocessor performance can no longer be improved by increasing clock frequencies; instead, higher performance will have to come from parallelism”.

Sequential software can only utilize a single core. In order to utilize multiple cores efficiently, existing software has to be adapted. Like many other discrete-event simulation engines, SimQPN, which is a simulator for Queueing Petri Nets (QPN), is a strictly sequential program and cannot utilize multiple cores so far. However, as multi-core processors become more and more common, it is highly desirable to parallelize the simulation. The adaptation from sequential to parallel QPN simulation will be discussed in this thesis.

In the following, we distinguish between parallel and distributed simulation. Distributed simulation means that subparts of the execution are placed on multiple computers connected by a network. In this context, network communication plays an important role in contrast to parallel simulation. Parallel simulation refers to the execution on one computer with multiple processors and shared memory. The umbrella term for the execution of multiple simulation parts at the same time is concurrent simulation. This thesis focuses on parallel QPN simulation. In the following, we apply the term concurrent simulation when we want to highlight concepts that apply for parallel and distributed simulation.

QPN is a general modeling formalism combining the modeling power and expressiveness of Petri Nets and queueing models. QPNs can be used for quantitative and qualitative analyses in different domains. For instance, QPNs are commonly used for the analysis of the performance of a system as it allows to model hardware and software contention in a combined formalism. The modeling power and convenience of QPNs have been shown in various case studies [KB03, Kou06, KNT07, KSBB08, SKB12]. All these publications state a high accuracy in performance prediction.

In [BBK95], Bause et al. introduced the HiQPN tool for the qualitative and quantitative analysis of QPNs. HiQPN performs the quantitative analysis by transforming it into a Markov chain. This approach can result in a high number of possible states limiting the size of models the can be analyzed. This problem is commonly known in literature as the state space explosion problem[Web09]. The state space explosion problem severly limits the usage of QPNs for system performance analysis as shown in [KD07].

In order to overcome this limitation, Kounev and Buchmann [KB06] proposed to use simulation techniques for the quantitative analysis of QPNs. They developed SimQPN, which is a discrete-event simulation engine optimized for quantitative analysis of QPNs. Discrete-event simulation of SimQPN scales better than Markov analysis [KB07].

The next section motivates research on parallel QPN simulation. At first, we explain the benefits for established and recent application areas. Then we refer to the challenges of parallelization. Some might have in mind that parallel simulation has been something that did not work. Hence, we refer to encouraging publications which justify research in the challenging field of parallel simulation.

1.1 Motivation

In recent years, QPNs have been applied in new contexts in which the time for analyzing a model is severely limited. For example, ongoing research at the Descartes Research Group investigates online performance prediction for a multi-tenant TPC-W application. A performance isolation framework runs the model concurrent to the proactive system reconfiguration to ensure the performance isolation at runtime by using performance predictions form a QPN. A speedup due to parallelism has the potential to improve this research.

Quantitative QPN analysis is a complex task that requires high computational effort and due to that a long simulation time. While the analysis speed is sufficient for offline analysis during software development and maintenance, this is not the case for online analysis.

While a lot of work on parallel and distributed discrete-event simulation exists, the simulation engine SimQPN still runs sequentially. SimQPN has not been parallelized so far, as parallel and distributed simulation is not a trivial task [Fuj00b]. Fujimoto [Fuj89a] described the challenge of parallel simulation by the following words: "From an academic point of view, parallel simulation is interesting because it represents a problem domain that contains substantial amounts of parallelism, yet paradoxically, is one of the most difficult to parallelize on existing machines". Misuse of parallelization approaches to unsuitable models can cause parallel simulation to be slower than sequential simulation. It depends on the chosen modeling formalism and the characteristics of the model which parallelization approach performs best. For some model characteristics no speedup through parallelization could be achieved so far.

In contrast to PN and QN, concurrent QPN simulation has been paid less attention so far. Only Jürgens [Jür97] considered the distributed simulation of QPNs. His master thesis focused on comparing different synchronization algorithms for distributed simulation. He concluded that the encouraging speedups on artificial models make further research promising. However, he excluded major parts of the QPN formalism, such as queueing strategies, which are relevant for concurrent simulation in performance modeling and other scenarios. A discussion on queueing strategies may help to utilize the full potential of the QPN formalism and increase the number of models that may benefit from concurrent simulation.

Furthermore, we can refer to recent advances in implementation techniques for parallel simulation. Ball and Bull [BB03] investigated the performance of synchronization-barriers in Java. The proposed active wait and multi-level barriers provide the basis for high performance barrier implementations like the JBarrier framework developed at the University of Bonn. By the use of this barrier framework, Pelschow [PVM09] reached a superlinear speedup in distributed simulation.

Our optimism for obtaining a speedup is based on local instead of distributed simulation and on technical advances. Nevertheless, parallel simulation remains a challenging task where success is not guaranteed. The risk is justified though a successful parallelization would improve established and recent fields of research. New application areas and the above-cited encouraging publications underscore the need for further research.

A performance-improved SimQPN would increase convenience for established application areas and possibly unlock new application areas. A faster and more scalable SimQPN version can advance the usage of QPN models for online performance and resource management. One way to improve performance and to make applications more scalable is to parallelize their execution. A successful parallelization has the potential to speed up simulation. In summary, the following benefits can be expected of a parallelized SimQPN version:

Faster Analysis The parallelized version of SimQPN has the potential to perform faster QPN analysis than the sequential version.

Better use of multi-cores The parallel SimQPN version utilizes common hardware better than the existing sequential SimQPN.

Better Scalability By the utilization of multiprocessors the possibility to analyze larger QPN models becomes available. In best-case scenario, the size of models scales with the number of processors.

Extended application areas The improved performance and scalability enables more complex models to be processed in reasonable time. This enables to research more complex software systems. Further, this makes QPNs more applicable for online performance prediction. An improved analysis tool makes QPNs attractive to a wider audience.

1.2 Aim of the Thesis

This diploma thesis focuses on exploring the potential of parallelizing the simulation of QPNs in the context of SimQPN. The aim of the thesis is to evaluate the possibility of speeding up the simulation of QPNs by means of parallel simulation.

This thesis consists of a conceptual and a practical part. The conceptual part investigates existing approaches for parallelization and checks their applicability to parallel QPN simulation.The second goal is to implement parallel versions of existing analysis methods within SimQPN and evaluate their potential for speedup.

In summary, the sub-goals of this thesis will be:

1. Design and apply a new software architecture for SimQPN which supports parallel simulation.
2. Identify existing techniques for parallel event-discrete simulation and assess their applicability to the simulation of QPNs. This includes an analysis of strengths and weaknesses of the QPN formalism and models for performance prediction in relation to parallel simulation.
3. Select a subset of the identified techniques and implement them in the existing SimQPN simulator.
4. Validate the correctness of the implemented parallel simulation techniques.
5. Evaluate the implemented parallel simulation techniques regarding their potential for improving simulation times.

1.3 Outline

The remaining chapters of this thesis are organized as follows:

Chapter 2 sets the foundations necessary to comprehend the essence of this thesis. It starts by a description of the QPN formalism based on its subformalisms in Section 2.1. Next, Section 2.2 characterizes simulation concepts and fundamental parallel simulation concepts. Section 2.3 is about the QPME tool and its SimQPN simulation engine.

Chapter 3 describes our approach for this thesis. This chapter refines the goals of the thesis and explains how we proceeded to reach them.

In Chapter 4 we discuss related work. Thereby, we differ between general publications on parallel simulation in Section 4.1, publications on parallel PN simulation in Section 4.2 and publications on parallel QN simulation in Section 4.3.

Chapter 5 describes how to design parallel QPN simulation and provides an overview on promising techniques. The chapter subdivides in the parallelization options at application level Section 5.1, event level Section 5.2 and functional level Section 5.3.

Chapter 6 provides technical details of our implementation. First, Section 6.1 describes the adaptation of the sequential software. Then, like the design chapter, we subdivide in application level Section 6.3, event level Section 6.4 and functional level Section 6.5. The implementation is validated in Chapter 7 and evaluated for performance improvements in Chapter 8.

Finally, Chapter 9 summarizes results, names our contributions Section 9.1 and outlines further research Section 9.2

2. Foundations

In this chapter, we lay foundations required to more easily follow the subsequent chapters. First we construct the Queueing Petri Net (QPN) modeling formalism in Section 2.1. Then we explain simulation concepts and fundamental parallel simulation concepts in Section 2.2. Finally, a short overview about the Queueing Petri Net Modeling Environment (QPME) tool and its simulation engine SimQPN is provided in Section 2.3.

2.1 Modeling Formalisms

The Queueing Petri Net (QPN) formalism is a combination of PNs and QNs. To understand the QPN formalism a basic understanding of its constitutive formalisms is required. We provide a short introduction to QNs in Section 2.1.1 and explain PN and some extensions in Section 2.1.2.

Then we construct QPNs by the inclusion of queues into the places of PNs in Section 2.1.3.

2.1.1 Queueing Networks (QN)

Queueing Networks (QNs) represent a system by a set of interconnected queues. Each queue consists of a waiting area and servers that process tasks from the waiting area. Figure 2.1 depicts an exemplary queue:

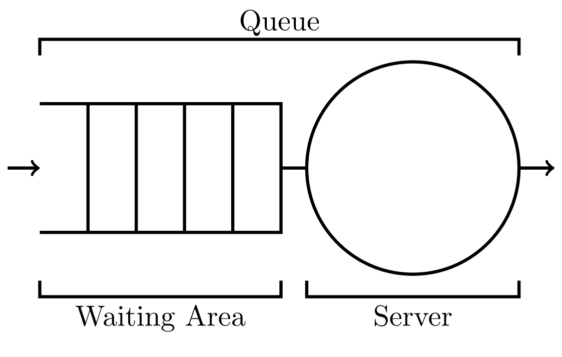

Figure 2.1: Queue Example

Queues represent service stations that process incoming requests. The scheduling strategy of a queue determines the order in which requests are processed. Examples for scheduling strategies are:

First-Come-First-Served (FCFS). Requests are processed in incoming order.

Round Robin (RR). Incoming requests get time slices of fixed length according to FCFS. Requests that have not been completely served during their time slice return to the queue and wait for the next time slice. This continues until the request has been completely served.

Processor Sharing (PS). A version of RR with infinitesimal small time slices. All requests are served simultaneously. PS is used to model Central Processing Units (CPUs).

Priority Scheduling (PRIO). Requests are performed according to their priority.

Infinite Server (IS). Delays any number of entities for a period of time. Queues with this scheduling strategy are often called delay resources or delay servers [Kou05].

Random Scheduling (RANDOM). Incoming requests are processed in random order.

In queueing theory, the standard description for a service station is *Kenadall's Notation*. Originally introduced as a three-tuple [Ken53], the common version consists of six parameters *A/S/m/B/K/SD* [Kou05]:

- **A** describes the distribution of inter arrival times
- **S** stands for the distribution of service times
- **m** stands for the number of servers
- **B** limits the number of requests a queue can hold. If not specified, the default is: $B = \infty$.
- **K** determines the maximum number of requests that can arrive in a queue. If not set: $K = \infty$.
- **SD** stands for the scheduling discipline (also scheduling strategy). If not specified: FCFS

We explain the interconnection of service stations using an example. Figure 5.1 shows a network with two service stations representing a dual-core CPU and a disk device. Incoming requests arrive at `service station 1`(CPU). Then the requests are served by one of the servers. Afterwards, the requests move with probability `p1` to `service station 2` (DSIK) or leave the system with probability `p2`. From `service station 2` requests move back to `service station 1`. In general, the way the requests process through the network is determined by routing probabilities.

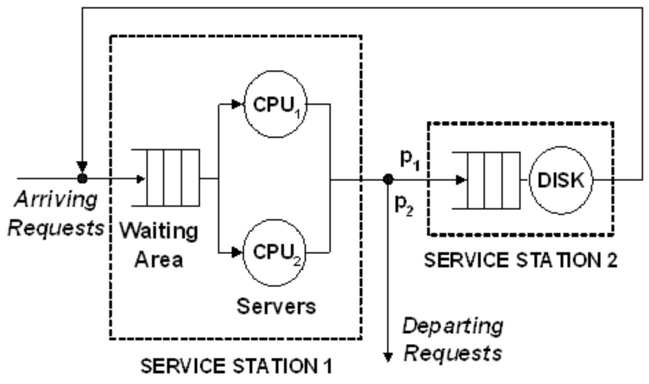

Figure 2.2: Queueing Network Example from [Kou05]

The QN modeling formalism easily enables to model hardware systems as can be seen in Figure 5.1. Overall, QNs are a powerful modeling technique to model hardware contention and scheduling strategies. By contrast, QNs have weaknesses in modeling software contention (contention for processes, threads, database connections and other software resources [KB03]). Software aspects, such as blocking, synchronization and simultaneous resource possession, cannot be modeled in an appropriate way [KB03]. Petri nets are more suitable for modeling software contention aspects.

2.1.2 Petri Nets (PN)

Petri Nets (PNs) (also called Place-Transition Net) are a general formalism that can be used for the analysis of concurrent systems. Carl Adam Petri introduced the formalism in 1962. We start with the introduction of a graphical representation in combination with the formal definition. Then we describe behavioral properties of PNs helpful for analysis. We use the basic definitions for PNs and all PN extensions from [KS11].

In mathematical terms, a PN is a directed bipartite graph with two different types of nodes named *places* and *transitions*. Places represent system states. Places can contain a number of *tokens/marks*. Tokens can stand for resource availability, jobs to perform, flow of control or synchronization conditions [Nie99]. The *initial marking* can be changed by the *firing* of transitions. Transitions are connected to input and output places with forward and backward *incidence functions*. The firing of a transition changes the system state by removing one token from each input place and adding one token to each output place. Firing order, like in concurrent systems, is nondeterministic. Figure 2.3 depicts a PN example. Places are drawn as circles. Tokens are illustrated as dots within the place. Transitions are pictured by rectangles and incidence functions by arrows.

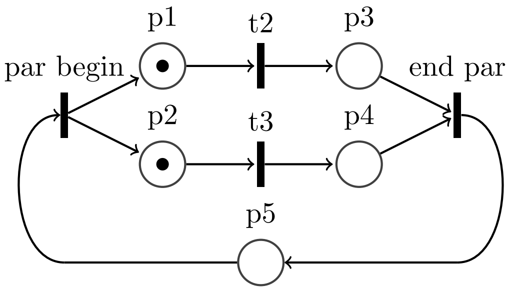

Figure 2.3: A Petri net representing deterministic parallel activities (reprinted from [Mur89])

In comparison to QNs, PNs lend themselves better to model concurrency and synchronization. However, PNs do not support a direct representation of scheduling strategies [Bau93a] and therefore are not suitable to model hardware contention [Kou05]. We define PNs as follows:

Definition 1 (Petri Net).
An ordinary Petri Net (PN) is a 5-tuple $PN = (P, T, I^-, I^+, M_0)$ where:

1. $P = p_1, p_2, \ldots, p_n$ *is a finite and non-empty set of places*
2. $T = t_1, t_2, \ldots, t_m$ *is a finite and non-empty set of transitions,* $P \cap T = \emptyset$
3. $I^-, I^+ : P \times T \rightarrow \mathbb{N}_0$ *are called backward and forward incidence functions, respectively,*
4. $M_0 : P \rightarrow \mathbb{N}_0$ *is called initial marking*

The PN formalism can be used for quantitative and qualitative analysis. Qualitative analysis searches for PN properties. A brief introduction to basic PN properties is listed below. For a more detailed discussion on properties we recommend [Mur89].

Reachability The reachability set $R(C, M_0)$ of a net C is the set of all markings M' reachable from initial marking M_0. If obvious, we abbreviate $R(C, M)$ to $R(M)$.

Boundedness A Petri net is *k-bounded* "if the number of tokens in each place does not exceed a finite number k for any marking reachable from M_0" [Mur89]. If a net is *1-bounded* we call it *save.* Every *k-bounded* net is bounded.

Liveness A petri net is *live* if it has no deadlocks, i.e. for every marking "it is possible to ultimately fire any transition of the net by progressing through some firing sequence" [Mur89].

Reversibility and Home State A Petri net is reversible, if the initial state is reachable from every marking $M \in R(M_0)$. In some scenarios it is not necessary to get back to the initial state M_0, but to some (home) state. A home state is a marking M', which is reachable from every $M \in R(M_0)$.

Coverability A marking M is coverable, if there exists a marking $M' \in R(M_0)$ with $M'(p) \geq M(p)$ for each p in the net.

Persistence A net is persistent, if for any two enabled transitions the firing of one transition will not disable the other transition.

Synchronic Distance The synchronic distance can be seen as the degree of mutual dependency between two events. The distance between two transitions t_1 and t_2 is defined as

$$d_{12} = \max_{\sigma} |\bar{\sigma}(t_1) - \bar{\sigma}(t_2)| \tag{2.1}$$

where σ is a firing sequence starting at some marking $M \in R(M_0)$ and $\bar{\sigma}(t_i)$ is the number of times that transition t_i fires in σ.

Fairness There exist a lot of fairness notions. We list two basic concepts on fairness. Those are *bounded fairness* and *unconditional (global) fairness.* A bounded fair net requires every pair of transitions to be bounded fair. A bounded fair relation between two transitions ensures the maximum number of mutual exclusive firings to be bounded. "A firing sequence σ is called unconditionally (globally) fair if it is finite or every transition in the net appears infinitely often. A Petri net (N, M_0) is said to be an unconditionally fair net if every firing sequence σ from $M \in R(M_0)$ is unconditionally fair". [Mur89]

The original PNs have the advantage of being very simple, expressive and having a proper mathematical foundation. But even for simple scenarios the graphs get very complex. Hence, many extensions to PNs have been proposed.

In some cases it is convenient to distinguish between different resources in one net. For this purpose Jensen [Jen81] introduced Colored Petri Nets (CPNs). In CPNs, token colors represent different resources. Firing rules can be defined according to colors. A color function C assigns different modes to one transition. This means one incidence function per color and transition. By that, C introduces different firing modes per transition and color. The greater modeling convenience comes along with no loss of PN properties as CPNs are a backward-compatible extension to the original PNs. We define CPNs as follows:

Definition 2 (Colored Petri Net).
A Colored Petri Net (CPN) is a 6-tuple $CPN = (P, T, C, I^-, I^+, M_0)$ where:

1. *$P = p_1, p_2, \dots, p_n$ is a finite and non-empty set of places*
2. *$T = t_1, t_2, \dots, t_m$ is a finite and non-empty set of transitions, $P \cap T = \emptyset$*
3. *C is a color function that assigns a finite non-empty set of colors to each place and a finite and non-empty set of modes to each transition.*
4. *I^-, I^+ are the backward and forward incidence functions defined on $P \times T$, such that $I^-(p,t)$, $I^+(p,t) \in [C(t) \to C(p)_{MS}]$, $\forall (p,t) \in P \times T$. The subscript MS denotes multisets. $C(p)_{MS}$ denotes the set of all finite multisets of $C(p)$.*
5. *M_0 is a function defined on P describing the initial such that $M_0(p) \in C(p)_{MS}$.*

Ordinary PNs lack the integration of temporal aspects needed in the field of performance evaluation [Mar90]. Stochastic Petri Nets (SPNs) enable a firing delay to each transition. This delay specifies the time a transition waits until it fires after it has been enabled. Ordinary PNs use immediate transitions whereas SPNs use timed transitions. If timed transitions are enabled at the same time, the next transition to fire is chosen based on *firing weights* (probabilities) assigned to the transitions [KS11].

Generalized Stochastic Petri Nets (GSPNs) enable both types of transitions. Once enabled, immediate transitions fire in zero time [BK02] like in ordinary PNs. Timed transitions enable to use a delay like in SPNs. Immediate (normal, black) transitions should be used to model instant and logical actions. GSPNs help to reduce the number of timed transactions, that are expensive to analyze. Figure 2.4 exemplifies this by a transformation from GSPN to a SPN:

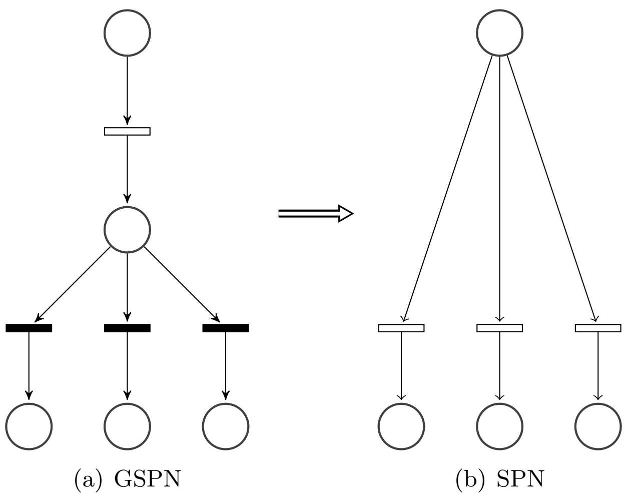

Figure 2.4: Transformation from GSPN to SPN

We define GSPNs as follows:

Definition 3 (Generalized SPN).
A Generalized SPN (GSPN) is a 4-tuple $GSCPN = (PN, T_1, T_2, W)$ where:

1. *$PN = (P, T, I^-, I^+, M_0)$ is the underlying ordinary PN,*
2. *$T_1 \subset T$ is the set of timed transitions, $T_1 \neq \emptyset$,*
3. *$T_2 \subset T$ is the set of immediate transitions, $T_1 \cap T_2 = \emptyset, T_1 \cup T_2 = T$*
4. *$W = (w_1, \ldots, w_{|T|})$ is an array whose entry $w_i \in \mathbb{R}^+$ is a rate of a negative exponential distribution specifying the firing delay , if $t_i \in T_1$ or is a firing weight specifying the relative firing frequency, if $t_i \in T_2$*

The combination of the colors of CPNs and the timed and immediate transitions of GSPN creates Colored Generalized Stochastic Petri Nets (CGSPNs) [BK02]. CGSPNs are a powerful modeling mechanism, but ”they do not provide any means for direct representation of queueing disciplines” [KB07].

2.1.3 Queueing Petri Nets (QPN)

Queueing Petri Nets (QPNs) combine the advantages of QNs and PNs by allowing the integration of queues into the places of a PN. A place with an integrated queue is called a queueing place. Queueing places are drawn with a horizontal line within the circle (see Figure 2.5).

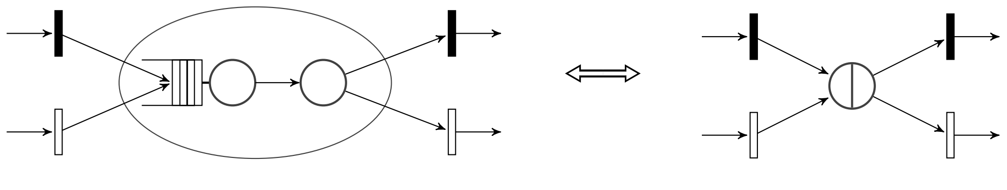

Figure 2.5: Queueing Place Example(reprinted from [KS11]).

QNs suit for quantitative analysis while PNs suit for qualitative analysis [Bau93b]. By the combination of both a powerful modeling formalism arises that suits well for qualitative and quantitative analysis. The QPN definition uses the modeling power and convenience of CGSPNs instead of ordinary PNs. The formal definition of QPN is:

Definition 4 (Queueing Petri Net).
A Queueing Petri Net (QPN) is an eight-tuple $QPN = (P, T, C, I^-, I^+, M_0)$ where:

1. $CPN = (P, T, C, I^-, I^+, M_0)$ *is the underlying colored Petri net*
2. $Q = \bar{Q_1}, \bar{Q_2}, (q_1, \ldots q_{|P|}$ *where:*
   - $\bar{Q_1} \subseteq P$ *is the set of timed queueing places*
   - $\bar{Q_2} \subseteq P$ *is the set of immediate queueing places,* $\bar{Q_1} \cap \bar{Q_2} = \emptyset$
   - q_i *denotes the description of a queue taking all colors of* $C(p_i)$ *into consideration, if* p_i *is a queueing place or equals the keyword "null", if* p_i *is an ordinary place.*
3. $W = (\bar{W_1}, \bar{W_2}, (w_1, \ldots, w_{|T|}))$ *where:*
   - $\bar{W_1} \subseteq T$ *is the set of timed queueing places*
   - $\bar{W_2} \subseteq T$ *is the set of immediate transitions,* $\bar{W_1} \cap \bar{W_2} = \emptyset$, $\bar{W_1} \cup \bar{W_2} = T$ *and*
   - $w_i \in [C(t_i) \to \mathbb{R}^+]$ *such that* $\forall c \in C(t_i) : w_i(c) \in \mathbb{R}^+$ *is interpreted as a rate of a negative exponential distribution specifying the fixing delay due to color c, if* $t_i \in \bar{W_2}$

Bause [BE03] showed that some qualitative characteristics of Petri nets still exist in QPNs. The concepts of ergodicity, boundedness, liveness and home states can be applied to QPNs [KS11]. The preservation of these qualitative properties is an essential precondition for quantitative analysis [Bau93a]. One way to perform quantitative analysis on QPNs is simulation. The next section provides an introduction to simulation.

2.2 Simulation

The previous sections discussed different modeling formalisms. This section will demonstrate how simulation can be used to perform quantitative analysis. Section 2.2.1 will present fundamentals on discrete event simulation. Then, in Section 2.2.2, steady-state analysis is explained. This section then will conclude with a description of parallel simulation concepts in Section 2.2.3.

2.2.1 Discrete-Event Simulation

Dynamic systems change their state over time. If the behavior over time can be described as a finite sequence of events, the simulation can be split into discrete events [Mer11]. Then discrete-event simulation is the method of choice. Otherwise, if the system cannot be split into discrete events, continuous simulation is applied. Physical phenomena with an infinite number of states between two time instants are one example for continuous simulation [Mer11].

The discrete-event simulation approach suits well for software system simulation [Mer11]. In a model view, computer systems are discrete. For example, user interactions or resource requests can be split into discrete events. As time plays a big role in simulation we define, like [Fuj00b], three different terms of time:

Physical time refers to the time in the physical systems

Simulation time is an abstraction used by the simulation to model physical time.

Wall-clock time refers to time during the execution of the simulation program

If we do not specify the term time we refer to simulation time. An instant is a value of simulation time at which the value of at least one attribute of an object can be altered. The time between two successive instants is named interval. The state of an object consists of its attribute values at a particular instant. The previous notions on time and state enable to introduce fundamental simulation concepts [Nan81]:

- An *activity* is the state of an object over an interval.
- An *event* is a change in an object state, occurring at an instant, and initiates an activity precluded prior to that instant.
- An *object activity* is the state of an object between two events describing successive state changes for that object.
- A *process* is the succession of states of an object over a span

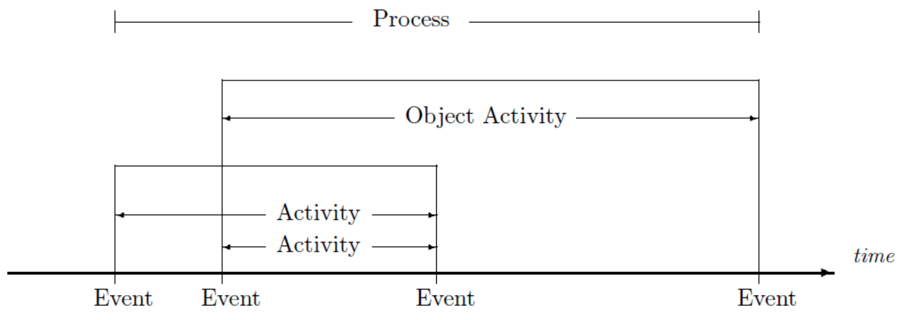

Figure 2.6: Illustration of Event, Activity and Process [Pag95]

Events, activities and processes form the basis of three primary world views:

Event scheduling ”In an event scheduling world view, the modeler identifies when actions are to occur in a model”[Pag95].

Acticvity scanning ”In an activity scanning world view, the modeler identifies why actions are to occur in a model”[Pag95].

Process interaction ”In a process interaction world view, the modeler identifies the components of a mode and describes the sequence of actions of each one” [Pag95].

In discrete-event simulation, world views (sometimes conceptual frameworks) are simulation modeling paradigms. All alternative world views are different perceptions of the same reality.

The choice for a world-view affects performance and scalability of the simulator. Merkle [Mer11] compared the most common event-scheduling and process-interaction in performance prediction scenarios. He concluded event-scheduling to perform and scale much better in the inspected scenarios.

Besides the three primary world views there exist several world views the simulation model can be build on. For additional surveys on world views see [DBN89] and [Peg10].

2.2.2 Steady-State Analysis

The previous section described technical approaches to simulation. This section describes how information can be gained from simulation. Steady-state analysis answers the question of how systems behave in the long run. From the mathematical view, we simulate a discrete-time stochastic process $X_i, i = 1, 2, \ldots$ and are interested in the steady-state behavior. For example, we can determine steady-state mean $\mu = \lim_{i\to\infty} E(X_i)$. In the context of performance prediction we assume the existence of a steady-state. One should keep in mind that steady-state analysis only makes sense if a steady-state exists. For instance, the average of a rising population would not be useful to determine.

2.2.2.1 Determination of Initialization Bias

In general, a simulation run is not steady-state from the beginning. In such cases the model has to be simulated until a steady-state is reached. If the steady-state behavior should be analyzed, the initialization bias $X_0, \ldots, X_{i-1}$ has to be removed. Otherwise initialization bias contaminates the result. We name X_i as truncation point. In literature this problem is also referred to as startup or initial transient problem. Stochastic processes reach their steady-state at different speed and behavior. Figure 2.7 shows the complexity of the problem by picturing shape types of different bias functions.

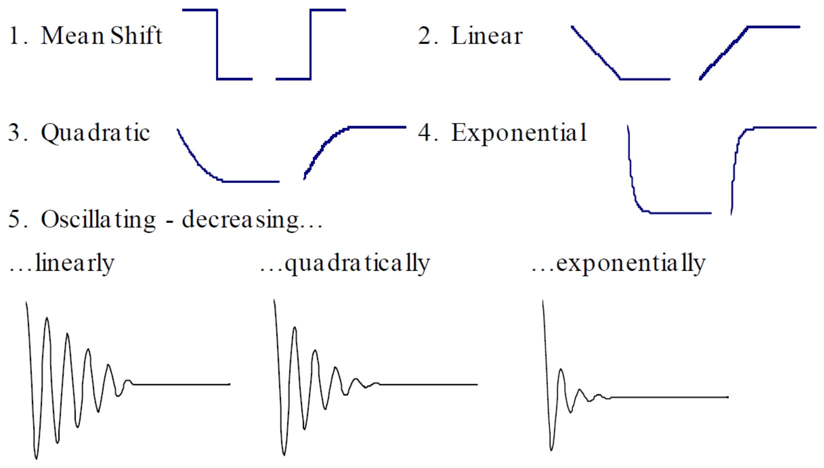

Figure 2.7: Shapes of the Initial Bias functions [HRD08]

The key problem in initial bias detection is to find a minimal truncation point of sufficient size to discard initialization bias. If the initialization bias is overestimated biased values falsify the result. If the initialization bias is underestimated steady-state values are thrown away and longer simulation runs have to be computed. In general, it is better to underestimate the initialization bias as this does not falsify the results.

The overall simulation speed for steady-state analysis profits from a short length of initialization bias. Some initialization bias detection approaches depend on user interaction like e.g. the method of Welch. In our opinion, approaches should not depend on user interaction to alleviate the combination with steady-state approaches. Further, initial bias detection approaches should be accurate and fast.

Many approaches to determine truncation point have been proposed. A general classification of initialization bias identification methods was done by Robinson [Rob04]. He classifies the approaches into the following five types:

- **Graphical methods** involve the visual inspection of time-series of the output data.
- **Heuristic approaches** apply simple rules with few underlying assumptions.
- **Statistical methods** build upon the principles of statistics for determining the warm-up period.
- **Initialization bias tests** identify whether there is initialization bias in the data. Strictly speaking, these are not methods for identifying the warm-up period but they can be combined with warm up methods for this purpose.
- **Hybrid methods** combine graphical or heuristic methods with an initialization bias test.

The following remarks on initialization bias methods base on [Rob04]. The drawback of graphical methods is the fact that they depend on user interaction for determining the truncation point. Examples are the method of Welch [Wel81, Wel83] and a simple time series inspection as introduced by Gordon [Gor69].

One heuristic approach that is very popular at the moment is Marginal Standard Error Rule-5 (MSER-5). Pasupathy [PS10] provides a survey of historical evolution of MSER-5. The Marginal Standard Error Rule (MSER) idea was first proposed as Confidence Maximization Rule (CMR) in the master thesis of McClarnon [McC90]. She proposed the use of the MSER statistic to determine initial bias. Franklin and White [FW08] see the strength of MSER-5 in the fact that it starts from the premise that observations near the end of a simulation run are most representative of the steady-state behavior. This premise allows the MSER-5 heuristic to work backwards.

Examples for statistical approaches are the Algorithm for a Static Dataset (ASD) and Algorithm for a Dynamic Dataset (ADD) proposed by Bause and Eickhoff [BE03] or the regression method of Kelton and Law [KL83].

Examples for the initialization bias tests are Schruben's maximum test [Sch82] and its successor Schruben's modified test [Nel92].

Besides the named approaches there are a lot more. Hoad et al. [HRD08] use Robinson's classification scheme and group 42 methods for initialization bias identification they found in literature. After the extraction of a short list the most promising algorithms are tested. They conclude MSER-5 approach to perform best and most consistently. MSER-5 confirms the good results in other experiments [WCS00] [MI04]. Freeth [Fre12] proposes a forecasting method. This recently proposed approach has not been reviewed externally so far. Summing up, we propose a short survey of initialization bias detection techniques.

2.2.2.2 Steady-State Statistics

By now, we know when steady state starts. During steady state we gather statistics to later combine them into a description. The statistics to search for in steady-state analysis are the mean μ, the variance σ and the corresponding *confidence interval*. The confidence interval describes the number of samples we have to gather during simulation to get reliable results. By that, the confidence interval determines the length we have to simulate the steady state.

We assume the stochastic process consists of independent and identically distributed random variables X_i with $i = 1, 2, \ldots$. Further, we assume X_i to have a finite mean μ and variance σ. The variance σ is unknown but we know that the sample variance $S^2(n)$ converges to σ for large n. Using this and the central limit theorem one gets a confidence interval for μ [Law06]. For sufficiently large n the confidence interval is given as:

$$\mu(n) \pm z_{1-\alpha/2}\sqrt{\frac{S^2(n)}{n}} \tag{2.2}$$

where $z_{1-\alpha}$ is the upper $1 - \alpha/2$ critical point for a standard normal random variable. Unfortunately it is not easy to determine what sufficiently large for n means. If the variance is unknown it can be estimated by the sample variance. In such scenarios the student t distribution is used. The student t distribution is a continuous probability distribution to estimate the mean in case the number of metered samples is very small. We replace the standard normal distribution by the student t distribution. The modified confidence interval looks as follows:

$$\mu(n) \pm t_{n-1,1-\alpha/2}\sqrt{\frac{S^2(n)}{n}} \tag{2.3}$$

The variable $t_{n-1,1-\alpha/2}$ is the upper $1 - \alpha/2$ critical point for the t distribution with $n - 1$ degrees of freedom. It is obvious that the estimation of the variance increases the confidence interval. Hence, the student t distribution is less peaked and has longer tails compared to the normal distribution [Law06]. With an increasing number of samples n the value for degrees of freedom increases and the student t distribution approximates the standard normal distribution. Although the difference between both distributions is small Law [Law06] recommends using the t confidence interval described in Equation 2.3.

According to Law [Law06], there are two basic approaches for point and confidence interval estimation. Sequential procedures work iterative until a certain precision has been reached. In contrast, fixed-sample-size procedures determine the steady-state length according to an initially chosen sample length.

Sequential Procedures increment the length of a single simulation run until an *acceptable* confidence interval can be constructed. Several techniques exist to determine the termination of the simulation run.

Fixed-Sample-Size Procedures perform a single simulation run of fixed length. Then the confidence interval is constructed from the available data.

Sequential procedures are more simple to parameterize, but fixed-sample-size methods are more efficient, as they do not have to determine the confidence interval incrementally. Moreover, fixed-sample-size procedures are often the basis for sequential procedures. Therefore, we focus on fixed-sample size procedures. Law [Law06] lists the following procedures: replication/deletion, batch means, autoregressive, spectral, regenerative and standardized time series analysis.

The replication/deletion approach is based on n independent *short* replications of length m observations. Replication/deletion can be applied to reduce variance. The advantage is that the simulation runs are independent of each other.

The five other approaches are based on one *long* simulation run. They have the advantage of simulating the transient period only once [Law06].

2.2.3 Parallel Simulation

Parallel simulation is one option to improve performance and scalability of simulation. It is helpful to classify the amount of parallelization methods. We prefer a subdivision in three orthogonal abstraction levels for event-discrete simulations to which parallelization can be applied. This classification is used for example in [Kau87] and [Jür97]. The three orthogonal levels are:

- **Application Level** describes the execution of separate simulation runs in parallel.
- **Event Level** uses the parallelism of the model. The simulation is partitioned into Logical Processes (LPs). Each processor executes a subpart of all events.

- **Functional Level** describes the parallelism independent of simulation runs. This can be parallelism in the analysis of a simulation run. Further, this can be the extraction and processing of functional helpers concurrent to the simulation run. Examples for functional helpers are event set processing, input/output processing and random number generation [Jür97].

Another classification was done by Palinkowski et al. [PYM94]. They distinguish between Multi Replication In Parallel (MRIP) and Single Replication In Parallel (SRIP). MRIP describes the run of independent sequential simulations on a set of concurrent CPU which is comparable to parallelism on application level. SRIP describes parallelism on event level.

On event level the simulation is partitioned in subparts called LPs. Each LP preserves its events to process in a queue. The decomposition of the event set in LPs can be done in three different ways. Figure 2.8 shows each decomposition method in a space-time diagram.

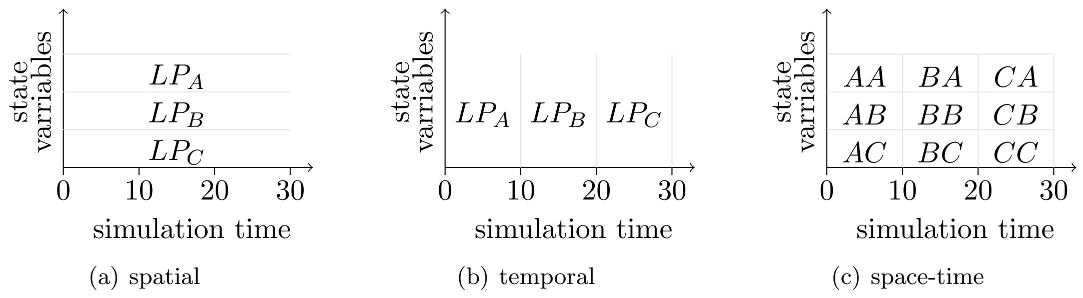

Figure 2.8: Decomposition of the Space Time Diagram into LPs

Temporal decomposition describes the horizontal split of the space-time diagram. Each LP performs a simulation of the entire system for the interval of simulation time covered by its strip of the space-time diagram. Further names for this approach are time parallel simulation [Fuj00b] or time scale decomposition [AD91]. *Spatial decomposition* describes the vertical split of the space-time diagram. The simulation network is split into subparts, often connected subnets. Each subpart-LP simulates the whole time interval. *Space-time* describes the vertical and horizontal split of the space-time diagram. This approach is a combination of spatial and temporal decomposition. If the parallelism derived by temporal or spatial decomposition is not sufficient the combination results in a higher amount of parallelism.

In case time parallel approaches are applicable, they are preferred to apply as LPs avoid expensive synchronization throughout the simulation. They enable massive parallelism as the simulated time span often is very long. A temporal decomposition requires time spans of independent simulation. For this purpose a system specification as recurrence equation or state descriptor is required [Fuj93a]. This precondition makes time parallel simulation only suitable for certain classes of simulation problems [Fuj00b].

The common way to parallelize on event level is spatial decomposition. As many LPs process the same time interval the temporal ordering of event processing is predetermined. The order of parallel processed events has to be equal to a sequential execution. LPs communicate by exchanging timestamped messages. Like that the LPs synchronize their advance and ensure the preservation of order. The preservation of order is often referred to as local causality constraint:

Definition 5 (Local Causality Constraint)**.**
The local causality constraint is met if each event, within an LP, is processed in timestamp order

The spatial discrete-event simulation can be classified in pessimistic/conservative and optimistic approaches [Fuj99, Fuj00b]. Pessimistic approaches meet the local causality constraint while optimistic approaches allow violations.

The following explanations on conservative and optimistic simulation base on the book `Simulation Systems` [Fuj00b].

2.2.3.1 Conservative Approaches

Conservative synchronization algorithms satisfy the local causality constraint. Namely, they ensure that each LP processes events in timestamp order.

A common distinction for conservative algorithms is drawn between asynchronous and synchronous algorithms. As well, conservative approaches can be separated in *first generation algorithms* and *second generation algorithms* [Fuj99]. Both ways of notations term the same families of algorithms.

Asynchronous Algorithms

One of the first solutions to parallel discrete-event simulation was the asynchronous null-message algorithm [PYM94]. The null-message algorithm was independently proposed by Chandy and Misra [CM78] and Bryant [Bry77]. Hence, it is also referred to as Chandy/Misra/Bryant (CMB) algorithm. LPs spread the information of a processed event timestamped to the adjacent/neighboring LPs. Each LP saves the received timestamped messages in a FCFS queue. The timestamp of the last message received is a lower bound on the timestamps of subsequent messages the LP will receive in the future. Each LP can process to its lower bound and then has to wait for new messages increasing that lower bound. A LP can process all events that are between current time and the lower bound timestamp without violation of local causality constraint. The lower bound timestamp value is the current simulation time plus lookahead.

Definition 6 (Lookahead)**.**
Lookahead is the time a LP can look ahead from the current time and process independently of other LPs without violation of the local causality constraint.

Lookahead is a complex function which highly depends on the simulation problem and the way it is programmed [Fuj88]. The lookahead varies with time and the type of event [Fuj88].

We regard a simple synchronization algorithm that solely sends processed token information to the receiver of the token. This information is stored in incoming queues. The messages in each incoming queue are sorted by timestamp. If each incoming queue contains at least one message LPs can ensure the local causality constraint by processing only smallest timestamps [Fuj00b].

This simple algorithm that shares only processed token information does not meet the above constraint. This simple timestamp synchronization can lead to a deadlock so that all LPs have to wait for each other. Consequently, the simulation state is frozen and cannot advance. Figure 2.9 pictures such a deadlock scenario. `LP1` is waiting to receive a message from `LP2`. `LP2` is waiting for `LP3`. `LP3` is waiting for `LP1`.

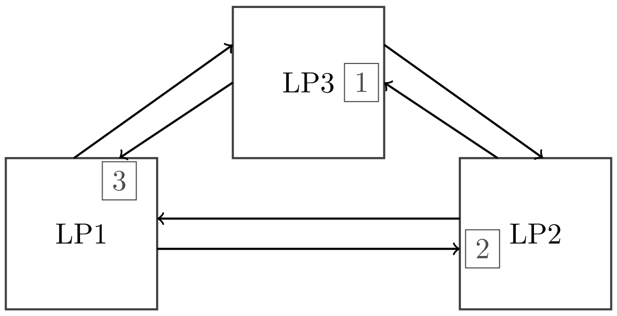

Figure 2.9: All LPs have simulation time smaller than 1. Each LP waits for the other LPs to send a newer timestamp to process its event. Without further information the simulation is deadlock.

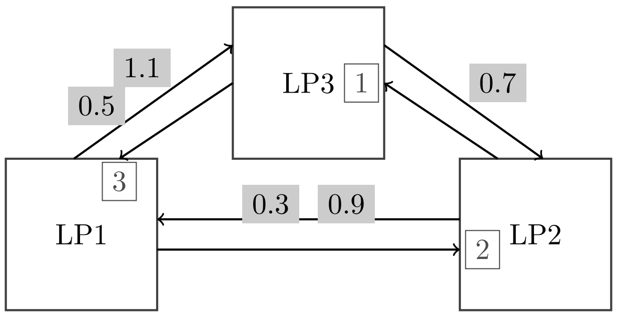

Figure 2.10: Simulation Time Creep. LP2 starts at simulation time `0.1`. The delay between LPs (lookahead) is `0.2`. The first null-message is send with timestamp 0.3 from LP2 to LP1. LP1 notifies LP2 and so on. The null-message algorithm needs 5 null-messages until the next event with timestamp 1 can be processed in LP3.

One method of deadlock avoidance is the use of null-messages. LPs send null-messages on each outgoing link after the processing of each event. A null-message contains no information except for a timestamp which increases the time interval for safe event processing in the receiving LP. Null-messages are processed like non-null-messages. If lookahead is small, there may be many null-messages which is exemplified in Figure 2.10. The key problem is that LPs can only advance in (small) lookahead increments. This ”simulation time creep” [Fuj00b] is solved by second generation algorithms.

Synchronous Algorithms

Synchronous algorithms use a central barrier for communication. The simulation time creep can be solved by including information on the timestamp of the next unprocessed event in computing Lower Bound on the Time Stamp (LBTS) for the next emitted token.

Definition 7 (Lower Bound on Timestamp)**.**
The Lower Bound on the Time Stamp (LBTS) for an LP is the lower bound on the time stamp of any message it can receive in the future. [Fuj00b]

This enables the synchronization protocol to advance immediately to the next unprocessed event. It is helpful to stop the computation at a barrier to make a snapshot of the system. This simplifies the determination of the smallest timestamped event.

A LP waiting for safe events to process is named *blocked*. Otherwise, LPs are named *unblocked*. Pursuant to this notation all LPs waiting at the barrier are blocked. Given a snapshot of the computation, LBTS is the minimum among

- Unblocked LPs: Current simulation time + lookahead
- Blocked LPs: Time of next event + lookahead

The assumption for unblocked LP works for blocked LPs as well. However, blocked LP produces a superior LBTS estimation than unblocked estimation. On condition that all LP are blocked the LBTS calculation can be improved. The knowledge of the next event is referred to as *conditional information* whereas the estimation for unblocked LPs bases on *unconditional information* [Fuj00b].

The difference between asynchronous and synchronous barrier-based algorithms is that barrier-based algorithms can utilize conditional information.

In general, *transient messages* have to be considered for LBTS calculation. A transient message is a message that has been sent, but not yet received. Simulation engine has to wait for transient messages to arrive. This plays a role in distributed simulation but on parallel machines transient messages do not occur.

The barrier changes the way of communication. While the communication of CMB is asynchronous, the barrier algorithms use synchronous communication.

Many synchronization algorithms work with a synchronous barrier. Examples are the bounded lag algorithm proposed by Lubachevsky [Lub89], Chandy and Sherman's conditional event approach [CS89], Ayani's distance between objects algorithm [Aya89], Nicol's Yet Another Windowing Network Simulator (YAWNS) protocol [Nic93] and Steinman's Time Buckets protocol [Ste91].

YAWNS protocol waits for all threads to run into the barrier. Then the event with the smallest timestamp is determined followed by the creation of a time-window in which events are safe. All LPs process to the end of the synchronization window. The size of the synchronization window is the worst-case lookahead among all LPs.

The conditional event approach also uses a global min-reduction to determine the LBTS for all LPs to safely process events. The LBTS is set to the minimum of earliest conditional events plus the lookahead among all LPs.

Besides the barrier solutions, another solution to the simulation time creep was proposed by Chandy and Misra [CM81]. They allow deadlocks and introduce a detection mechanism followed by a deadlock resolution. The deadlock can be resolved by having each LP sending null-messages indicating a lower bound on the timestamp of future messages. The messages with the smallest timestamps in the entire simulation are always safe to process. By that, we have the next event(s) to process. The deadlock can be resolved by sending null-messages to all LPs owning an event with smallest timestamp. The smallest timestamp can easily be determined as the simulation is deadlocked and the creation of new events has stopped. This solution utilizes the fact that if all LPs are blocked, they can immediately advance to the time of the minimum timestamp event in the system.

Assessment of Conservative Approaches
The development of conservative synchronization algorithms has achieved algorithms feasible in real-world simulation problems [Fuj00b]. The success of parallelization strongly depends on the lookahead characteristics of the model. Pawlikowski [PYM94] states that a reasonable speedup is possible, provided that a given simulation model is highly decomposable. Lookahead plays a crucial role in the performance of conservative synchronization algorithms. Hence, the focus on simulation application development is to maximize the lookahead [Fuj00b]. Even if the model has a healthy amount of parallelism conservative algorithms struggle to achieve good performance as soon as the simulation application has poor lookahead [Fuj00b]. Until now, there is no reliable automatic lookahead determination within an application and code restructuring in favor of improving lookahead[Fuj00b].

Conservative synchronization algorithms can only progress until LPs reach their lookahead border. By that, conservative approaches cannot fully exploit the inherent parallelism of the model [Fuj00b]. Optimistic approaches try to exploit this inherent parallelism to a greater extend.

2.2.3.2 Optimistic Approaches

In contrast to conservative synchronization, optimistic synchronization allows violations of the local causality constraint to occur and provide mechanisms to recover [Fuj00b]. Jefferson [Jef85] introduced the basic *time warp* algorithm that actual optimistic methods are build on. It introduced many fundamental concepts as rollback, anti-messages, and Global Virtual Time (GVT).

Jefferson [Jef85] introduced a Time Warp Logical Process (TWLP) that differs from the known LP by the fact that events in the event set may be changed by messages of other LPs. Hence, processed events are not thrown away but preserved in a queue. This is necessary to roll back and reprocess previously processed events. A TWLP queue shows processed and unprocessed events.

A violation of the local causality constraint occurs if a message arrives with timestamp $t_i <$ Local Virtual Time (LVT). These late arriving messages are called *straggler messages.* If the TWLP has not sent any message with a higher timestamp only the local history has to be refreshed. Otherwise the sent messages have to be undone. The corresponding time warp mechanism is called *anti-messages.* An anti-message deletes its corresponding message in the receiving TWLPs.

If the local causality constraint is violated, a rollback is performed. A rollback needs earlier states. These states need to be stored in a queue which leads to a high amount of memory allocation. If we do not want to preserve all earlier states, we have to introduce a lower bound on the timestamp of any future rollback. This lower bound is named Global Virtual Time (GVT). Besides the reduction of memory costs the GVT reduces the possible rollback length.

Definition 8 (Global Virtual Time)**.**
The GVT at wallclock time T (GVT T) during the execution of a time warp simulation is defined as the minimum timestamp among all unprocessed and partially processed messages and anti-messages in the system at wallclock time T [Fuj00b]

The original time warp has performance hazards. While the rollback/anti-message mechanism tidies incorrect simulations the incorrect message propagates through the system. In worst case the incorrect computation stays one step ahead of the canceling operations. Then the wrong message will never be canceled. The effect is commonly known as *dog is chasing its tail* [LAH99, Fuj00b]. This effect is enhanced by a low level of parallelism. A low level of parallelism raises the probability that incorrect computation is spread directly [Fuj00b]. An example of the dog is chasing its tail effect is depicted in Figure 2.11:

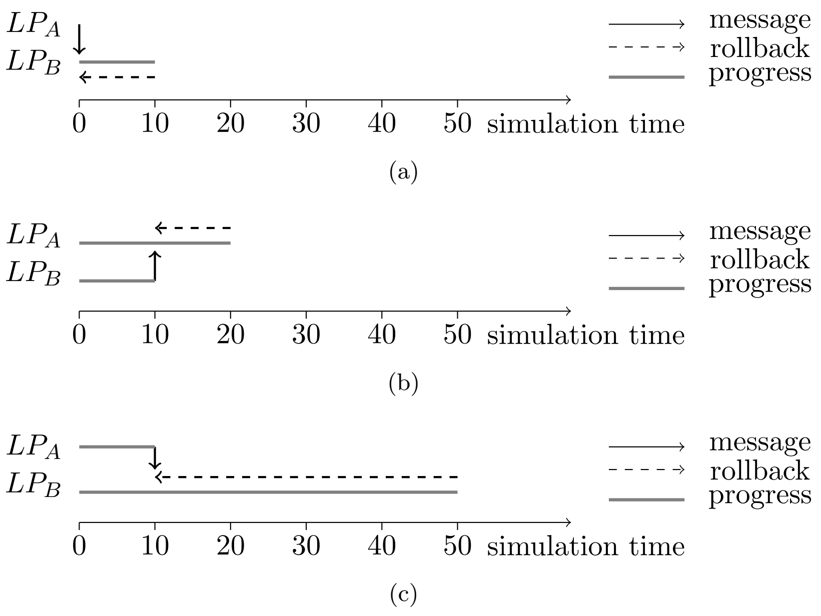

Figure 2.12: Echo Example [Fuj00b] (a) LP_A sends a message to LP_B, causing LP_B to roll back (b) While LP_B rolls back, LP_A advances in simulation time (c) Sequence repeats with the length of rollback expanding at an exponential rate.

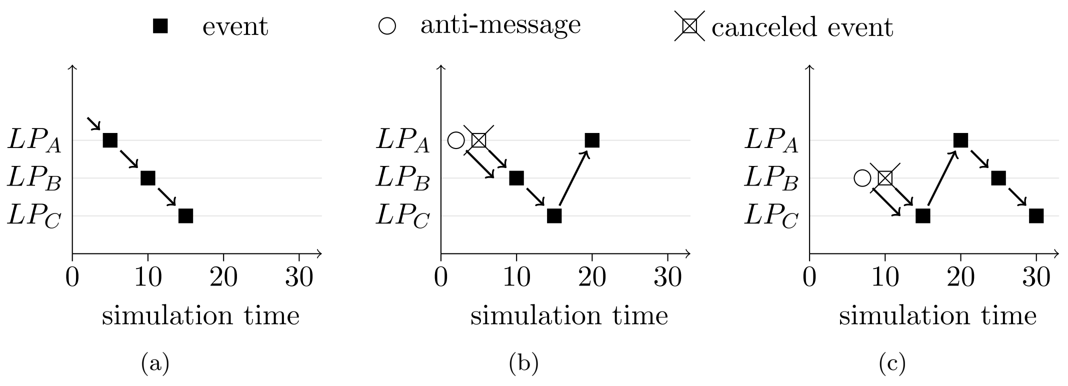

Figure 2.11: Dog Chasing its Tail Effect

Another performance hazard is called *rollback echo*. An echo occurs if the rollback consumes more time than the forward progress. We explain the echo effect by an example of [Fuj00b] pictured in Figure 2.12. The scenario consists of two logical processes LP_B and LP_A that are mapped to different processors. LP_B has advanced 10 units of simulation time from LP_A. Subsequently, LP_A sends a message to LP_B. LP_B recognizes that it has advanced too far and rolls back to simulation time zero (2.12(a)). At the same time LP_A advances 20 units of simulation time. Then LP_B notifies its timestamp to LP_A which causes LP_A to roll back (2.12(b)). During the rollback of LP_A, LP_B advances 40 units of time which causes a rollback of the same length (2.12(c)).

Advanced Optimistic Approaches

There exist several approaches to avoid performance hazards. We will provide a short overview.

Moving Time Window (MTW) sets a time window of size W which defines how far ahead one LP can advance from the other LPs. In the technical solution this means that LPs are not allowed to process beyond GVT $+W$.

Window-based approaches block without reference to model properties. In contrast, lookahead-based blocking tries to use the information obtained by applying a conservative synchronization protocol. Safe events can be processed without further control mechanisms whereas unsafe events are processed with optimistic synchronization mechanisms. This leads to hybrid conservative/optimistic synchronization algorithms.

The Breathing Time Buckets (BTB) approach combines synchronous event processing with barriers, time windows and local rollback. Local rollback ensures that LPs do no spread wrong messages. It is not allowed to send messages until it can be guaranteed that they will not be rolled back.

The combination of multiple time warp extension has been performed in the Georgia Tech Time Warp (GTW) [FDP$^+$97]. The GTW includes the proposed and many other time warp extensions.

Assessment of Optimistic Approaches
In general, optimistic approaches can exploit more parallelism than conservative approaches. Optimistic approaches are not sensitive to lookahead characteristics. Disadvantages of optimistic approaches are the high memory consumption and the additional overhead. Many approaches tend to be overoptimistic which causes expensive rollbacks.

2.3 QPME

Queueing Petri Net Modeling Environment (QPME) [KS09] is a tool for modeling and simulating QPNs. Together with the HiQPN[BBK95] tool developed at University of Dortmund, QPME is the only tool to model and analyze QPNs. HiQPN analyzes QPNs by a Markov chain transformation and the resolution of the balance equations. According to [TM06] the HiQPN tool greatly suffers from state space explosion problem and takes huge amount of resources and time for large number of user requests and/or components. In contrast to HiQPN, QPME analyzes the model with discrete-event simulation which avoids the state space explosion problem. QPME uses SimQPN, a highly optimized simulation engine, for QPNs analysis. The graphical editor QPE enables to build intuitively QPNs.

QPME is often used for the performance analysis of computer systems. Many case studies prove a successful usage of QPME for modeling and analysis in various scenarios [Kou06, KNT07, KSBB08, NKJT09, KS09, Sac10, SKB12].

The development of QPME started 2003 at University of Darmstadt. The first release was published 2007 in Cambridge. Since the first release, the tool has spread to over 120 universities and organizations [SKM12]. Currently, the tool is developed and maintained by the Descartes Research Group at KIT [SKM12]. It is available as open-source software under Eclipse Public License v 1.0 since May 2011. QPME is fully java-based and therefore easy portable to various platforms.

SimQPN supports *batch/means* and *replication/deletion* for steady-state analysis and the *method of Welch* for manual truncation point analysis. Besides the parametrization for theses simulation procedures, SimQPN enables to set more parameters to specify statistic collection. SimQPN collects statistics on a per location basis where a location has one of the following four location types [KS11]:

1. Ordinary place.
2. Queue of a queueing place (considered from the perspective of the place).
3. Depository of a queueing place.

4. Queue (considered from the perspective of all places it is part of).

For each location a *stats-level* can be set. The stats-level determines the amount of statistics to be gathered for this location during the run. A level of 0 means no data collection. A level of 1 means lowest output detail, level 5 yields the most detailed output files. The more detailed description from [KS11] :

stats-level 0 In this mode no statistics are collected.

stats-level 1 This mode considers only token throughput data, i.e., for each location the token arrival and departure rates are estimated for each color.

stats-level 2 This mode adds token population, token occupancy and queue utilization data, i.e., for each location the following data is provided:

- Token occupancy (for locations of type 1 or 3): fraction of time in which there is a token inside the location.
- Queue utilization (for locations of type 2 or 4): proportion of the queue's server resources used by tokens arriving through the respective location.
- For each token color of the respective location:
  - Minimum/maximum number of tokens observed in the location.
  - Average number of tokens in the location.
  - Token color occupancy: fraction of time in which there is a token of the respective color inside the location.

stats-level 3 This mode adds token residence time data, i.e., for each location the following additional data is provided on a per-color basis:

- Minimum/maximum observed token residence time.
- Mean and standard deviation of observed token residence times.
- Estimated steady state mean token residence time.
- Confidence interval (c.i.) for the steady state mean token residence time at an user-specified significance level.

stats-level 4 This mode adds a histogram of observed token residence times.

stats-level 5 This mode additionally dumps token residence times to a file for further analysis.

3. Approach

This chapter describes our approaches to reach the previously declared goal of the thesis to evaluate different strategies for parallelizing a Queueing Petri Net (QPN) simulation. Before we go into detail, we refine our goal into subgoals. The goal can be split in the following sub-goals:

- **Design** The aim for the theoretical part is to identify different techniques for the parallelization of event-discrete simulation. This sub-goal includes an analysis of their applicability to QPN simulation and the selection of a subset of the identified techniques for implementation.
- **Implementation** We define the following sub-goals for the implementation:
  - Develop a new software architecture for SimQPN which supports parallel simulation. Then refactor the current SimQPN code towards the new design.
  - Implement parallelization techniques in the SimQPN simulator.
- **Evaluation** We define the following sub-goals for the evaluation:
  - Validate the correctness of the implemented parallel techniques by comparing the parallel and sequential version.
  - Evaluate the techniques in order to determine the performance improvements through parallelization.

According to our subgoals, our approach can be split into identification, implementation and evaluation of parallelization approaches. Section 3.1 explains how we identify the parallelization approaches to implement. Section 3.2 describes the approach we chose for the implementation. Section 3.3 specifies how we evaluate the implemented approaches.

3.1 Design

We consider three levels of parallelization: application level, functional level and event level.

Application level Exploiting the parallelism on application level means to run multiple replications in parallel. This parallelization level is favorable as each replication can run on its own CPU without any synchronization effort. Application level parallelization requires analysis approaches with multiple runs. We determine the approaches where application level parallelism is applicable.

Event level There is a lot of previous work on the parallelization on event level. For the purpose of event level parallelization we performed an extensive literature research. We provide a survey of the state-of-the-art methods and algorithms. We identify three major issues that need to be solved for a successful parallelization:

- synchronization algorithm,
- decomposition, and
- lookahead.

The discussion on synchronization algorithms considers their strengths and weaknesses and compares them to the characteristics of QPNs. The characteristics that determine the capability of our QPN models for different synchronization algorithms depend to some extent on decomposition and lookahead.

Synchronization algorithms require a model to be decomposed into disjunct parts. To the best of our knowledge, there is no publication on QPN decomposition for parallelization. Therefore, we review strategies proposed for QNs and PNs. In addition, we discuss extensions for QPNs.

At lookahead calculation, the situation is similar to decomposition. A general discussion on lookahead for QPNs has been absent so far. Again we employ strategies from QNs and PNs.

Functional level On functional level we can extract certain parts of the simulation logic into helper functions and execute them in parallel to the simulation. Furthermore, we can also run the analysis of simulation results in parallel to the simulation. A helper function is a candidate for extraction and parallelization if it has a remarkable impact on performance and offers a service that can be computed parallel to the not extracted part. This service is either a precalculation or a subsequent calculation which is not instantly accessed by the core simulation. Possible helper functions have to be determined. To find functional helpers we access experiences form other publications and analyze the source code for sequential simulation.

3.2 Implementation

In this section, we explain our approach on the adjustment of SimQPN towards a parallel simulation engine. We explain requirements and point out difficulties for implementation.

SimQPN is a grown software system. Before starting to parallelize the simulation, we refactor SimQPN to improve modularity and state encapsulation. This refactoring is necessary to integrate the new parallel simulation engine into SimQPN.

Application level We chose the Replication/Deletion approach for application level parallelization. For the application level we do not need to parallelize the core simulation loop. The existing sequential simulation engine can be reused and just be started multiple times in parallel.

Event level We parallelized the Batch/Means approach on event level. A specialty of sequential SimQPN is that it uses a mixture of process and event based scheduling. The firing of transitions is implemented based on process scheduling. SimQPN creates only queue-events for scheduling tokens in queues. This reduces the number of events within the simulator. The parallel implementation keeps this performance optimization. Apart from this performance optimization the SimQPN behaves like other discrete event simulators.

To implement event level parallelization techniques sequential simulators need some adaptation. For SimQPN, or any other sequential simulator, adjustments at the following parts have to be performed for parallelization:

- net decomposition
- simulation logic
- progress communication
- lookahead calculation.

The net needs to be decomposed into LPs. Besides decomposition, we implement a merge function. This merge function enables to merge two adjacent LPs. Thereby, the size of LP can be varied.

The simulation logic can be reused with minor changes from the sequential version to a great extend. New parts to implement are the handling for incoming tokens and progress communication. The progress communication mechanism depends on the synchronization algorithm. The considerably more complex part of progress communication is the implementation of lookahead for different queueing strategies. Last but not least, dependent on the chosen synchronization algorithm further parts need to be implemented.

Functional Parallelism It is not useful to parallelize every possible functional helper. Before extracting helper functions we evaluate their performance impact. We meter the share on overall runtime before we implement parallel execution of functional helpers. We search for promising functions by the help of a profiler. During search we kept a special focus on common helper functions.

3.3 Evaluation

This section presents the procedure of parallel SimQPN evaluation. We subdivide the evaluation into validation and a performance testing.

3.3.1 Validation

We performed the validation on existing example models from previous case studies. SimQPN selects statistics during simulation. These statistics are for example the token arrival and departure counts, mean service time, mean token occupancy. To test for correctness, we compare the resulting statistics for equality.

An existing test framework for SimQPN assumes equality by a t-test with significance level of five percent. Within the test framework the results alternate because of a randomly chosen initial random seed. To improve faith in the precision of the parallel implementation, we performed experiments where we set the initial random seed equal for the compared runs. This enables to demonstrate the precision of the parallel implementation.

3.3.2 Performance Impacts

After the validation we evaluated for performance impacts and overheads introduced by the parallel SimQPN version. The analysis for performance improvements is split into the analysis of Replication/Deletion and Batch/Means.

The parallelization of Replication/Deletion is on application level. We test the speedup on a model from an existing case study. The Batch/Means approach has been parallelized on event level. On event level the speedup depends on model characteristics. We test parallel implementation on artificial models where we can determine and modify degree of parallelism and workload. The artificial models enable a more systematic evaluation of certain model characteristics compared to the complex models form case studies. This further enables distinct conclusions about strengths and weaknesses which are not blurred by mixed characteristics.

The experiments on performance impact of parallelization compare the run times for the sequential and the parallel version. By the help of artificial models we investigate effects of multiple model characteristics on performance. Experiences from QPN subformalisms show that not all models suit for parallel simulation. Our experiments help to asses the suitability of models inspecting different model characteristics. The experiments consider the following aspects:

- Level of statistic collection
- Events between barrier synchronizations
- Number of concurrently processing LPs

SimQPN determines the statistics on a per net element basis. We evaluate the impacts of six statistic levels on performance. SimQPN does not collect any statistic data during initialization bias. Thus initialization bias can be emulated by statistic level zero. This enables to discuss the suitability of parallel simulation for the different statistic levels and for the initialization bias. Besides, we explore the effects of different workloads by an increase of events between synchronizations. Furthermore, we modify the model inherent parallelism which influences the number of concurrently processing LPs.

4. Related work

In this chapter, we provide an overview of previous research on approaches to parallel and distributed simulation. The related work can be divided into publications on parallel and distributed simulation in general and approaches applying these principles to the analysis of Petri Nets (PNs) and Queueing Networks (QNs). At first, we refer to general publications on concurrent simulation in Section 4.1. Section 4.2 is about publications on concurrent simulation of PNs and its extensions. Parallelizations of QN simulation are discussed in Section 4.3.

4.1 Parallel and Distributed Simulation

Discrete event simulation is a complex field. Hence, many publications exist on the difficulties and opportunities without focus on a concrete domain. One very famous publication of Fujimoto [Fuj93b] asks whether the field of parallel discrete event simulation will survive.

An approach that works well, independent of the modeling formalism, is the parallel execution of multiple replications. Pawlikowski et al. [PYM94] consider traditional approaches and conclude to apply Multi Replication In Parallel (MRIP) to distributed stochastic discrete-event simulation instead of parallelizing the execution of a single run. They provide an implementation with linear speedup within the AKAROA simulation package. AKAROKA automatically creates the environment to run MRIP on the workstations of a local area network.

In many scenarios the problem cannot be decomposed into multiple runs. If we cannot subdivide the problems into multiple runs the execution of a single run has to be parallelized. Implementations often focus on applications areas, model characteristics and hardware at hand. This complicates comparability and repeatability. Thereby, the choice for an optimal algorithm is difficult. Multiple conservative and optimistic algorithms have been proposed. Ewald et al. [EHU$^+$06] try to improve the choice for a scenario dependent parallelization with the help of simulation.

Himmelspach et al. [HELU10] employ large scale simulations where parallelization can be performed on multiple orthogonal levels. Hereby, hardware resources are limited and often not sufficient for doing everything in parallel. Himmelspach et al. try to exploit the available hardware best by an efficient combination of parallel tasks. They implemented the simulation system JAMES II which is a proof of concept for their high-level approach to parallelization.

Besides the conservative synchronization algorithms named in Section 2.2.3.2 there exist a lot more. As an example we mention a few optimizations for conservative simulation dependent on model characteristics.

Such special model characteristics may occur at large models. More precisely, we refer to large models where distributed parts cannot affect each other. The conservative barrier-based *bounded lag* algorithm by Wagner et al. [WLB88, WL89] utilizes this fact. The bounded lag algorithms allows only those LPs to communicate, which actually can affect each other. By that, the communication overhead can substantially be reduced.

Lookahead characteristics influence the choice for algorithms. Liu and et al. [LNT01] assume lookahead to be fixed. This precondition empowers optimizations and higher speedups. The deterministic behavior enables the avoidance of locking operations. Liu et al. state their conservative asynchronous *lock-free* algorithm performing well when the number of LPs assigned to each processor is small.

Moreover, multiple extensions to the CMB algorithm have been proposed. The approach of Xiao et al. [XUSC99] improves scheduling throughout determination of limiting paths. Their Critical Channel Traversing (CCT) algorithm enables to schedule critical paths first. By that, waiting times can be reduced.

A further approach was proposed by Liu and Rong [LR12]. They combine asynchronous (CMB) and synchronous (YAWNS protocol) synchronization in a hierarchical composite synchronization algorithm.

To the best of our knowledge, adaptations of optimistic simulation algorithms to special model characteristics have not been proposed. General optimizations for optimistic simulation have been discussed in Section 2.2.3.2.

4.2 Parallel and Distributed Simulation of Petri Nets

In this section we focus on publications on parallel and distributed analysis of the PN formalism and its extensions. We see mainly two different approaches. The first family of approaches builds on special Single Instruction Multiple Data (SIMD) hardware. According to Flynn's taxonomy [Fly72], a SIMD hardware exploits multiple data streams against a single instruction stream. Examples for such hardware are Graphics Processing Units (GPUs) and vector processors. This hardware is able to solve vector operations very fast. These vector operations appear when solving PN recurrence equations. The second family of approaches is LP-based simulation as described in Section 2.2.3. The parallelization is independent of the underlying hardware. A more detailed description and comparison of both approaches is provided in [Fer94].

Baccelli and Canales [BC93] parallelize the simulation of SPNs for a SIMD architecture. They transform the net into recurrence equations and then parallelize the emerging matrix operations. Associative matrix multiplication enables to alter the order of multiplications. Baccelli and Canales distinguish two multiplication orders, which represent a spatial and a temporal decomposition. They propose using the spatial approach for large networks and the temporal for small networks. In general, Baccelli and Canales exploit the SIMD architecture to speedup matrix operations.

Geist et al. [GHSW05] use SIMD architecture as well. They propose a method that uses GPUs (NVidia 5-series and 6-series) common in desktop computers. Chalkidis et al. [CNM11] utilize GPUs to analyze biological processes called biopathways. They use the Hybrid Functional Petri Net (HFPN) modeling formalism introduced by Matsuno [MTA$^+$03]. HFPNs include several PN enhancements (CPNs, SPNs and Hybrid Petri Nets

(HPNs)). Hereby, the (not previously listed) HPN formalism enables to model discrete and continuous events.

Multiple LP-based parallelizations have been proposed. Nicol and Roy [NR91] apply a conservative synchronization protocol for discrete-event analysis of Timed (Transition) Petri Nets (TPNs). They identify three types of TPN events. Those event types are the arrival of a token, the start and the end of transition firing. The tripartition of event types enables Nicol and Roy to provide an efficient conservative parallelization for TPNs.

Chiola and Ferscha [CF93] describe how TPNs can be split in LPs to apply conservative and time warp synchronization. They define *minimum regions* by the use of (extended) conflict sets. The minimum regions form a partition of the net such that each transition and each place is in exactly one region. Conflicting transitions and their places reside in the same region. Then the regions are merged according to five merging rules. The merged regions form LPs. In general, they provide a good overview about advantages of different partitionings. They state, for example, that forward incidence functions ($f \in Place \times Transition$) have more impact on performance than backward incidence functions ($f \in Transition \times Place$).

Nketsa and Khalifa [NK01] modified the lookahead definition of [CF93] to better suit asynchronous communication. Fang et al. [FXY07] refine [NK01] approach to fit on extended TPNs.

Ammar and Deng [AD91] apply an optimistic time warp simulation to SPNs. They use a combination of spatial and time scale decomposition to derive LPs. Their time scale decomposition divides a large network into small subnets by separating short activities and long activities into different subnets.

Ferscha [Fer99] tries to make the time warp algorithm more robust to different model domains. He proposes an adaptive time warp algorithm for timed PNs. His approach is based on monitoring of synchronization messages and statistical analysis.

Jürgens [Jür97] applies distributed simulation to Hierarchical Queueing Petri Nets (HQPNs). He implemented a pessimistic and an optimistic synchronization protocol. In his experiments on artificial models the pessimistic protocol reaches better speedups than the optimistic protocol. Two model characteristics that limit speedup are degree of parallelism and the lookahead characteristics. Therefore, he considers distributed simulation to be worthwhile only for a subset of models with good characteristics.

Jürgens' and our own intuition behind the term *degree of parallelism* is different from the definition used in [vdA96] and [VJ03]. In our opinion, Van der Aalst [vdA96] PN-based degree of parallelism is suitable to determine bottlenecks in Business Process Management (BPM) scenarios, but does not suit to be a good indicator of the possible speedup a simulation can reach.

Concluding Remarks
Except for Jürgens' [Jür97] approach, none of the presented approaches parallelized the execution of QPNs. In contrast to our approach, Jürgens applies distributed instead of parallel computation. By the distribution, the communication costs consume a lot of the possible speedup.

A further difference is that Jürgens does not take queueing places into account. In his models, timed transitions were used to model the temporal aspects. He does not consider queueing places with scheduling strategies.

The publications on LP based simulation have in common that they investigate formalisms that form a foundation for the QPN formalism. Hence, the proposed publications about conservative and optimistic approaches can help to find an optimal splitting in LPs. Timed Transition PNs are a super class of stochastic PNs. Hence, all of the proposed lookahead research can improve lookahead calculation for QPNs.

The publications that use SIMD architecture transform PNs into recurrence equations. The SIMD publications apply a conservative parallelization approach without lookahead. Their speedup is solely reached by specialized hardware but not by an efficient strategy. This thesis is about parallel simulation and not about parallelizing of recurrence equations. Hence, the SIMD publications are considered to be dispensable in the context of this thesis.

4.3 Parallel and Distributed Simulation of Queueing Networks

Queueing Networks (QNs) have been a common benchmark for the evaluation of different synchronization protocols e.g. [Nic88, RMM88, Fuj88, WLB88]. Conservative QN simulation performed only well in subclasses of the formalism. Many studies conclude conservative simulation to be unsuitable for speeding up the simulation of QNs, or limit it to special subclasses with good lookahead characteristics. Often the models have been restricted to First-Come-First-Served (FCFS) networks.

The reactions to these experiments were different. Some, like e.g. Fujimoto [Fuj89b], switched their focus to optimistic simulation. Others, like e.g. Lazowska [WL89] [LL90], searched for improved lookahead estimations. Wagner and Lazowska [WL89] improve lookahead calculation for non-FCFS scheduling strategies. Their work is of importance because the QPN formalism enables multiple scheduling strategies. Later, Lin and Lazowska [LL90] provided further experiments on the improved lookahead calculation. FCFS networks still provide better lookahead than most other queueing strategies. Nevertheless, a wider range of models can be simulated with benefit. The formalisms QNs and QPNs have the same queueing strategies. Hence, improvements on QN lookahead can improve QPN lookahead calculation.

5. Design

This chapter provides an overview of techniques for parallel simulation of QPNs. Parallelization techniques can be categorized in three orthogonal levels of parallelism. These levels are:

- application level
- event level
- functional level.

The following subsections specify parallelization approaches for each level. Section 5.1 describes the parallelization of multiple runs on application level. Section 5.2 discusses various aspects on event level parallelization. In particular, we discuss synchronization algorithms in Section 5.2.1, net decomposition in Section 5.2.2 and lookahead calculation in Section 5.2.3. Last but not least, we mention possible functional helper functions that can be extracted to be executed parallel to simulation in Section 5.3.

5.1 Application level

Exploiting the parallelism on application level means to run a set of sequential simulation programs on different processors. Application level parallelization promises speedup proportional to the number of cores as each simulation run may run on its own CPU without any synchronization effort. We recommend to apply this parallelization whenever possible. It is applicable to all analysis approaches that perform more than one simulation run. SimQPN supports three analysis approaches as described in Section 2.3.

Replication/Deletion runs n independent short replications each of length m observations. Thereby it imitates a run of length $n * m$.

Batch/Means performs one long run and divides into m batches for analysis.

Method of Welch uses multiple runs to determine initialization bias.

Replication/Deletion and Method of Welch use multiple runs for analysis and lend itself very well for a parallelization on application level. In contrast, Batch/Means does not fulfill the requirement of multiple runs.

By now, the interface of QPME does not support testing multiple parameter settings in parallel. In general, cases exist where different models or parameterizations have to be tested. The emerging runs are candidates for parallelization. A precondition for parallelization is that the executions of analysis approaches are independent of each other. Applications that support a combined analysis approach for examination of different parameter settings can parallelize the execution of corresponding runs.

The application parallel program structure avoids communication between runs during their execution. Algorithm 1 shows the skeleton of application level parallelism.

Algorithm 1 Application level parallelism skeleton

```
 1: function RUN(numRuns)
 2:     runArray[numRuns];
 3:     initializeRunArray();
 4:     for all run in RunArray do
 5:         startSimulationRunThread();
 6:     end for
 7:     for all run in RunArray do
 8:         waitForSimulationRunToJoin();
 9:     end for
10:     combineResultsOfRuns();
11: end function
```

This level of parallelism is not expected to create as complex challenges as the other levels. However, one should keep in mind that each simulation run needs its own random number generation. For some simulators the generation of random numbers has to be adapted.

5.2 Event level

Parallelization on event level is based on a decomposition of the model into LPs. The synchronization algorithm controls how the individual LPs communicate with each other to execute the whole simulation. Lookahead enables LPs to predict the behavior of other LPs. This allows to anticipate future system states and improves the utilization of model inherent parallelism in parallel simulation. This section on event parallel QPN simulation answers the general questions concerning:

- synchronization algorithm
- model decomposition
- lookahead calculation

We analyze QPN lookahead characteristics and compare them to the strengths and weaknesses of synchronization algorithms in Section 5.2.1. Thereafter we describe details of net decomposition in Section 5.2.2 and lookahead calculation in Section 5.2.3.

5.2.1 Synchronization Algorithm

No synchronization mechanism exists that performs best for all scenarios. This section tires to determine the most promising algorithm for the QPN formalism and our application area of performance models.

The choice for the optimal scheduling algorithm encompasses two fundamental decisions. Section 5.2.1.1 compares conservative and optimistic synchronization and explains why we decide for the more promising conservative paradigm. The choice of either asynchronous or synchronous communication is discussed in Section 5.2.1.2.

5.2.1.1 Conservative vs. Optimistic

The most general classification of synchronization algorithms is to differentiate between conservative and optimistic synchronization. Conservative algorithms ensure local causality constraint (Definition 5 in Section 2.2.3.1) whereas optimistic algorithms enable violations. Details on conservative and optimistic synchronization have been set in Section 2.2.3.1 and Section 2.2.3.2.

In the case of conservative algorithms, the individual LPs communicate with each other before they could possibly get too far ahead in time. By this intense communication they avoid any violations of the local causality constraints. If we use an optimistic synchronization, the LPs continue in order to avoid intense communication. We see a fine-grained communication to be the bottleneck of conservative simulation. Communication intervals increase when LPs can estimate the time of the next token emissions. This estimation is called lookahead. Hence, the success of conservative parallelization strongly depends on the lookahead characteristics of the model.

Models with good lookahead characteristics tend to work best with conservative synchronization. Models with poor lookahead characteristics and high inherent parallelism tend to perform better with optimistic synchronization.

Developers rely on model analysis, their expert knowledge and their intuition to judge lookahead characteristics of the models when choosing an appropriate synchronization algorithm. The only experiments on parallel QPN simulation by Jürgens [Jür97] state the conservative approach to perform better than the optimistic approach. The overhead for the prevention of straggler messages or the otherwise costly rollbacks introduce high overheads that hinder speedup. His experiments make conservative simulation appear more promising. Based on his experiences and recommendations, we decide to use conservative synchronization for parallelizing SimQPN.

5.2.1.2 Asynchronous vs. Synchronous

The next step is to decide about the communication mode. This decision on communication mode is independent of conservative and optimistic synchronization. The communication between LPs is either asynchronous or synchronous. Asynchronous communication employs direct neighbor communication between LP. Synchronous algorithms use a central barrier to control simulation progress.

Sometimes, according to their order of publication, conservative approaches are grouped into first generation (asynchronous) and second generation algorithms (synchronous). To explain the difference between these two paradigms we use differentiation of unconditional and conditional information from [Fuj00b].

Unconditional Information describes guarantees based on local information. For example, if an LP has advanced to simulation time T, and it has a lookahead of L, then the LP can unconditionally guarantee to its successors not to send messages before T + L

Conditional Information Conditional information is information provided by a LP that is only guaranteed to be true if some predicate is true. On condition that no token will arrive before the actual token will finish we we can set the LBTS (Definition 7 in Section 2.2.3.2) to finish time of the actual token.

The referenced algorithms in the following have been described in Section 2.2.3.1. Asynchronous algorithms, like CMB, transmit only unconditional information. Synchronous barrier-based algorithms supplementary utilize conditional information. Conditional information provides benefits if the estimation for the next arriving token is poor so that we expect the arrival of a token soon, although it takes time until the next token arrives. In this case the lookahead is smaller than the actual service time of the token in the queue or in the timed transitions. Models containing cycles with small lookahead entails simulation time creep (has been depicted in Figure 2.10) at asynchronous communication. The probability of such simulation creep in QPN models cannot universally be answered. To judge this probability we have to go deeper into lookahead characteristics of QPN.

The QPN formalism unifies many formalisms. From its sub-formalisms QPNs got elements with good and poor lookahead characteristics. This enables to build well and poor performing models for parallel simulation. In the following, we focus on the parts of these formalisms which hinder a good lookahead computation. Here, especially some queueing strategies have to be reviewed.

Every queue has a service time distribution, which determines the service times. Independent of scheduling strategy, the minimum of the service time distribution is a lower bound for the lookahead estimation. No token can pass faster than the minimum service time. In general, the service times for queues may be chosen according to arbitrary distributions. Performance models often show distributions with minimum set to zero which means a zero lookahead. Thus we have to review the queueing strategies individually.

The prediction of lookahead in FCFS queues is a feasible problem. Incoming tokens cannot pass tokens that are already enqueued. The token that has entered the first will be the next which finishes service. The service time of the first token determines the maximum lookahead. Hence, we cannot improve the LBTS calculation by the use of conditional information.

In contrast, IS and PS come along with poor lookahead characteristics. These queueing strategies allow incoming tokens to surpass currently enqueued tokens at every point in time. Additionally, performance models tend to have service distributions with a lower bound of zero. Then we cannot set a lower bound on the next emitted token. Consequently, we cannot predict lookahead. The predictability of IS and PS queues could greatly be improved if we assume fixed service times. Fixed service times ensure no passing of tokens. The lookahead can be estimated equal to the fixed service time. Fixed service times are a special case. As mentioned before distributions may have its minimum set to zero. The worst case scenario of zero lookahead is likely.

Asynchronous conservative algorithms, like CMB, require to ensure progress on each cycle within the net. At least one lookahead value on each cycle between LPs has to be bigger than zero. If a cycle with zero lookahead exists, first generation algorithms cannot be applied. In case of QPN performance models we cannot exclude the case of zero lookahead cycles. To ensure the application of CMB algorithm the model can be modified. The minimum of distributions is set to a small non-zero value which would only marginally affect final results. However, applicable does not mean suitable. In such a scenario, we ensure a non-deadlocked CMB but simulation time creep is very likely.

The model characteristics may vary at different application areas. Jürgens' [Jür97] models descend from transportation scenarios and provide different characteristics than performance models. In his models timed transitions are used to model the temporal aspects. He does not consider queueing places with scheduling strategies. The models shipped with QPME descend from performance engineering. Here, queues are used to model delays and scheduling behavior. Scenarios with IS or PS are more likely to produce lookahead creep than Jürgens' models. Consequently, we propose the implementation of a second generation algorithm instead of CMB. Synchronous barrier-based second generation algorithms are more robust in case of small lookaheads. Instead of lookahead time creep, barrier-based algorithms progress at each barrier synchronization to the next event [Fuj00a]. Thereby, barrier-based algorithms avoid lookahead creep. Therefore, we propose the more promising synchronous barrier-based synchronization.

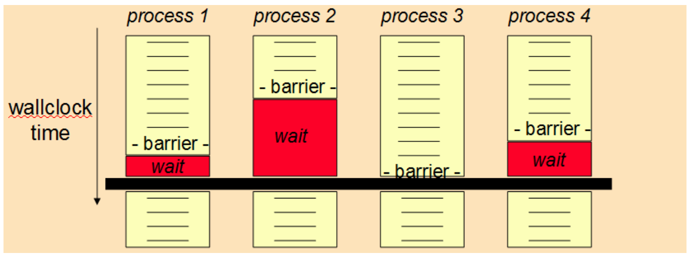

Figure 5.1: Barrier synchronization from [Fuj00a]

Concluding remarks

The optimal synchronization algorithm does not depend on the formalism. It depends on the models and their capability. No algorithm exists that solves all scenarios perfectly. We focused on conservative synchronization approaches as they perform better than optimistic approaches on QPNs as shown in [Jür97]. Further, this section points out characteristics of our models which depend to a certain degree on the formalism. In our context, we expect models with small cycles containing poor lookahead characteristics. Hence, we argued for a barrier-based synchronization for QPN.

To reach speedups, especially the lookahead characteristics play an important role. The decomposition into LPs influences lookahead calculation as it can preserve or ruin lookahead characteristics. In the following we describe possible decomposition strategies for QPNs before we go into detail of lookahead calculation.

5.2.2 Model Decomposition

Event level parallelism requires the simulation model to be partitioned into disjoint LPs. In our case, the decomposition should support the goal of "as fast as possible" simulation. This requires to keep in mind some general ideas about parallel simulation. Parallel simulation can be fast if the decomposition enables the utilization of the available cores and minimizes synchronization overhead.

Decomposition should provide high lookahead between LPs. This helps to minimize the count of synchronization operations. Further, decomposition should consider granularity. A fine-grained decomposition supports the utilization of the model inherent parallelism. However, it introduces synchronization even if the local causality constraint requires sequential execution. Sequential passages of the model should reside in one LP to avoid non-necessary synchronization operations. Thereby multiple events can be scheduled in one LP before next synchronization. This reduces synchronization effort as a look on the extreme case shows: A model decomposed into one LP requires no synchronization but cannot benefit from parallelism. The challenge is to decompose the net fine-grained, where we can employ parallelism, and coarse-grained where we cannot employ parallelism.

In general, the model can be split spatially and temporally. For QPNs, we cannot anticipate future system states and we would have to execute temporal LPs in sequential order. Hence, a temporal decomposition would not have any positive effect on performance. Furthermore, temporal decomposition does not scale with the size of the net as spatial decomposition does [Fuj00b]. Spatial decomposition promises a higher potential for parallelism as stated in [NR91, CF93] and [NK01]. The spatial decomposition enables each processor to load only the relevant subparts of the model and thereby to save memory costs.

One partition method is to transform each place and each transition into a LP. The disadvantage of this simple approach is that it adds fine-grained synchronization to the simulation. More sophisticated decomposition rules may help to minimize synchronization overhead.

In some cases, we can derive hints from structural properties of the modeling formalism. Jürgens [Jür97] states the hierarchies in HQPNs to be a good indicator for decomposition. Each hierarchical place forms the net region dedicated to one LP. The heuristic was tested on artificial models and has not been evaluated on real world models. This complicates the assessment of the general qualification of this approach.

Jürgens' approach [Jür97] requires hierarchical structures that have to be set by the user during model creation. If the model has no native model structure an expert may partition the net by hand. From our point of view, human interaction to speed up simulation should be minimized for multiple reasons. Besides the fact that we expect human interaction not to be available in general, we see this as a possible source of errors. Furthermore, human interaction is more expensive than computational effort.

In order not to rely on human interaction we require an automatable approach. For QPNs no decomposition approach has been proposed. Hence, we review PN-strategies. PN decomposition rules have been proposed in [CF93] and [NK01]. They start by the definition of minimum regions followed by the application of merging rules on these minimum regions to form LPs.

Normally, we reach a point where it is not reasonable to merge further as this would destroy potential parallelism. In most cases, we have more LPs than processors. Instead of running multiple LPs on one processor in parallel, multiple LPs can be packed into one multiprocess. This multiprocess processes all safe events of its LPs sequentially. Then it synchronizes with the other multiprocesses. This idea reduces synchronization overhead and has been applied for example in [PR94] and [LSCD04]. The multiprocess concept is not a replacement for decomposition, it is a supplement.

QPN decomposition is specified in three parts. In Section 5.2.2.1 we explain Chiola and Ferscha's [CF93] concept of minimum regions. Afterwards we discuss the application of their PN merging rules to QPNs in Section 5.2.2.2. Finally, we propose additional merging rules in Section 5.2.2.3.

5.2.2.1 Minimum Regions

Chiola and Ferscha [CF93] focused on minimization of network communication of distributed Stochastic Petri Nets (SPNs) simulation. For this purpose, they optimized the decomposition of SPNs. They define a *conflict set* as all transitions that share an input place. Together with their input place those transitions form a minimum region. According to their argumentation, LPs can combine but not subdivide minimum regions. Their key idea is that arcs from places to transitions effect the enabling and firing of transitions. In contrast, arcs from transitions to places do not affect firing decisions. Formed into a decomposition rule, LPs should be cut at arcs from transitions to places. This has the positive effect that in case of transition firing the random choice of the next transition to fire depends only on one LP.

In distributed environments, like in [CF93], the conflict resolution is very expensive. Nevertheless, even local environments can benefit by avoiding non-necessary context switches. Therefore, the concept of minimum regions suits well for parallel QPN simulation.

5.2.2.2 Merging Rules from Petri Nets

By now we have minimum regions. The next question is how to form optimal regions from minimum regions. Chiola and Ferscha [CF93] propose a set of heuristics which focus on the reduction of network communication. Their merging rules work with PN characteristics which we have already defined in Section 2.1.2. Moreover the following notes support the comprehension of Chiola's and Ferschas' rules. SPNs order the firing at one time using priorities. In this connection, π_i denotes the priority of a transition t_i. The weight $w(t,p)$ for a transition t and a place p terms the service time of the transition when it picks tokens from that place. A *P-invariant* ensures conditions for a set of places P in every reachable state of the net. At optimistic synchronization algorithms, a LP may progress ahead of others. Then, messages may arrive too late which causes rollbacks. These late arriving messages are called straggler messages. A more detailed description has been set in Section 2.2.3.2.

Going back from introductory notes to the merging rules of Chiola and Ferscha [CF93]:

Rule 1 Mutually exclusive (ME) transitions go into one LP since they bear no potential parallelism. Two transitions $t_i, t_j \in T$ are said to be mutually exclusive, denoted by $(t_i \; ME \; t_j)$, if and exclusively if they cannot be simultaneously enabled in any reachable marking. A sufficient condition for $(t_i ME t_j)$ is that the number of tokens in a P-invariant out of which t_i and t_j share places prohibits a simultaneous enabling. Another sufficient condition for $(t_i ME t_j)$ is that $\exists t_k : \pi_k > \pi_j$ such that

- $\forall p \in$ input arcs of $t_k, p \in$ input arcs $t_i \cup t_j$
- $w(t_k, p) \leq \max(w(t_i, p), w(t_j, p))$.

Rule 2 Endogenous simulation speed is balanced to prevent from rollbacks, i.e. the probability of receiving straggler messages is reduced by balanced virtual time increases in all LPs

Rule 3 LPs with high message traffic intensity are merged to safe message transfer costs

Rule 4 Persistent net parts and free choice conflicts are always placed to the output border to allow sending out messages ahead of the LVT (lookahead) without possibility of rollback, (i.e. sending ahead messages that will be inevitably generated by future events – unless local rollback occurs)

Rule 5 Transitions having a single input place can also be connected to the input border, since the enabling test can be avoided for these transitions (firing can be scheduled immediately upon receipt of the positive token message without additional overhead)

The next question to solve is whether we can apply these merging rules for QPNs. We provide a review for each rule.

Review Rule 1 `Rule 1` is applicable to QPN and should be applied. The example models shipped with QPME do not have any mutual exclusive transitions. Furthermore, the application of this rule requires a complex structural analysis to find mutual exclusive parts of the net.

Review Rule 2 Rule 2 is about optimizations to minimize inter-LP communication for an optimistic synchronization protocol. Conservative simulation cannot benefit from this rule.

Review Rule 3 Rule 3 proposes to merge at transitions with high firing frequency which is recommendable for parallel QPN simulation. Relevant transitions can be derived comparing service time (distributions) of all transitions. This can be applied to a subset of the QPN formalism. The approach requires timed transitions with different service times. However, the QPN formalism enables to build models solely with immediate transition type which means zero service time at all transitions. Under these conditions it is more difficult to determine transitions with high firing frequency.

Review Rule 4 Rule 4 is an optimization rule to reduce rollbacks for optimistic synchronization. The principle can be ported to QPNs, but has no impact on conservative synchronization.

Review Rule 5 Rule 5 describes an LP-internal improvement which has no impact on decomposition. The rule assumes transitions which need only one token to be enabled. However, the QPN formalism enables more complex transition modes. Transition modes may require more than one token to be enabled. Moreover, transition modes may require special token color combinations to fire. Summing up, the application of Rule 5 has many restrictions on QPN formalism.

5.2.2.3 New Merging Rules

The previous section showed that certain merging rules proposed for PNs can be applied to QPNs. In this section we suggest additional merging rules in order to improve the decomposition. Therefore, we search for representative structures. Structures we analyzed are non-branching lanes and branching transitions. We explain factors that indicate whether merging or non-merging of LPs is is appropriate.

Non-Branching Lanes

Something that we did not see published in the context of net decomposition and merging is the idea of lanes. For convenience, we define places and transitions to be nodes.

Definition 9 (Lane)**.**
A lane is an interconnected list of nodes with only one input and one output arc each.

Intuitively one does not expect parallelism within a lane. Consequently nodes respectively LPs in a lane would be a candidate for merging.

In case of QPN simulation, this intuition is a false friend. Even lanes offer inherent parallelism. This is for example depicted in Figure 5.2. The figure shows a lane from place `p1` to queueing place `p4` where `t1` and `t3` can fire concurrently. When we expect the service times of `p4` to be less or equal to `p2` then even the queueing places can be processed in parallel. The optimal decomposition in this scenario is in two LPs, LP_1 (`p1` ... `t2`) and LP_2 (`p3` ... `p4`).

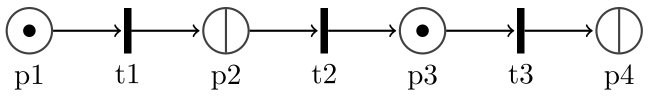

Figure 5.2: A lane of places and transitions

Despite of the negative example, the idea of merging LPs in a lane is not totally wrong. The question is when to merge nodes respectively LPs into lanes. If each token leaves the LP before a new token arrives then the LP has no inherent parallelism. The arrival of new tokens can be appraised by the help of lookahead of the predecessor. The processing speed of a LP is the sum of service times of nodes in the lane. The combination of predecessor lookahead and the sum of LP service times determines the optimal length of the lane. The corresponding rule is:

Rule 6 Merge LPs in a lane as long as the cumulated service times of the lane do not exceed the lookahead of the preceding LPs.

Even before simulation we can estimate the lowest token frequency of the predecessor. We know the service time distributions of the predecessor. However, the estimation before simulation can be less precise than during simulation. `Rule 6` is a lower bound on the length of the lane. If lookahead prediction deviates strongly from what we expect from predecessor service time distribution, the rule can be modified to merge more elements.

Instead of the lookahead we can set a statistical border of cases where the service time may less. Thereby, we increase the border and can introduce more elements into the lane. We use the service time quantiles of the predecessor. The quantile is a point taken from the cumulative distribution function of the service time distribution. The variable q denotes 100 percent of the observations. The k^{th} q-quantile is the value x so that at most k/q will be less than x and at least $1 - k/q$ are higher than x. The $0,1$-quantile x means that at most $10/1$ percent of tokens arrive faster than x. The lane processes incoming tokens before a new token arrives for at least 90 percent of all cases.

Lookahead and service times determine when to merge LPs within lanes. A corollary of `Rule 6` is that nodes with zero lookahead, like places and immediate transitions, should always be merged into lanes.

Branching Transitions

Lanes do not possess intersections. Now we take a look at *branching transitions*. We name a transition a branching transition if it has more than one outgoing arc. We deal with the question whether to merge at branching transitions or not. Therefore, we will refine branching transitions into *distributor transitions* and *choice transitions*. We start introducing distributor transitions:

Definition 10 (Distributor Transition).
We name a transition a distributor transition if it is a branching transition where all modes fire on all output-arcs.

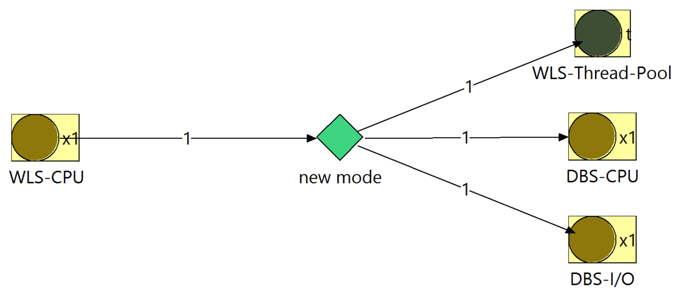

Figure 5.3: Distributor transition

A distributor transition creates parallelism as it fires on all output arcs in parallel. The transition in Figure 5.3 creates three parallel actions after firing in `new mode`. All subsequent queueing places `WLS-Thread-Pool`, `DBS-CPU` and `DBS-I/O` simultaneously receive a token. All subsequent places can process in parallel. The example illustrates: We should avoid merging distributor transitions and succeeding LPs. We form this perception into a rule:

Rule 7 LPs with distributor transition at the output border should not be merged with the LPs connected to the distributor transition.

There also exist branching transitions which are suitable for merging. In contrast to distributor transitions, choice transitions are promising merging points. We define choice transitions as follows:

Definition 11 (Choice Transition).
We name a transition a choice transition if it is a branching transition and each mode fires at only one output arc.

The example in Figure 5.4 shows a choice transition with three modes (`mode1, mode2, mode3`). The firing of the transition sends a token to only one of the out places (`WLS-Thread-Pool`, `DBS-CPU` and `DBS-I/O`). The choice for the mode is mutually exclusive. Regardless of the mutual exclusive mode choice, the out places can execute tokens concurrently. This happens when the transition fires all modes before either `WLS-Thread-Pool` or `DBS-CPU` or `DBS-I/O` finishes service. A choice transition creates parallelism when the firing frequency is high. The scenario has no parallelism when -for example- the service time of the in place `WLS-CPU` is higher than the service of the out places.

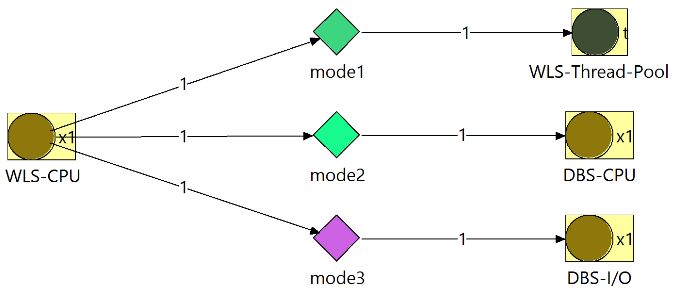

Figure 5.4: Choice transition

For the most part, a choice transition indicates mutual exclusive parts of a net. If so, one should merge succeeding LPs at the choice transition. Especially, when the firing frequency of the choice transition is low. A high firing frequency of the choice transition raises the chance to experience parallelism. The corresponding rule reads as follows:

Rule 8 LPs with choice transition at the output border should be merged with the LPs connected to the distributor transition, except for transitions with high firing frequency.

Subsequent to the decomposition and merging rules the next section explains how to calculate the lookahead.

5.2.3 Lookahead Calculation

This section explains lookahead calculations for QPN. The previous Section 5.2.2 proposed decomposition into multiple element LPs. Hence, this section starts by an explanation on how the lookahead for an LPs can be composed from the lookahead of places and transitions. Afterwards we explain the lookahead calculation for places, queueing places, immediate transitions and timed transitions.

5.2.3.1 Logical Processes

Lookahead is the time a LP can look ahead from the current simulation time and process independently of other LP without violation of the local causality constraint. Lookahead is a guarantee that the LP will not receive tokens with smaller timestamps. The more precise we can predict the next token emittance of predecessors, the longer the time span to safely process events. Hence, the objective is to predict a high lookahead. An optimal lookahead calculation predicts exactly the arrival of the next token that affects simulation.

To calculate lookahead for LPs we define *in places* and *out transitions*. Places connected to another LP via incoming arcs are in places. Transitions connected to another LP are out transitions. New tokens arrive at in places. Tokens leave the LP via out transitions.

To predict the emittance of the next token we review the paths throughout the LP. Therefore, the number of places and transitions on such paths is irrelevant. Instead, we apply node weights like Dijkstra's shortest path algorithm [Dij59]. The weights of places and transitions are equal to the lookahead of the corresponding net elements. The length of a path is the sum of its weights.

The lookahead calculation is different if the places of the LP hold any token or if they do not. If the LP does not hold any token we name it empty. In the empty case the LP can only emit tokens if it first receives a token from another LP. The lookahead is the shortest path from an in place to an out transition. Non-empty LPs can emit the tokens they already hold plus new arriving tokens. Hence, we additionally have to take the shortest paths from tokens to out transitions into account.

To combine empty and non-empty scenarios we walk backwards starting form out transitions. The end of a path is either an in places or a place containing a token. A valid estimation for the combined (LP) lookahead is the shortest path backwards from an out transition to a place containing a token or an in place. Hereby, the shortest path forms a valid solution that is optimal in many scenarios. In the following we will analyze different net structures to exemplify when and how the prediction can be improved.

QPN formalism enables multiple transition modes. At first we consider transition modes that can be enabled by one token. For multiple modes at one transition an anticipation of future choice of transition modes can further improve calculation. If we have no anticipation of mode choice the shortest path approach is optimal. Otherwise, we can improve the conservative shortest path approach to a shortest possible path approach. A previous scheduling of branching decisions can provide this additional information. This previous scheduling cannot be performed by every simulator. It has to be considered how the simulator performs the random choice of next transition mode. If the simulator, like SimQPN, considers the modes currently enabled for mode choice a previous scheduling is impossible. Nevertheless, we run through a short example. We examine Figure 5.5 to explain how better lookahead may be calculated. We assume that the next three times the transition is enabled `mode 1` is chosen. Under these conditions an improved lookahead for `Client` can be calculated. The shortest possible path is the one for the fourth token arriving at `WLS-CPU`. To consider the fourth arriving token instead of the first constitutes the improvement.

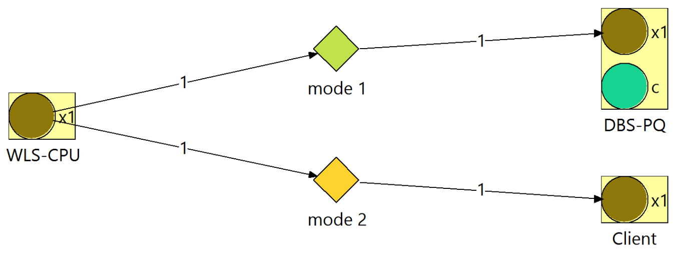

Figure 5.5: Transition modes with one input token

The QPN formalism allows transition modes which require multiple tokens. Again we can improve the conservative shortest path approach to a shortest possible path approach. For transitions that require multiple tokens to fire we determine the minimum subset of incoming paths to enable the transition. For lookahead calculation we follow the path with the maximum length from this minimum subset.

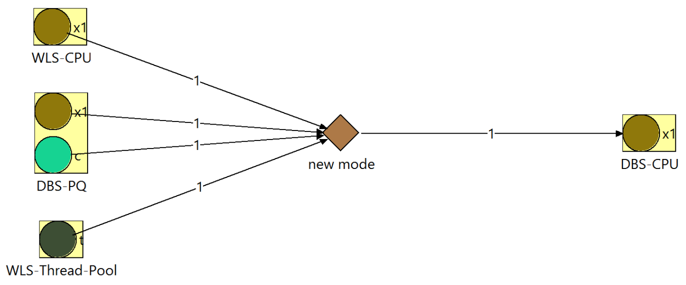

Figure 5.6: Transition mode with multiple input token

For visualization, we consider Figure 5.6. The firing depends on tokens in multiple places and/or multiple token colors. The transition with `new mode` needs tokens in `WLS CPU`, `DBS-PQ` and `DBS-Conn-Pool` to be enabled. Further, `DBS-PQ` has to provide both colors. The enabling subset consists of four paths. The subset contains the path for a brown token in `WLS-CPU`, paths for a brown and a turquoise token in `DBS-PQ` and a backward path from `WLS-Thread-Pool`. The path with the maximum length is considered for lookahead calculation. The lookahead calculation considers the last token that is needed to enable the transition.

The shortest possible path procedure is as follows: We process backwards through the net and follow the same procedure at every transition. We follow all transition modes. If the transition mode has one incoming arc we follow this arc. For more incoming arcs, we pursue all incoming paths. Either the mode is enabled by multiple paths and we consider the path of maximum length or we choose the shortest path.

Moreover the consideration of branching decisions may improve lookahead estimation. This happens when tokens take other directions than the lookahead calculation expected. Until now, we expected that each token branches directly towards the actual place in focus. If the token branches in another direction, places receive the next token later then predicted. Branching of token streams may occur when a place on the previous path is source for multiple transitions. Usually, we cannot predict which transition fires next and we assume each to fire next. Remedy for the knowledge gap can be achieved by a precalculation of branching decisions. By that, the prediction for the next token arrival can be improved. We assume a place $p1$ which is connected to the transitions $t1$ and $t2$. Both transitions enable simultaneously. The knowledge that $t1$ is chosen three times before $t2$ is chosen allows to consider the time for the fourth token for $t2$ which improves lookahead prediction.

Finally another option for such branchings exists which is a transition with multiple modes where none of the modes is connected to all out places. For those transitions some out places may starve. A prescheduling of the transition mode choice may improve the prediction. An example has already been depicted in Figure 5.5. Please note that we only require a prescheduling for the described transition type.

The prediction and precalculation of transition modes at branching transition is only feasible if the choice of branching decision is independent of the current state. SimQPN chooses the transition mode dependent on the currently enabled transition modes. In SimQPN, as well as for transition modes, the scheduling of the next transition to fire depends on the current state. In general, the prescheduling approach is limited to simulators where the choice is independent of current state.

Summing up, this section proposed a baseline technique to combine lookahead followed by improvements which enable more precise token arrival time prediction. These improvements enable an increased lookahead which is a essential for successful parallel execution. Until now, we assumed that places and transitions may provide lookahead. The next sections explain difficulties and solutions to lookahead generation for all net elements.

5.2.3.2 Places

The question about lookahead for places can easily be answered. The service times for places without queues is always zero. Hence, the lookahead is zero too. In contrast, service times and lookahead for queueing places depend on the employed scheduling strategy and service time distribution.

5.2.3.3 Queueing Places

The lookahead for a queueing place depends on the queueing strategy. QPME supports the following scheduling strategies for queues which we already described in Section 2.1.1.

- First-Come-First-Served (FCFS)
- Infinite Server (IS)
- Processor Sharing (PS)
- Random Scheduling (RANDOM)
- Priority Scheduling (PRIO)

All these queueing strategies have in common that the simulator creates service times according to a distribution. A sequential simulator has no benefit of choosing the service time and branching destination any sooner than required. In contrast, parallel execution can benefit form an early choice. The key idea is to select a job's service time and branching destination before it arrives at the queue. This allows an improvement on lookahead prediction.

For each incoming transition random service time values can be sampled before the next job arrives. The in advance sampled service times can be stored in a future list as introduced by Nicol [Nic88].

The basic functionality of a future list is that every job that arrives peeks the next service time from the future list that has not been peeked so far. The finishing of a service removes the head of future list. Next we explain how the different scheduling strategies can benefit from a future list based on Wagner and Lazowska [WL89].

First Come First Served

A First-Come-First-Served (FCFS) queue processes jobs in incoming order. Jobs that arrive in a FCFS queue cannot be passed by jobs that will arrive later. If the queue has already jobs enqueued, the service time of jobs which has arrived first equals the head of future list. Otherwise, if the queue has no jobs enqueued the service time of the next event is the head of future list too.

We assume queue Q to be at time t and the head of future list to have service time s. Then Q will send no jobs before time $t + s$. For a FCFS queue the lookahead is the head of the future list.

$$lookahead = \text{head of future list}$$

Infinite Server

Infinite Server (IS) delays all incoming jobs according to a random distribution. An IS (also called delay resource) delays each job according to its service time. The lookahead of a IS-queue is the minimum service time of all possibly arriving jobs. If no precautions on the token arrival are taken into account lookahead equals the minimum of the service time distribution. In performance models, we typically use exponential distributions where the minimum is zero. Hardware does not react in zero time but distributions commonly have a lower bound of zero.

Wagner and Lazowska [WL89] proposed a simple approach to improve the lookahead prediction. This approach requires to set up an upper bound on the jobs that can be processed in parallel. For a queue that has a limit of jobs it can hold the lookahead can be determined with a previously calculated sample of service times of same size. The minimum of these previously sampled service times is a lower bound for the next token emittance. Often the queue-model does not provide limitations on the number of jobs a queue can hold. Instead, the upper bound on arriving tokens can be used to create a limit. The bound problem description is modified from maximum acceptance to maximum arrival. The new limitation does not derive from the queue but from the number of tokens in the network model. Models that ensure a non-rising overall token population ensure an upper bound on token arrival. The global number of tokens is an upper bound on token arrival for each queue. More accurate bounds may be derived by the help of structural analysis.

Provided by an upper bound of n tokens we can improve lookahead estimation. We use the set of service times $s_1, \dots, s_n$ of jobs already enqueued or next to be enqueued:

$$\textit{lookahead} = \min\{s_i | 1 \leq i \leq n\}$$

The set $s_1, \dots, s_n$ is equal to the first n jobs of the future list.

Processor Sharing

Processor Sharing (PS) is an idealization of RR scheduling with infinitesimal time slices. When n jobs are enqueued the jobs are served with n^{th} share of the full processor power. The determination when the next event finishes is difficult even if we know current time is t and the job i with the shortest residual time r_i. Given that no token arrives before the currently enqueued jobs have been processed, the lookahead would be $t + r_i$. This job may finish later than $t + r_i$ if a new job arrives that steels time slices. Incoming jobs slow down the jobs already enqueued. Nonetheless, a job may finish before $t + r_i$. A newly arriving job may pass the enqueued jobs. The worst case that lookahead calculation has to consider is that a new job arrives that has a smaller service time than the smallest residual time of enqueued tokens.

Wagner and Lazowska [WL89] proposed the following lookahead calculation for PS queues. The lookahead is the minimum of the enqueued service residual times and the minimum service time of future jobs. We assume n customers with residual times $r_1, \dots, r_n$ and a the minimum service time of the next incoming service. Then the lookahead is

$$\textit{lookahead} = \min\{\{nr_i | 1 \leq i \leq n\} \cup (n+1)a\}$$

The first bracket of the equation $\{nr_i | 1 \leq i \leq n\}$ describes the minimum residual time of enqueued services and $(n + l)a$ the minimum processing time of future jobs.

The idea of Wagner and Lazowska requires that the queue either has a maximum capacity or an upper bound on tokens that may arrive concurrently. Otherwise an unbounded future list has to be reviewed to determine a.

In performance modeling, PS queues are used to model CPUs. In most models the CPU sets no limit on customer population. Furthermore, CPUs may finish small work packages in infinitesimal time slices which results in a lower bound on service time distribution set to zero. Then a is zero and consequentially, the lookahead is zero too. Normally, CPUs set no limit on concurrently arriving tokens. Similar to IS we can estimate the maximum token count that can arrive at the queue. This upper bound ub determines the number of future list jobs used for determination of a. The minimal service time a is the minimum of the first ub future list service times.

Random Scheduling

The RANDOM scheduling strategy randomly chooses the jobs from waiting line. Hence, this queueing strategy is disadvantageous for lookahead prediction. Nevertheless, we apply the same approach as in IS. We estimate an upper bound n on the tokens that may arrive. We use the set of service times $s_1, \ldots, s_n$ of tokens already enqueued or next to be enqueued:

$$lookahead = \min\{s_i | 1 \leq i \leq n\}$$

The set $s_1, \ldots, s_n$ is equal to the first n jobs of the future list. If we cannot determine an upper bound on the incoming tokens the lookahead is the minimum of the service time distribution.

Priority Scheduling

For Priority Scheduling (PRIO) it is important to define how the queue reacts on incoming jobs with same priority. We assume FCFS ordering if two (or more) jobs arrive with the same priority. For this case, Wagner and Lazowska's [WL89] proposed an array based strategy.

Lookahead calculation for PRIO queues considers the enqueued job with the highest priority, which is the job actually being scheduled. This job can only be passed by incoming jobs with higher priority. Hence, the lookahead is the minimum of the (not yet fully processed) service time and the heads from future list with priority less or equal to the actual job. The implementation requires a priority list for every priority level. Wagner and Lazowska name the set of lists a future array. Let *prio* be the highest priority level of any job currently at the queue. If no job is enqueued *prio* is set to the minimum of all priorities. The lookahead for PRIO queues is set to:

$$lookahead = \min(\{\text{head of future list} | \text{priority level future list} \geq prio\})$$

In summary, this section discussed lookahead for multiple queueing strategies. The lookahead for queues can be transfered to corresponding queueing place. Next, we explain lookahead for further net elements.

5.2.3.4 Immediate Transitions

Immediate transitions have no service times. They fire in zero time. Hence, the lookahead of an immediate transition is zero. In contrast, timed transitions have service times and lookahead. The lookahead for immediate transitions is:

$$lookahead = 0$$

5.2.3.5 Timed Transitions

Lookahead calculation for timed transitions behaves like the calculation for FCFS queues. The queue processes jobs in incoming order. The service time of timed transitions depends on a random distribution. Analog to the FCFS-queues we introduce a future list that holds future service times. The lookahead is:

$$lookahead = \text{head of future list}$$

5.3 Functional level

Functional parallelization means the extraction of helper functions of the simulation algorithm and their execution parallel to the simulation algorithm. Possible helper functions are random number generation and statistic processing for example parallel batching of intervals for the Batch/Means algorithm. Further common helpers are I/O management and event management.

In addition to the classical helpers at the sequential algorithm, the parallel algorithm introduces new helper functions. The lookahead calculation requires future lists which have to be filled in an effective way. One option is to perform an initial filling of all future lists and then to perform a refill if an object has been polled. This creates a (potentially high) delay at the beginning of the simulation. We see two main strategies avoiding this disadvantage.

Lazy Filling means to create jobs of the future list only at the time lookahead calculation requires them.

Idle Filling means to utilize CPU cycles which cannot be utilized through regular simulation and would otherwise remain unused.

It is obvious that these strategies compete. The advantage of idle filling is that it may utilize idle CPU times which are common in conservative simulation. The advantage of lazy filling is that it creates only jobs that are necessary compared to idle filling. Tests on the simulator have to show whether the functional parallelism of idle filling performs better.

From the excursus of future lists back to general remarks on functional parallelism. Before extracting helper functions one has to test their relevance for overall performance. A helper function should be chosen for extraction and parallelization if it has a remarkable impact on performance. Otherwise parallelization is not likely to offer speedup. Even a negative impact on overall performance is possible. To implement functional parallelism a performance relevant function has to exist and has to be identified. It is not guaranteed that such an appropriate function exists. Notice that badly written helpers suit well for functional helpers. Before parallelization we recommend to evaluate if this function can be tuned without parallelism.

Functional parallelism enables speedup by a constant factor but does not scale with the size of the model. Hence, functional parallelism is not qualified to solve large scale problems.

5.4 Summary

Since this chapter contains some of the most important parts of the thesis, a short summary will be provided before continuing with the next part. We have presented an overview on parallelization approaches for QPNs simulation on a theoretical basis. We discussed the possibilities for the three parallelization levels:

- application level
- event level
- functional level

The application level parallelization executes multiple independent runs in parallel. This requires an experiment design that allows the decomposition of the problem into multiple runs. Meeting this requirement, the emerging runs can be simulated in parallel.

The event level parallelization is independent of experimental design. For event level parallelization the characteristics of the QPN models have to be considered. We identified parts of the QPN formalism that complicate speedup in parallel simulation. We discussed different synchronization algorithms and argued that a barrier-based conservative synchronization fits best for our models. Before parallel execution, the QPN is decomposed into disjunct LPs. We reviewed decomposition rules from PNs and proposed additional rules. Furthermore, we discussed advanced techniques for lookahead calculation for QPNs. Therefore, we transfered concepts from QNs.

The functional level parallelization requires an analysis of the algorithm and its implementation. Functional helpers can be extracted and be computed parallel to execution. In contrast to the other levels, the speedup for functional parallelism does not scale with the model size. The next chapter is about the implementation of what this chapter has discussed on a theoretical basis.

6. Implementation

The previous chapter dealt with parallel QPN simulation on a theoretical basis. The current chapter describes the implementation of these concepts in QPME.

In Section 6.1, we explain the adjustment of SimQPN towards a parallel simulation engine before we introduce parallelism. The adaptation yields a new component architecture which Section 6.2 depicts. Section 6.3 is about the application level implementation. Then Section 6.4 describes parallelization on event level. Finally, Section 6.5 is about the potential for functional parallelism within SimQPN.

6.1 Adaptation of SimQPN

SimQPN is a grown software system which we restructure to improve modularity and state encapsulation. The refactoring creates adaptation points for the new simulation engine. Moreover, the refactoring allows to reuse existing implementations and to organize the implementation to cope with its complexity.

The established SimQPN version unifies the core simulation engine and three analysis approaches in one class. We separate analysis approaches and the core simulation engine into different classes. The new implementation accesses the extracted classes via newly introduced interfaces. In the new version, the simulation controller chooses via strategy pattern which analysis approach to use. The analysis approaches choose via strategy pattern which simulation engine to use. These modifications enable a simple integration of new core simulation engines and new analysis approaches. Both modifications are required as insertion points for the new parallel simulation approaches.

The main simulator class in the established version contains QPN entities in a non-encapsulated manner. Parallel and sequential simulators have to access the same QPN entities to perform simulation. To avoid code duplicates we encapsulate the entities representing the QPN in a `Net` class available to sequential and parallel core simulation engines. This refactoring also improves code readability as it reduces the amount of code within the core simulation engine class.

The established version has a queue class which unifies the implementation for all queueing strategies. We make the queue class abstract and extract the specific queueing strategies into subclasses. This object oriented hierarchical organization simplifies the integration of additional queueing strategies into SimQPN. Furthermore, the implementation for lookahead calculation can be attached per corresponding queueing strategy class. Figure 6.1

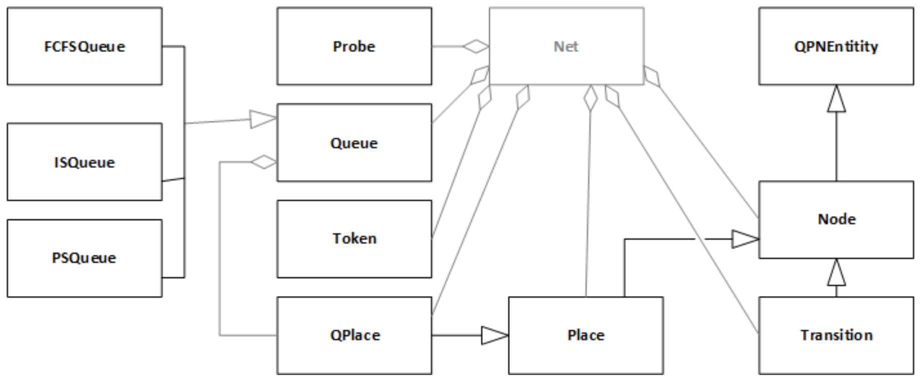

Figure 6.1: QPN Class Diagram

depicts the classes that represent a QPN within SimQPN. This class diagram includes previously mentioned refactoring of queue and the newly introduced net class.

Parallel simulation requires a component that loads the model as well as sequential simulation. The net loading of the established SimQPN version is nested in the sequential core engine. We extract the QPN loading into a newly created component to keep it reusable for new simulation engine. Moreover, we extract parts related to loading such as net validation and the flattening function for HQPNs.

We do not only refactor but also speedup the existing code. For the multiple replication case the established SimQPN loads the simulation model multiple times from file. We introduce a net copy function which avoids costly loading of the model from file. This performance improvement benefits from our previous encapsulation of QPN entities in the `Net` class. We implement the net copy as a deep copy function for all net elements. The interconnection between net elements are set in a final step. We test on a model with less than 70 transitions and saved about half a second per run. This copy function is not straightforward related to parallelism but speeds up all approaches that use multiple replications. The result of SimQPN adaptation is a new component architecture which is described in the next section.

6.2 Component Architecture

The adaptation at SimQPN divides the existing code into a set of components. Figure 6.2 shows a component diagram of the new design. The `SimulationController` component manges multiple services to create a composed simulation service that simulates an XML specified model. The interface towards QPME has not changed compared to established SimQPN. The `SimulationController` component has three sockets. It requires a `Net-LoaderService`, an `AnalyzerService` and a `drawProgressService` to provide its service. In the following we explain the services provided by the components.

Net Loader The `Net Loader` loads a `Net` from an XML file. To fulfill its service the `Net Loader` uses a `HQPN Flattener` and a `NetValidator`.

- `HQPN Flattener`
  The `HQPN Flattener` transforms Hierarchical QPNs to flat QPNs by expanding subnet places. It takes a a (hierarchical) net description as input and produces a (flattened) net description. The `Net` is created with the modified description.

- `Net Validator`
  The `Net Validator` validates if the net description within the file is correct.

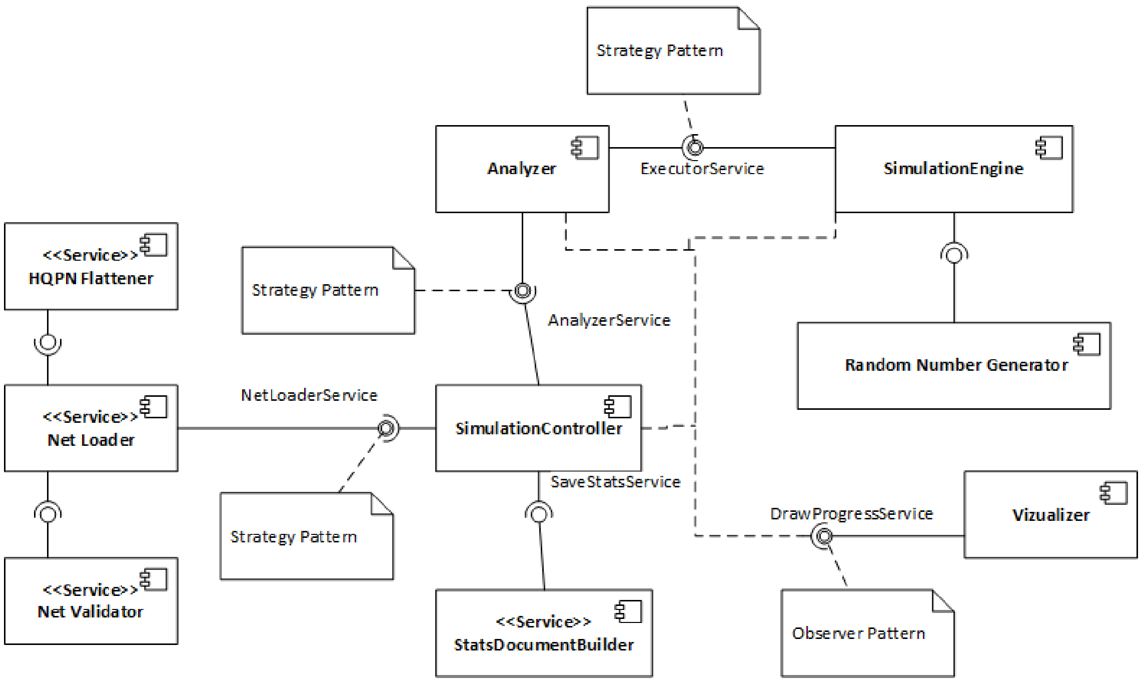

Figure 6.2: Component diagram of the new SimQPN architecture

It requires the net description from XML file and returns a boolean indicating whether the description is correct or not.

Analyzer The `Analyzer` analyzes a `Net` with either Batch/Means, Replication/Deletion or Method of Welch. Its parameters are a `Net` and a configuration which specifies parameterizations for analysis approach and core simulation. To perform the core simulation loop it uses the `SimulationEngine` component. After the simulation run(s), the component returns the statistics of collected analysis results in an array.

SimulationEngine The `SimulationEngine` requires a `Net` plus a configuration which specifies various parameterizations such as the length of ramp up and total run length. It produces a `Net` enriched with statistics of the simulation run. For the simulation run it requires a `Random Number Generator` component.

- `Random Number Generator`
  The `Random Number Generator` component returns random numbers and distributions needed for simulation. SimQPN utilizes a library from the Colt project[1] developed at European Organization for Nuclear Research (CERN).

StatsDocumentBuilder The `StatsDocumentBuilder` component persists the statistics in a *.simqpn file. It requires the statistics array that has been created by the `AnalyzerService` and is provided by the `SimulationController`.

Visualizer The `Visualizer` component shows the progress of the simulation. `SimulationController`, `Analyzer` and `SimulationEngine` notify their progress to the chosen visualization method. The component contains a textual and a graphical visualization. The graphical representation is embedded in QPME.

[1] `http://acs.lbl.gov/software/colt/`

6.3 Application Level

Parallelization on application level does not need a parallel simulation engine. The sequential simulation engine can be started multiple times in parallel. Nevertheless, certain technical adjustments were performed. The parallelization accesses Java implementations from the `java.util.concurrent` library. All mentioned classes and functionalities are available in this library. We extend the sequential executor to support the `Callable` interface. The new `call` method returns a reference to a `Net` which has been enriched with statistics during simulation. The use of the common `Callable` interface suits for parallel simulation and further enables an easy adaptation towards distributed simulation.

`Callable` objects can be passed to an `ExecutorService` to be executed. For execution of the callable objects we use a threadpool with fixed number of threads. The number of threads is equal to number of available processors. The threadpool creation is performed by `Executors.newFixedThreadPool` from the `java.util.concurrent` package.

Our ambition was to ensure the results of parallel Replication/Deletion equal to the sequential execution. To reach this, the chosen random distributions have to be equal. The random distributions are created during initialization due to the initial random seed. The order of creation determines the pseudo random values of the distribution. A parallel initialization would mix the creation order and we would receive different results. For this reason we did not parallelize the initialization. The time for initialization is a very small fraction on overall simulation time. Hence, it is justifiable to keep the initialization sequential. Algorithm 2 depicts our implementation of the parallel Replication/Deletion algorithm.

Algorithm 2 Parallel Replication/Deletion

```
 1: function RUN(numRuns)
 2:     List<Callable<Net>> listOfModelsToSimulate;
 3:     List<Future<Net>> listOfSimulatedNets;
 4:     for numRuns do
 5:         listOfModelsToSimulate ← callable refererence for new run;
 6:     end for
 7:     ExecutorService executor ← new ThreadPoolExecutor(numCores)
 8:     listOfSimulatedNets ← executor.invokeAll(listOfModelsToSimulate)
 9:     for future ∈ listSimulatedNets do
10:         simulated net ← future.get()
11:         finalStatistics ← statistics of simulated net;
12:     end for
13: end function
```

The next section provides insights into event level implementations.

6.4 Event Level

Unlike application level parallelism, the event level parallelization requires a parallel execution of the core simulation loop. For this purpose, our implementation performs a spatial decomposition of the net. The resulting LPs communicate their progress using a barrier-based synchronization algorithm. This algorithm constantly alternates between event processing and updating of the times safe to process. The execution of these two phases is separated by barrier operations. The time safe to process is the upper bound of time the LP can process to before it enters the barrier. A barrier synchronization means any LP must stop at this point and cannot proceed until all LPs have reached the barrier.

LPs get blocked when they enter the barrier. The barrier waits until all LP-threads have entered before a barrier release unblocks all LPs. During each barrier synchronization we test for termination condition. LPs stop processing when the termination condition is fulfilled. Finally, the results of all LPs are merged into one combined result.

The implementation of the decomposition is described in Section 6.4.1. The remaining sections deal with the implementation of parallel LP simulation. The simulation procedure is explained in Section 6.4.2. This section illustrates LPs communication and progress during simulation run. Section 6.4.3 points out the how times safe to process are set. The termination condition for simulation is explained in Section 6.4.4. The high performance barrier implementation is described in Section 6.4.5.

6.4.1 Decomposition

The decomposition can be split into two parts. First we decompose the net into minimum regions. We implement a minimum region decomposer which creates a set of LPs by walking through the net structure. At the beginning a list of all free places is created that represents the places not yet attached to a minimum region. The list of places is processed by iterative removal. Each initial removal starts a new minimum region. For each removed place we check all outgoing transitions whether they are connected to another incoming place not yet attached to a minimum region. These input places are in conflict to the currently viewed input place. The aim is to clamp conflict sets in a LP. Thus, we remove all places from the free list which are input places for the transition and add it to the minimum region. The corresponding transitions are added to the minimum region as well. Then we test for the newly added places whether they share transitions with places still in the list of free places not yet attached to a minimum region. If this procedure does not yield a new place, the next removal from the free place list starts a new minimum region. The overall procedure stops once the list of free places is empty.

A LP is created for each minimum region. The LPs can be merged according to merging rules as described in Section 5.2.2.2 and Section 5.2.2.3. A merge function has been implemented. The fundamental merge function enables the integration of multiple merging rules. We implemented a subset of the merging rules described in Section 5.2.2.2 and Section 5.2.2.3. The implementation supports `Rule 6` and `Rule 7` with the constraint that we merge LPs in lane without consideration of lookahead borders. Additional rules have not been implemented as they are not needed in the evaluation for this thesis. The current implementation enables to decompose QPNs. However, parallel simulation is very sensitive to the decomposition. Complex models require additional merging rules for an optimal decomposition. The current implementation has to be extended to be applied in a fully automated process. A more detailed decomposition strategy is left for future work.

Given that we use an active wait for barrier synchronization, there is an additional constraint to decomposition. An active wait barrier synchronization implementation requires a decomposition with less LPs-threads than cores available. The number of threads can be reduced to be less than the number of LPs by a multiprocess which unifies multiple LPs. This enables to fit the number of threads to the number of available cores. The multiprocess approach has been described in Section 5.2.2. However, the optimization of the load balancing is not in the focus of this thesis. Therefore, the multiprocess functionality has not been implemented so far.

6.4.2 Simulation Procedure

The decomposition step delivers a set of LPs which we then simulate in parallel. For the event level parallelization we build upon the classes from the `java.util.concurrent` library. The LP class implements the `Runnable` interface. The corresponding `Runnable` is passed to a `Thread`. The main simulation procedure starts all LPs-threads and waits until they finish simulation. The main simulation logic is implemented within the LP. Before the LP threads enter the main simulation loop the data structures representing the local state have to be initialized. The initialization of the LPs is performed in parallel. We ensure that all LPs have been initialized by an additional barrier synchronization before the main simulation loop.

After initialization, each LP simulates its region of the net and communicates with the other LPs to increase its time safe to process. LPs enter the inner loop when all LPs finished initialization. The inner loop is the actual simulation. One part of the inner loop is barrier synchronization. Barrier synchronization enables improvements at the LBTS calculation. However, barrier synchronization is expensive and may be a performance bottleneck. The common 2^{nd} generation algorithm skeleton in Algorithm 3 uses two barriers within the inner simulation loop.

Algorithm 3 Communication with two barrier synchronizations

```
1: while not simulation finished do
2:     processEventsSafeToProcess();                    ▷ Details see Algorithm 5
3:     waitForBarrier();
4:     updateTimeSafeToProcess();
5:     waitForBarrier();
6: end while
```

Barrier synchronization is costly. It is possible to avoid the second barrier synchronization by performing some actions within the barrier. The LBTS calculation is performed sequentially before the barrier is released. The skeleton of the optimized algorithm is depicted in Algorithm 4.

Algorithm 4 Communication with one barrier synchronization

```
1: barrier ← global barrier action
2:           ▷ The barrier action sequentially sets times safe to process for all LPs
3:              ▷ The barrier action is executed when LPs have entered the barrier
4: while not simulation finished do
5:     processEventsSafeToProcess();                    ▷ Details see Algorithm 5
6:     waitForBarrier();
7: end while
```

A general consideration of Algorithm 3 and Algorithm 4 yields no superiority of one approach. Algorithm 4 minimizes barrier operations. Algorithm 3 enables parallel processing of the time safe to process. In context of SimQPN, time for event processing is very small. The smaller the time for event processing, the more expensive is the barrier operation compared to event processing. Hence, Algorithm 4 suits better for the parallelization of SimQPN.

The event processing within LPs slightly differs from sequential simulation. Small changes to simulation logic have to be applied. The firing logic of transitions has to be adapted. The transition has to check whether it fires to a place within the LP or whether it fires to another LP. If the place is within the LP, the transition just fires as in sequential mode. If the place is in a different LPs, the transition adds a token to the incoming token list of the successor. Later the successor removes the token from incoming token list and adds it to the corresponding place.

The incoming token list has to be synchronized as the LP and its predecessors can access concurrently. For some LPs the incoming token list is not necessarily unique. A LP that employs a FCFS incoming token list implementation requires as much incoming queues as it has predecessors. For multiple predecessors the order of timestamps is not necessary equal to the order of arrival. The order of timestamps is only ensured per predecessor because each LP sends tokens in timestamp order. The incoming lists can be merged if the order of events is ensured.

We unified all incoming token lists by the help of a *PriorityQueue*. This brings advantages. At the processing of incoming token events only one instead of multiple lists has to be checked. The tokens are already ordered. The insertion operation for a FCFS-Queue is in $\mathcal{O}(1)$ and for PriorityQueue is in $\mathcal{O}(n)$, whereas n terms the number of currently enqueued events. However, when the load level is low we can assume an almost constant overhead. Under the assumption of almost constant overhead for we chose a PriorityQueue implementation instead of multiple queues.

For LPs with one predecessor a priority queue creates unnecessary operations for the right insertion point. A simple FCFS queue would perform better in this scenario. Hence, we initialized the incoming token list scenario dependent on the number of predecessors:

```
if numberOfPredecessors $>$ 1 then
    incomingTokenList $\leftarrow$ new PriorityBlockingQueue<TokenEvent>();
else
    incomingTokenList $\leftarrow$ new ConcurrentLinkedQueue<TokenEvent>();
end if
```

From now on, we expect the LPs to be initialized start the description of LP internal event processing. The basic strategy is to process incoming token events and queue events in timestamp order and to fire transitions whenever possible. Algorithm 5 depicts the previously mentioned strategy and formalizes how safe events are processed. `Line 5` describes the actualization of residing service times for PS queues. `Lines 7` to `17` describe the case the event list holds jobs not yet processed. The case when no jobs are enqueued is depicted in the `lines 19` to `32`. Here we process incoming tokens if available. The representation slightly differs from the original implementation. For formating reasons some if clauses are split into two if clauses but functionality remains exactly the same. Compared to the sequential simulation, the processing of queue events has to be synchronized with the processing of incoming token events. Moreover, the queue event list has to be tested for emptiness. An empty queue event list does not occur in sequential simulation except for the case that the simulated QPN model is not *live* or in a deadlock situation (PN properties in Section 2.1.2). For LP simulations this scenario appears more frequently due to decomposition. Apart from the mentioned changes, the simulation logic remains untouched for the most part. The next section describes how the time safe to process is determined.

6.4.3 Lookahead

The update of times safe to process is a time critical aspect. An update that takes more time than the event processing precludes speedup. To reach a speedup, the update has to be markedly faster than event processing. Hence, we had to take a tradeoff decision between extensive search for best lookahead and efficient calculation of lookahead.

The performance models shipped with QPME use different token colors. In most cases the distinction of the incoming places is sufficient to differentiate between different colors. Hence, the implementation does not differentiate token colors.

Some of the queueing strategies inhibit good lookahead characteristics. Section 5.2.3.3 discusses how the prediction can be improved. The better lookahead has to be translated

Algorithm 5 Processing of events safe to process

```
function processSafeEvents
    QueueEvent nextQueueEvent = null;
    TokenEvent nextTokenEvent = null;
    while true do
        updateQueueEvents();                    ▷ Updates service times in PS queues
        if (nextQueueEvent = eventList.peek()) != null then
            if nextQueueEvent.getTime() <= timeSafeToProcess then
                                    ▷ timeSafeToProcess will be described in Algorithm 7
                while (nextTokenEvent = incomingTokenList.peek()) != null do
                    if nextTokenEvent.getTime() < nextQueueEvent.getTime() then
                        processNextTokenEvent();
                    else
                        break;
                    end if
                end while
                processNextQueueEvent();
            end if
        else if nextTokenEvent = incomingTokenList.peek()) != null then
            if nextTokenEvent.getTime() <= timeSafeToProcess then
                while true do
                    processNextTokenEvent();
                    if (nextTokenEvent = incomingTokenList.peek()) != null then
                        break;
                    end if
                    if nextTokenEvent.getTime() <= timeSafeToProcess then
                        break;
                    end if
                end while
            else
                fireTransitions();
                break;
            end if
        end if
        fireTransitions();
    end while
end function
```

into an increased number of processed events between barrier synchronizations. Otherwise we cannot benefit from advanced rules. We tested the potential for models from existing benchmarks applying overly optimistic lookahead (the time of the next event or higher). This breaks the local causality constraint but enabled an estimation for the potential speedup due to improved lookahead calculation. The number of events between barrier synchronizations did not rise sufficiently to justify the additional overhead. In our test cases it was not worth the effort.

In order to determine the time safe to process we determine for each LP the time of the next event assuming no token arrives. Algorithm 6 depicts the implementation for the next event time. This time is the minimum among all times in the incoming token list and the times of already enqueued jobs in the event list. We return zero to indicate that both lists are empty.

Algorithm 6 Time of next event function

```
 1: QueueEvent queueEvent ← eventList.peek();
 2: TokenEvent tokenEvent ← incomingTokenList.peek();
 3: if queueEvent != null then
 4:     if tokenEvent != null && tokenEvent.getTime() < queueEvent.getTime() then
 5:         return tokenEvent.getTime();
 6:     end if
 7:     return queueEvent.getTime();
 8: else
 9:     if tokenEvent != null then
10:         return tokenEvent.getTime()
11:     else
12:         return 0;                                ▷ Indicates the LP has no token
13:     end if
14: end if
```

Next, the results are combined to receive a global time safe to process. Algorithm 7 combines minimum event times to a global time safe to process which is spread to all LPs. The combination is improved by two optimizations. We consider only those LPs which may influence successors. Moreover, we can ignore LPs containing no token.

Algorithm 7 Global time safe to process function

```
 1: double timeSaveToProcess = Double.MAX_VALUE;
 2: for lp ∈ lpArray do
 3:     if lp.numOfSuccessors == 0 then          ▷ Ignore LPs that have no successors
 4:         continue;
 5:     end if
 6:     double time ← lp.getNextEventTime();
 7:     if timeSafeToProcess > time then
 8:         if time != 0 then                     ▷ Ignore all LPs that have no token
 9:             timeSaveToProcess ← time;
10:         end if
11:     end if
12: end for
13: return min
```

It has to be stated, that this time save to process implementation does not exploit the advanced techniques proposed in Section 5.2.3. Lookahead calculation is a tradeoff decision between speed and accuracy. This decision depends on the models at hand. The chosen implementation may not be the best tradeoff decision for all scenarios. Hence, the evaluation of additional LBTS strategies is an option for future work.

6.4.4 Termination Condition

SimQPN offers two options to end simulation. Simulation ends when either the total run length has been reached or required statistics have been gathered. The first termination condition is globally known before simulation starts. The sequential SimQPN version checks after each event whether condition for termination is reached. However, at parallel simulation time may be different at each LP. Our implementation checks for termination within the barrier. Thereby, the parallel simulation may stop few events later than sequential simulation.

The second termination condition checks for each net element whether the collected statistics suffer to reach the required precision. In parallel simulation, each LP checks the statistics for the net elements dedicated to it. LPs that have finished statistic collection cannot stop simulation but have to progress simulation as other LPs may rely on their tokens. The simulation procedure continues but statistic collection may be turned off. Simulation stops when all LP have collected statistics at required precision.

6.4.5 Barrier Optimizations

In parallel simulation the workload between barrier synchronizations depends on lookahead and the simulator implementation. In case of SimQPN and software performance models the processing of workload takes few microseconds. The barrier is entered very often during simulation. Due to that, an effective barrier implementation is a precondition for an effective parallelization. We applied two optimizations compared to standard barrier implementations, like the `CyclicBarrier` provided in SUN's `java.util.concurrent` library. We used

- busy wait
- hierarchical barriers

Busy wait means that the LP does not release the CPU when it enters the barrier. Instead, it repeatedly checks to see whether the other LPs reached the barrier. Busy wait wastes CPU cycles compared to passive wait. However, barriers applying busy wait in Java are remarkably faster than a passive *wait and notify* strategy. The effect can be explained by the weak Java memory model [BB03].

The barrier entering function has to be synchronized and LPs have to enter the barrier sequentially. This is no problem if LPs arrive sequentially at the barrier with sufficiently large time gaps at the barrier. But in a perfectly balanced parallelization scenario all LPs try to enter the barrier at the same time. Small workloads during simulation cycles increase the chance for access contention. Further, the effect increases, when an increasing number of LPs access the barrier. To weaken access contention we used hierarchical barriers instead of one centralized barrier.

The idea behind hierarchical barriers is to reduce the access contention by the introduction of multiple low-level barriers. The set of LPs is partitioned and each subset is assigned to a low-level barrier. The low-level barriers notify higher level barriers if all dedicated LPs have entered. Once the highest level barrier has been notified by all its sub-barriers, the barrier release message can be processed backwards through the tree and the barrier is released.

Common barriers are tree barriers and butterfly barriers [Fuj00b]. The JBarrier library [2] developed at University of Bonn includes these and multiple other hierarchical barrier implementations. A small restriction of their implementation is that the hierarchical implementations require the number of threads dedicated to the barrier to be a power of two. All barriers provided by the JBarrier library use busy wait. Their implementation is based on the fundamental paper of barrier synchronization in Java by Ball and Bull [BB03]. Experiments [3] show the high performance of barrier implementations from the JBarrier library. Our goal of the fastest possible simulation requires a fast barrier implementation which prompted us to included their library. Our implementation uses a butterfly barrier if the number of LPs is a power of two. Otherwise a central barrier implementation is used.

6.5 Functional Parallelism

The parallelism on functional level can be applied orthogonal to the other levels. Furthermore, it offers the possibility to utilize idle times of the event level approaches. This approach requires the identification of functional helpers with relevant impact on performance. If the impact is too low, the overhead may destroy positive effects. Hence, it is not useful to parallelize every possible functional helper.

We search for relevant functions via sampling and profiling. We start from sampling and then profile promising functions with VisualVM [4]. We meter the share on overall runtime before we implement parallel execution of functional helpers. The higher the share, the better the potential of the function. The next subsections describe our approach on the two most promising candidates: statistic processing and random number generation.

6.5.1 Statistic Processing

Statistic processing is a candidate that suits well for parallelization on a theoretical basis. During simulation run small statistic operations can be swapped to another core. However, SimQPN does not provide a statistical function with a relevant share on overall simulation time. Hence, it was impossible to implement a beneficial parallel statistic processing. Instantaneous statistic processing performs better than expensive distribution. This makes SimQPN not very promising to perform statistical processing in idle times during event-parallel simulation. Another option for parallelization is the analysis subsequent to simulation runs. The final processing of statistics subsequent to simulation is performed within few milliseconds. Hence, no noteworthy improvement could be expected when forecasting final analysis into the idle times.

Summing up, SimQPN provides no statistical helper functions with relevant impact on performance. Hence, we did not parallelize SimQPN's statistic processing.

6.5.2 Random Number Generation

A function that has often been proposed for extraction is the random number generation. SimQPN utilizes the `Colt` library [5] for random number generation. Colt has been developed at CERN for high performance scientific and technical computing in Java. The random number generation is performed via Mersenne twister [MN98]. This method for pseudo random number generation is efficient and enables fast random number generation. We tested the runtime share of random number generation on different models. The share

[2] `http://net.cs.uni-bonn.de/de/wg/cs/anwendungen/jbarrier/`

[3] `http://net.cs.uni-bonn.de/de/wg/cs/anwendungen/jbarrier/jbarrier-performance/`

[4] `http://visualvm.java.net/`

[5] `http://acs.lbl.gov/software/colt/`

on overall time was very small. Table 6.1 shows the share for two representative scenarios. A detailed description of the models can be found in Section 7.1.

	Share on overall runtime in percent	
Model	cern.jet.random.Exponential.nextDouble()	all Colt functions
pepsy-bcmp2.qpe	1.3	5.3
SPECjms2007Model.qpe	2.5	4.1

Table 6.1: Experiment on impact of random number generation

The maximum improvement due to parallel random number generation is estimated to be less than six percent. In order to do not miss the chance of performance improvement, we implemented a separate thread that circulates through the queues and fills the future list. We tested on a machine with eight cores so that the parallel progress of the thread can be ensured. For the sequential simulation we experienced neither a performance degradation nor a performance improvement. Hence, for sequential simulation it is not worthwhile to use this parallel extension.

For the event-parallel implementation more random numbers have to be calculated to fill the future lists. The more random numbers have to be generated, the higher the share of random number generation on overall runtime rises. However, this effect amortizes even for short simulation runs as future lists have only one big fill operation at the beginning of simulation. The share of random number generation rises marginally at event-parallel simulation. Hence, a parallelization would not promise any improvements.

One approach is to fill the future list in idle times that may appear during event level parallelization. The test with the additional random number thread showed the search for idle times to be aimless. Hence, future lists are filled by lazy filling instead of idle filling.

Neither for event-parallel nor for sequential simulation did we reach measurable speedup due to parallel random number generation. Hence, the delivered version employs sequential random number generation.

7. Validation

The objective of this chapter is to validate the correctnes of the parallel simulator implementation developed in the context of this thesis. It is important that the parallel version yields the same precission as the sequential version. Before we describe the experiments we explain the models we tested on in Section 7.1. Then, in Section 7.2, we describe the validation.

7.1 Models

In order to derive realistic scenarios we use models and parametrization from the following examples shipped with the actual QPME version:

- **SPECjAppServer2001** The QPN models a distributed e-business system [KB03]. The file is named ispass03.qpe.
- **Product form QN** The QPN is a closed product-form queueing network with two request classes[KB06]. The network pepsy-bcmp2.qpe is a transformation of an example shipped with the Performance Evaluation and Prediction SYstem for Queueing NetworkS (PEPSY-QNS) tool [BK94].

7.2 Validation

To replace a sequential by a parallel simulation engine it is required to get the same precision. We compare the final statistics of analysis methods. Those statistics contain mean and standard deviation for multiple variables. The variables are for example, the token arrival and departure counts, service time, token occupancy, et cetera.

The following experiments ensure the same seeds for the runs to be compared. The colt package sets the initial random seed according to a `java.util.Date` object. Therefore, we synonymously use the terms *date* and *initial seed* in the following. Please do not attach importance to the randomly chosen dates. We picked the following dates/seeds for each experiment:

- Thu Jan 01 01:00:00 CET 1970
- Fri Oct 12 02:00:03 CEST 2012
- Fri May 31 13:33:20 CEST 2013

- Sun Mar 13 08:06:40 CET 2011
- Fri Jul 14 04:40:00 CEST 2017

We validated parallel versions of Replication/Deletion and Batch/Means. In case of parallel Replication/Deletion we tested only for one example as the simulation engine did not change. The parallel version ensures exactly the same results as sequential simulation. In spite of highly resolved statistics none of the five experiments showed any deviation between sequential and parallel version.

We tested with `pepsy-bcmp2.qpe` and reused an existing configuration and set the number to eight replications. Table 7.1 lists the elements and their success with regard to passing the test on equality of sequential and parallel simulation.

Model	Element	Part	Test for equality	
			Passed	Missed
pepsy-bcmp2.qpe	Terminals	Queue	X	
pepsy-bcmp2.qpe		Depository	X	
pepsy-bcmp2.qpe	CPU	Queue	X	
pepsy-bcmp2.qpe		Depository	X	
pepsy-bcmp2.qpe	Disk1	Queue	X	
pepsy-bcmp2.qpe		Depository	X	
pepsy-bcmp2.qpe	Disk2	Queue	X	
pepsy-bcmp2.qpe		Depository	X	
pepsy-bcmp2.qpe	Dis k3	Queue	X	
pepsy-bcmp2.qpe		Depository	X	

Table 7.1: Validation of Replication/Deletion

We tested the Batch/Means implementation on two models. The experiments on ispass03.qpe, depict in Table 7.2, showed no deviation between sequential and parallel version.

Model	Element	Part	Test for equality	
			Passed	Missed
ispass03.qpe	WLS-Thread-Pool	Ordinary Place	X	
ispass03.qpe	Client	Queue	X	
ispass03.qpe		Depository	X	
ispass03.qpe	DBS-Conn-Pool	Ordinary Place	X	
ispass03.qpe	DBS-Process-Pool	Ordinary Place	X	
ispass03.qpe	WLS-CPU	Queue	X	
ispass03.qpe		Depository	X	
ispass03.qpe	DBS-PQ	Ordinary Place	X	
ispass03.qpe	DBS-CPU	Queue	X	
ispass03.qpe		Depository	X	
ispass03.qpe	DBS-I/0	Queue	X	
ispass03.qpe		Depository	X	

Table 7.2: Validation of Batch/Means on ispass03.qpe

The Batch/Means experiments on ispass03.qpe showed no deviation. However, experiments on pepsy-bcmp2.qpe show deviations between parallel and sequential simulation.

Model	Element	Part	Test deviation ≤ 1 percent	
			Passed	Missed
pepsy-bcmp2.qpe	Terminals	Queue	X	
pepsy-bcmp2.qpe		Depository	X	
pepsy-bcmp2.qpe	CPU	Queue	X	
pepsy-bcmp2.qpe		Depository	X	
pepsy-bcmp2.qpe	Disk1	Queue	X	
pepsy-bcmp2.qpe		Depository	X	
pepsy-bcmp2.qpe	Disk2	Queue	X	
pepsy-bcmp2.qpe		Depository	X	
pepsy-bcmp2.qpe	Disk3	Queue	X	
pepsy-bcmp2.qpe		Depository	X	

Table 7.3: Validation of Batch/Means on pepsy-bcmp2.qpe

These deviations can be explained by the differences in the choice of the next transition. In both versions the next transition to fire is chosen based on an array that contains the transition weights of enabled transitions. In sequential simulation the passed array has a size of all transitions belonging to the net. In parallel simulation, the array contains only the transitions belonging to the LP. The distribution that chooses the next transition is the same but has different samples due to different input. Hence, even the same initial random seed cannot ensure equal results. A logging showed that the effect of different transition choices occurs at pepsy-bcmp2.qpe. According to the law of large numbers we can expect that simulation results for parallel and sequential simulation converge the longer simulation proceeds. Table 7.3 compares sequential and parallel implementation for pepsy-bcmp2.qpe. To show that the difference is small, we compare the runs assuming sequential and parallel version to be equal if the deviation is below 1 percent. The deviations are comparable to a comparison of sequential simulations at different random seeds.

In some scenarios the parallel version may simulate a few additional events. Compared to the sequential version, our parallel Batch/Means implementation does not test the termination condition after each event. Instead, we test within the barrier. Therefore, small deviations may occur when two preconditions are fulfilled. The first precondition is that multiple events are processed between the last two barrier executions. The second is that these events exceed the termination condition more than once. Through this, multiple events are processed before the final test for termination and a slightly enhanced time interval may be simulated. The high number of events usually processed make this effect insignificant.

The validation includes a subset of the whole SimQPN functionality. We restricted the test to fixed sample size length simulation. SimQPN offers termination at relative and absolute precision which has not been tested as they are not required for speedup analysis evaluation for this thesis. Likewise HQPNs and probes have not been tested.

Summing up, we performed a basic validation for both parallelization approaches. The parallelization of Replication/Deletion returns exactly the same statistics as sequential simulation.The parallel Batch/Means shows no deviation to sequential simulation whenever no random choice of the next transition is required. If the model requires a random choice of the next transition to fire, the final statistics may slightly differ. These effects are negotiable as the employment of a different random seed has a similar effect.

8. Evaluation

This chapter the performance evaluation of the parallel SimQPN version is described. We analyze the parallel implementations of Replication/Deletion and Batch/Means. For each approach we compare the parallel to the original sequential version. The parallel implementation is presented in Chapter 6 and validated in Chapter 7.

This chapter starts with the description of the experimental setup in Section 8.1. We name the test environment in Section 8.1.1. Then we define the metrics *speedup* and *level of parallelism* in Section 8.1.2 which will be used for the comparison of the parallel and sequential simulation. Moreover, we explain the structure of the QPN models used for analysis in Section 8.1.3.

First, we describe the experiments for the parallelization of the Replication/Deletion approach on application level in Section 8.2. Second, we test the event level parallelization of Batch/Means in Section 8.3. The experiments on event level parallelism contain a comparison of active versus passive wait in Section 8.3.1 and an impact analysis of different levels of statistic collection in Section 8.3.2. Experiments that vary workload between barrier synchronizations ar shown in Section 8.3.3. Experiments that vary the number of LPs that may process concurrently are described in Section 8.3.4

8.1 Settings

Performance evaluation needs a proper setting to show strengths and weaknesses of the parallel simulation. There exist mainly two degrees of freedom for evaluation:

Simulation Environment Describes the setting independent of the model.

- Architecture (number of cores, computer architecture, cache sizes)
- Software (operating system, Java version)

Simulation Model Describes performance-relevant properties of the QPN (size, degree of parallelism, initialization, level of statistics)

Besides these aspects, this section defines the metrics for evaluation.

8.1.1 Simulation Environment

The test environment has a high impact on the results. We ran our experiments on a Intel Xeon E5430 2x4core, 2.66GHz, 12MB L2 Cache per CPU and 32 GB RAM. We tested on a Linux CENTOS 5.8 (Final). All tests were conducted by the help of Eclipse Modeling Tools Version: Juno Service Release 2. The code was compiled and run with Java 64bit JDK 1.7.0_04.

8.1.2 Metrics

The relevant basic metrics for the performance evaluation are the number of cores and the runtime of the investigated simulation. These basic metrics enable to build the composed metrics speedup and level of parallelism. We define the speedup as follows:

Definition 12 (Speedup)**.**
For a number of c cores the speedup S_c of a program is defined as

$$S_c = \frac{T_1}{T_c}$$

whereas T_1 terms the time the program runs on a single core and T_c is the time the program needs on c cores.

The usual range of S_c is defined as $1 \leq S_c \leq c$. A speedup equal to the number of cores c shows a full utilization of cores. A speedup equal to the number of cores $S_c = c$ is called linear speedup. A speedup $S_c \geq c$ is called superlinear speedup. The speedup heavily depends on the number of cores which makes it a non-intuitive metric. Hence, a metric that is less dependent on the number of cores has its benefits. We introduce the level of parallelism as follows:

Definition 13 (Level of Parallelism)**.**
The level of parallelism of a program is defined as:

$$L = \frac{S_c}{c}$$

whereas c terms the number of cores and S_c terms the speedup of the program.

8.1.3 Simulation Model

To test for speedup we used multiple models. In some experiments can reuse the real world models introduced in Section 7.1. For most experiments a simple model structure enables to enlighten several effects. Hence, we create artificial models which enable variations to explain influencing factors. Especially the speedup on event level parallelization depends on the model characteristics. The net properties determine the degree of parallelism the parallel implementation can utilize.

The artificial model is composed as follows. The model generates tokens and distributes them to multiple lanes. Token generation and distribution remains a constant block whereas the number and length of lanes can be varied.

The lanes can execute in parallel. Thereby, each incrementation of the lane number enables the utilization of an additional processor. The length of the lanes determines the number of events scheduled before next synchronization. By that, the variation of length enables to increase or decrease the workload until the next synchronization.

An example for a generated model is depicted in Figure 8.1. The example has three lanes and two queueing places per lane. We term the model `lane 3 x 2`. A model with five lanes and ten places per lane would be `lane 5 x 10`.

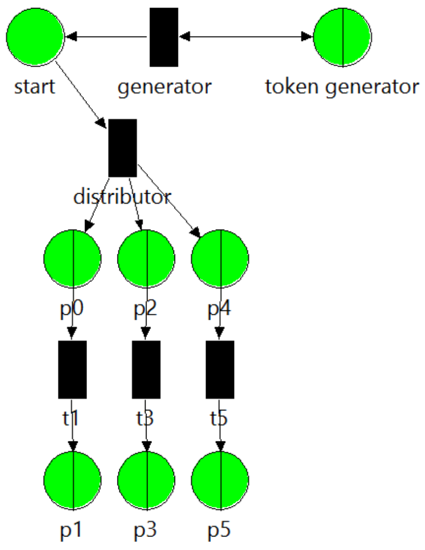

Figure 8.1: Generated Model with Three Lanes Each of Length Two Places.

We adapt the token generation frequency to the length of lanes. The service time of `token generator` is set to the length of lanes. A new token passes the `token generator` when the lanes finish the service at the last queueing place.

What happens on a technical level? The initial marking of the QPN contains one token in `token generator`. The `token generator` processes the initial token and fires the `generator` transition. The transition multiplies this token and fires one token to place `start` and one back to `token generator`. The `distributor` transition removes the token from start and fires a token to each lane. The tokens process through the lanes. Then the procedure starts from the beginning.

The decomposition for this models creates one partition for the token generation (`token generator`, `generator`, `start`, `distributor`). Moreover, each lane forms a partition. The simple structure of our generated models enables a prediction of theoretical speedup. For n lanes of length m and k being the constant overhead for the token generation LP the possible speedup is

$$speedup = n * m + k \tag{8.1}$$

This view excludes memory effects and other decomposition effects. Nevertheless, we see this a as good indicator to assess the parallelization on event level. The next subsections describe different experiments on performance.

8.2 Performance of Application Level Parallelism

In this section we evaluate the application level parallelization of the Replication/Deletion approach. The runtime for all experiments in this section are depicted in Table 8.1.

	Runtime in seconds			
Model	Number of Replications	Sequential	Parallel	Speedup
pepsy-bcmp2.qpe	2	43,738	25,801	1,6952056122
pepsy-bcmp2.qpe	3	62,715	24,984	2,5102065322
pepsy-bcmp2.qpe	4	83,187	27,321	3,0448007028
pepsy-bcmp2.qpe	5	104,769	26,663	3,9293777894
pepsy-bcmp2.qpe	6	130,795	30,084	4,3476598857
pepsy-bcmp2.qpe	7	141,002	26,005	5,4221111325
pepsy-bcmp2.qpe	8	170,497	27,878	6,1158260994

Table 8.1: Comparison of sequential and parallel Replication/Deletion

To show the speedup of the parallel implementation of Replication/Deletion, we test on the pepsy-bcmp2.qpe model that has already been used for validation in Section 7.1 Figure 8.2 shows speedups compared to the number of replications executed in parallel. The speedup increases linear with the number of replications but remains slightly below the number of replications.

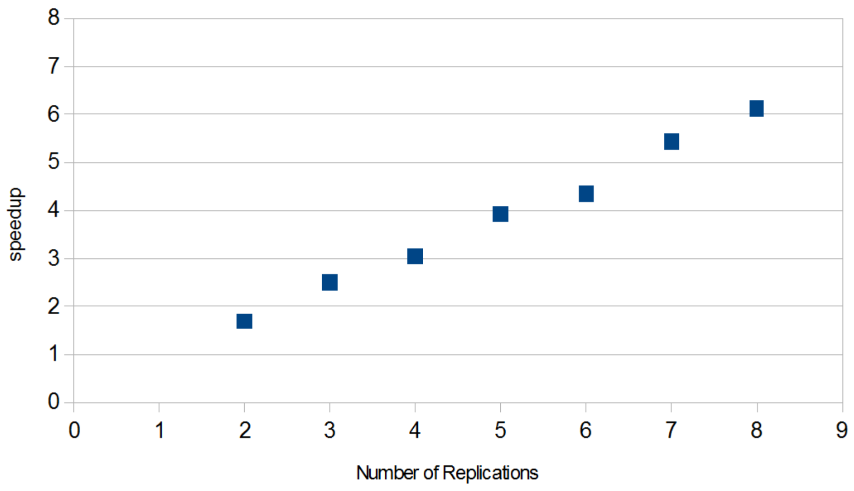

Figure 8.2: Speedup Replication/Deletion

A reason for the deviation from linear speedup can be different start seeds for each run. The Replication/Deletion approach starts multiple replications, each with a different start seed to study the indeterministic behavior of the model. Hence, the duration for execution time for the runs may vary the start seed. We made further experiments with a deterministic model which is independent of transition choices and random service times. The deterministic model yields a higher speedup but speedup that, however, still deviates from the number of replications used.

We further investigated possible reasons why we deviate from linear speedup. We tested for I/O-Wait to exclude negative effects of hard drive access. Except for the case the model contains the rarely used statistics level 5, the hard drive is never accessed during simulation except for logging purposes. The tests for CPU utilization show full utilization except for a limited time at the end of the simulation when some of the replications have already finished simulation. Software contention describes the effect of multiple transactions competing for the same resources. Whenever software contention occurs the utilization of cores decreases. The tests for CPU utilization showed that software contention for resources can be excluded to be responsible for slowdown. Next we consider additional overheads due to parallelization. The only aspect that creates additional computational effort is the thread pool creation which is performed in about 0.05 seconds. On our search for aspects limiting speedup we investigated the sequential parts of the implementation which are:

- net copy,
- garbage collection and
- logging.

We investigate the sequential net copy and metered about 0.03 seconds for all net copy operations. By that, the net copy operation cannot be a relevant slowdown factor. The overhead due to garbage collection in SimQPN is small. Nevertheless, even small delays may sum up to bigger delay. Hence, we tested for garbage collection effects. The overhead for garbage collection can be displayed on console throughout the VM argument *-verbose:gc*. Table 8.2 shows experiments on garbage collection. The garbage collector speed increases when using parallel simulation. We reach a speedup of 5,6 with parallel garbage collector instruction and 4,8 without instructions compared to sequential simulation. The repetition of these experiments showed only slight deviations so that we can assume garbage collector overhead to be constant. Throughout the Java internal parallelization of garbage collection the effects of garbage collection are negligible regarding overall parallelization.

Execution mode	Java VM instruction	Runtime in seconds
sequential		0.640161
parallel		0.134327
parallel	-XX:+UseParallelGC	0.113950
parallel	-XX:+UseParNewGC	0.473132

Table 8.2: Experiment on the effect of garbage collection

We tested for multiple possible slowdown factors. None of them explains the deviation from speedup equal to the number of cores. The limited time frame of this thesis prevented from testing all aspects that might cause the slowdown. We assume that cache misses play a role but we have not tested to far. An explanation for slowdown would be that replications change the executing core during execution. If a replication thread changes the core it cannot benefit from the information stored in the L1 Cache. The L1 cache stores loaded values per core. A change of core results in an increased cache miss rate. A solution would be to pin threads to cores. Pinning threads to cores is still experimental in Java. To the best of our knowledge there exists no platform-independent implementation so far. Another reason for slowdown may be contention for the shared LP2 cache. Each core has its own L1 cache but all cores of a CPU share the L2 cache. The replications are in contention for the L2 cache. Each replication overwrites the cache. During parallel simulation an increased cache miss rate may occur. The problem of inter-thread cache contention on a chip multi-processor architecture has been described in [CGKS05].

The discussion about possible slowdown factors should not depreciate the positive effects of application level parallelization. The implementation reaches an almost linear speedup slightly below the number of replications for less replications than cores. In case of more replications than cores, the speedup is slightly below the number of cores. Parallel Replication/Deletion performs markedly faster than the sequential version in all scenarios.

8.3 Performance of Event Level Parallelism

In this section, we evaluate the parallelization on event level with the Batch/Means approach. The first experiment in Section 8.3.1 shows a comparison of active wait and passive wait at the synchronization barrier. In these experiments, active wait outperformed passive waiting. Hence, the remaining experiments of this section use active wait as described in Section 6.4.5.

The speedup for event level parallelization heavily depends on the capability of the model for event parallel simulation. For these experiments we use artificial models because they enable to vary net characteristics. In order to evaluate the influence of statistic levels SimQPN allows to set, we show experiments that vary these levels in Section 8.3.2. Moreover, this section discusses on parallel simulation of ramp up which is equal to statistics level zero. Section 8.3.3 shows the effect of different lane lengths. This varies the number of events and thereby the workload between synchronizations. In Section 8.3.4 we vary the number of lanes. This enables to utilize different numbers of cores.

8.3.1 Experiments on Barrier Synchronization

Barrier synchronization can either be done waiting actively on barrier release or waiting passively on release. Active waiting threads enter the barrier and claim CPU-cycles without performing computations. Thereby, threads save the time to reenter the CPU when the barrier is released. A thread that uses passive wait releases the CPU on entering the barrier and reenters the CPU on barrier release. The wait and notify strategy of passive wait implies a high overhead within the operating system.

The experiments in Table 8.3 show a comparison of runtimes for active and passive wait for several models. Active wait outperforms passive wait in each scenario. In some scenarios the difference is less explicit. In those scenarios the the share of barrier executions on overall runtime is low. The more barrier operations the more evident the performance difference gets. The strength of active wait clarifies at fine-grained synchronization scenarios.

			Runtime in seconds	
Model			CyclicBarrier (passive wait)	JBarrier (active wait)
pepsy-bcmp2.qpe			23.907	14.453
	Runlength	Stats-level		
lane 4 x 1	10^7	0	296.369	38.285
lane 4 x 1	$4 * 10^5$	4	20.423	14.687
lane 2 x 2	$4 * 10^5$	4	11.293	8.893

Table 8.3: Experiment on barrier synchronization

The active barrier wait performs remarkably faster than passive wait in all test cases. As the goal of this thesis is to simulate as fas as possible, active wait is the method of choice. The following experiments only consider active waiting.

8.3.2 Experiments on Statistics Level

SimQPN distinguishes between six levels of statistic collection, called stats-level. These levels have been described in Section 2.3. The lower the stats-level, the faster the processing of events is done. This section observes the effects of different levels of statistic collection on runtime. Indirectly, we observe the effect of ramp up on event processing speed. Table 8.4 shows variations of stats-level at `lane 6 x 2` model parameterized with run length of 10^6 and a negligible ramp up length of 10.

Runtime in seconds				
Model	Statistics level	Sequential	Parallel	Speedup
lane 6 x 2	0	18.065	28.098	0.6429
lane 6 x 2	1	20.502	26.736	0.76683
lane 6 x 2	2	25.26	28.58	0.88383
lane 6 x 2	3	42.205	44.047	0.958181
lane 6 x 2	4	43.631	46.961	0.92909
lane 6 x 2	5	731.126	467.746	1.5638

Table 8.4: Experiment on different statistics level

The experiment shows the impact of statistic collection on runtime. An increased collection of statistics raises the runtime of sequential simulation. We deduce two useful items of information from this experiment. The first conclusion is that speedup increases by the overhead for statistics. The long runtime for stats-level 5 comes from continuous writing to a statistics file. Runtimes for stats-level 3 and 4 are almost equal. In general, models with lower stats-level suit less for parallel simulation. This insight helps to predict the suitability of QPN models for event parallel simulation.

What surprises on the first sight is that parallel execution with stats-level 1 performs faster than stats-level 0. This can be explained by barrier contention. The workload at stats-level 0 is infinitesimal so that all threads try to enter the barrier at the same time. Stats-level 1 creates more workload which decreases barrier contention.

During the ramp up phase, like stats-level 0, no statistics are calculated. The lack of statistic collection decreases the workload per event. This influences speedup as barrier synchronization requires a certain workload to benefit from parallel simulation. The workload between barrier synchronizations has to reach a certain level to make parallel simulation faster than sequential simulation. The workload during ramp up is almost unsuitable for parallel simulation within SimQPN.

For the chosen model we can gain speedup by the use of parallel implementation during steady state at least at stats-level 5. During ramp up parallel simulation is much slower. Our experiment points out future implementations should separate ramp up and steady state analysis. We recommend to run the ramp up phase sequentially and steady state in parallel.

8.3.3 Experiments on Length of Parallel Sections

The workload between barrier synchronizations influences the speedup. The aspect of statistic collection on workload has been shown. Moreover, the workload depends on the number of events that can be processed before the next synchronization. The more events we can process between barrier synchronizations the less synchronization operations for parallel simulation are required. We modify the workload in the artificial models by an extension of the length of lane. The length of the lane determines the number of events that each Lane-LP processes before it synchronizes at the barrier. Table 8.5 shows experiments with lane length variations. The used parameterization is run length of 10^7, an insignificant ramp up of 10 and a stats-level of 4.

Runtime in seconds			
Model	Sequential	Parallel	Speedup
lane 6 x 1	54.805	85.383	0.6418725039
lane 6 x 2	44.425	43.115	1.0303838571
lane 6 x 5	45.782	23.319	1.9632917364
lane 6 x 10	44.586	17.067	2.6124099139
lane 6 x 20	50.66	12.368	4.0960543338
lane 6 x 50	99.668	14.02	7.1089871612
lane 6 x 75	161.946	12.282	13.1856375183
lane 6 x 100	233.625	13.378	17,4633727015

Table 8.5: Experiment on the effect of different lengths of lanes

We can see that the higher the length of the lane, the higher the speedup. These experiments show that the event level parallelization enables superlinear speedup. This speedup higher than the number of cores is far ahead of the model inherent parallelism. Figure 8.3 illustrates the speedup effect by comparing lane length to speedup.

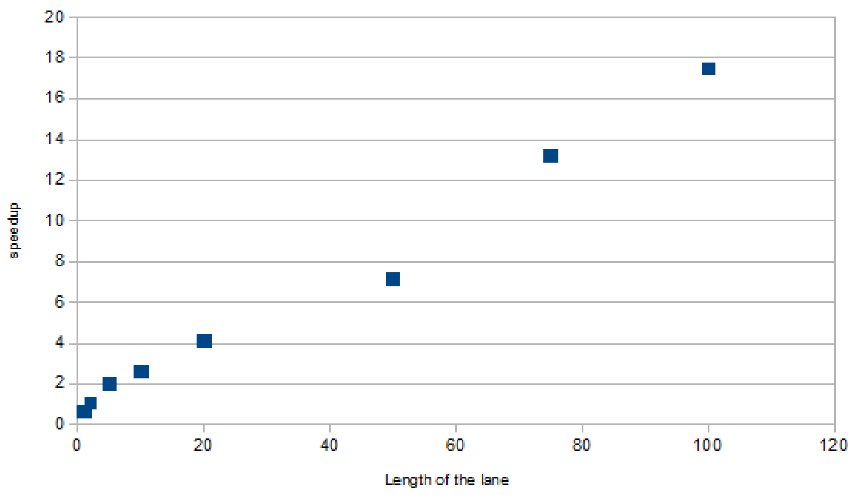

Figure 8.3: Experiment on the effect of different length of lanes

The general effect of an increased speedup for an increased lane length can be explained by the fact that the number of synchronization operations decreases with the length of the lane. This decrease of barrier operations reduces the overhead due to barrier contention which explains speedups up to linear speedup. We figured out two reasons for superlinear speedup. The first reason is caching effects. Each of the eight cores has its own L1 cache. In sequential simulation, caches have to hold the whole simulation model. In event parallel simulation, the decomposition into LPs reduces the amount of memory that caches have to hold during simulation. L1 caches do not have to hold the whole net during parallel simulation. Parallel simulation can utilize L1 caches much better sequential simulation.

Besides these caching effects, we can benefit from additional effects of decomposition. The decomposition lowers the load-level of event queues. The event queue during sequential execution holds all events currently created and not processed. In parallel simulation the queue events are stored per LP. Thereby, the load level at the local event queues during parallel simulation is lower than the global load level in sequential simulation. Decomposition brings a lower filling degree of the local LP event queues compared to the global event queue in sequential simulation. The sequential version has multiple events in event queue whereas LP event queues contain only one event. Thereby, insertion and removal operations at parallel simulation require less effort than in sequential simulation.

In case of multiple concurrently enabled transitions, SimQPN choses the next transition dependent on a random value. If only one transition is enabled, this transition can be fired directly which avoids the computation of the random value. While the sequential simulator choses the next transition based on all transitions, parallel simulation delegates this choice to its LPs which consider only transitions dedicated to them. Considering only subsets of transitions decreases the number of simultaneously enabled transitions. Hence, in parallel simulation it is unlikely to have expensive statistical choices of the next transition compared to sequential simulation.

8.3.4 Experiments on Number of Parallel Sections

Parallelization is the distribution of software execution to multiple cores. The distribution of the execution of the artificial model to cores can be varied in the number of lanes. An increase of lanes yields an increase of processor cores we can utilize. Table 8.6 compares simulation runs on models with different number of lanes. The used parameterization is run length of 10^7, a ramp up of 10 and a stats-level of 4.

Runtime in seconds			
Model	Sequential	Parallel	Speedup
lane 2 x 10	13,194	10,599	1,2448344183
lane 3 x 10	20,643	10,922	1,8900384545
lane 4 x 10	27,61	11,431	2,4153617356
lane 5 x 10	36,756	14,056	2,6149686966
lane 6 x 10	44,586	17,067	2,6124099139

Table 8.6: Experiment on the number of lanes

An increased number of lanes increases the number of cores that can be utilized by the parallel implementation. More events can be processed in parallel. However, synchronization costs for the central barrier rise. Figure 8.4 shows speedup compared to the number of lanes. The curve flattens for an increase number of parallel LPs competing for the barrier. This experiment indicates that each new section that is processed in parallel raises the contention for the barrier.

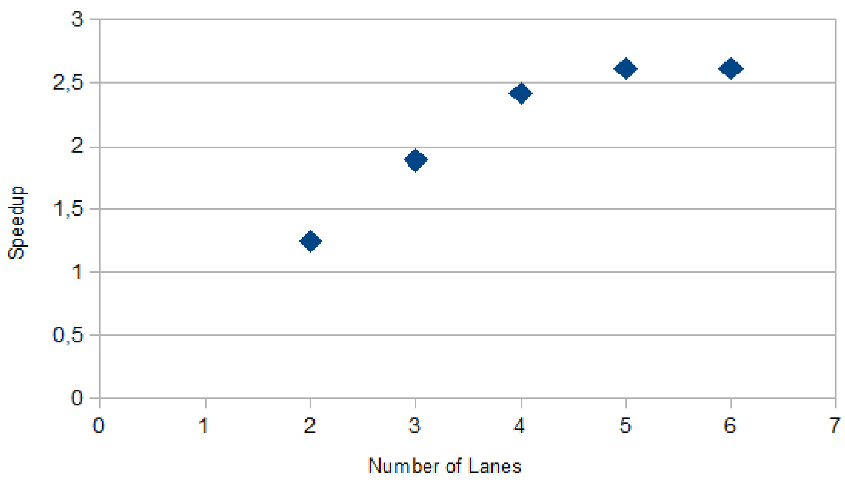

Figure 8.4: Experiment on the effect of different numbers of lanes

9. Conclusion

This thesis investigates the possibilities of parallel simulation of Queueing Petri Net (QPN). Thereby, it splits into two parts. The theoretical part discusses the possibilities of parallel simulation of QPNs in general. The practical part is about the implementation of different approaches in SimQPN. We parallelized the SimQPN simulation engine and evaluated the implementation on models from previous case studies and on artificial models.

We performed a theoretical analysis of strengths and weaknesses of the QPN formalism with respect to parallel simulation. To keep it reusable, this survey is independent of SimQPN. We surveyed existing parallelization approaches with a focus on Queueing Networks (QNs) and Petri Nets (PNs). Based on this, we provided a systematic analysis on three parallelization levels:

- application level
- event level
- functional level

The insights for these levels can be summarized as follows. On application level, multiple runs are simulated in parallel. This approach is independent of the formalism but can only be applied to analysis approaches that employ multiple independent runs. For event level parallel simulation we provided a summary of synchronization algorithms and named their strengths and weaknesses. We analyzed the characteristics of QPN models from previous case studies and proposed the use of a conservative barrier-based synchronization algorithm for QPN simulation. Furthermore, we generalized lookahead calculation and net decomposition rules from QNs and PNs to QPNs. On functional level, we identified common helper functions in the simulation which are candidates for moving to a separate thread.

Based on the theoretical analysis, we picked promising parallelization approaches for SimQPN. We parallelized the Replication/Deletion approach on application level and the Batch/Means approach on event level. For event level parallelization we applied a high performance barrier implementation which uses active wait and hierarchical synchronization. Both parallelization approaches were implemented based on the sequential simulation engine. Before parallelization we adapted the existing design to suit for parallel simulation. Functional parallelism was not implemented because the profiling of SimQPN showed no adequate functional helpers where significant performance improvements could be expected by running them in parallel to the core simulation loop.

We validated the correctness of both parallelizations by a comparison of sequential and parallel version. The experiments show the accuracy of the parallel implementations.

Finally, the speedup of the parallel implementations was evaluated on multiple scenarios. The Replication/Deletion showed a speedup slightly below the number of replications. One reason for this effect consists in different replication lengths due to different start seeds. However, a small overhead occurs as well.

The speedup for the event parallel simulation heavily depends on the capability of the model. Performance models from previous case studies did not suit for event parallel simulation. A systematic analysis on artificial models showed superlinear speedups to be achievable. We varied multiple model characteristics to show their influence on speedup. In particular we varied the level of statistic collection, the number of events between barrier synchronizations and the number of concurrently processing LPs.

9.1 Contributions and Benefits

In summary, the main contributions are

- a roundup for event level parallelization of QPNs which includes
  - a discussion on lookahead characteristics of QPNs and a comparison of these characteristics to existing synchronization approaches.
  - an overview about QPN decomposition which is required for event parallel simulation. Therefore, we transfered decomposition rules from SPN to QPN and introduced additional rules for lanes, distributor transitions and choice transition.
  - a disquisition on lookahead calculation for QPNs. For this purpose we transfered advanced lookahead calculation for multiple queueing strategies from QNs to QPNs.
- an implementation of a parallel simulation engine for QPNs. The SimQPN simulation engine has been parallelized to speed up analysis and provides
  - an event level parallelization for Batch/Means
  - an application level parallelization for Replication/Deletion
- a case study on artificial models which reviews the effects of different model characteristics for speedup on event level parallelization. In particular we review variations of the
  - level of statistics collection.
  - number of events between barrier synchronizations.
  - number of concurrently processing LPs.

9.2 Future Work

During this thesis many new questions arose. To conclude this thesis, we point out the most promising questions with regard to future work.

- Only a subset of QPN models suits for event parallel simulation. Further research may target the assessment of model suitability for event parallel simulation. The experiments in Section 8.3.2, Section 8.3.3 and Section 8.3.4 show the influence of statistics level, number of events between barrier synchronizations and degree parallelism. A goal for future work might be to derive techniques to determine the model inherent parallelism of QPNs in advance.

- Some models cannot benefit from parallel simulation during ramp up simulation whereas the steady state performs faster with parallel simulation. Section 8.3.2 depicts an example. Relevant nets have to be identified and the simulator has to be adapted to process ramp up sequentially and steady state analysis in parallel.
- The decomposition into minimum regions and few merging rules have been applied for QPN decomposition. The decomposition can by improved by
  - the implementation of additional merging rules to derive bigger partitions which minimizes communication overhead.
  - an adaption of the number of LP threads to the number of available cores. This may include to combine multiple LPs into a multiprocess.
  - the usage of runtime statistics for decomposition.
- We proposed a static assignment of LPs to threads which does consider changing workloads during simulation. A dynamic scheduling of LPs on physical threads may improve load balancing.
- The automation of truncation point analysis seems a reasonable extension for SimQPN. The MSER-5 method has been identified as state of the art truncation method in Section 2.2.2.1. An implementation of MSER-5 offers two benefits:
  - More precise determination of the truncation point which reduces simulation run length.
  - Avoidance of human interaction, which is required for Method of Welch.

Bibliography

[AD91] H. Ammar and S. Deng, "Time warp simulation of stochastic petri nets," in *Petri Nets and Performance Models, 1991. PNPM91., Proceedings of the Fourth International Workshop on*, Dec 1991, pp. 186–195.

[Aya89] R. A. Ayani, "A parallel simulation scheme based on the distance between objects," in *SCS Multiconference on Distributed Simulation 21*, March 1989, pp. 113–118.

[Bau93a] F. Bause, "Queueing petri nets-a formalism for the combined qualitative and quantitative analysis of systems," in *Petri Nets and Performance Models, 1993. Proceedings., 5th International Workshop on*, oct 1993, pp. 14 –23.

[Bau93b] ——, "Qn + pn = qpn - combining queueing networks and petri nets," Department of CS, University of Dortmund, Germany, Technical Report No.461, 1993.

[BB03] C. Ball and M. Bull, "Barrier synchronisation in java," *Available from World Wide Web: http://www. ukhec. ac. uk/publications/reports/synch_java. pdf*, 2003.

[BBK95] F. Bause, P. Buchholz, and P. Kemper, "Qpn-tool for the specification and analysis of hierarchically combined queueing petri nets," in *BAUSE (EDS.) QUANTITATIVE EVALUATION OF COMPUTING AND COMMUNICATION SYSTEMS, LECTURE NOTES IN COMPUTER SCIENCE.* Springer, 1995, pp. 224–238.

[BC93] F. Baccelli and M. Canales, "Parallel simulation of stochastic petri nets using recurrence equations," *ACM Transactions on Modeling and Computer Simulation*, vol. 3, pp. 20–41, 1993.

[BE03] F. Bause and M. Eickhoff, "Simulation output analysis: truncation point estimation using multiple replications in parallel," in *Proceedings of the 35th conference on Winter simulation: driving innovation*, ser. WSC '03. Winter Simulation Conference, 2003, pp. 414–421. [Online]. Available: http://dl.acm.org/citation.cfm?id=1030818.1030876

[BK94] G. Bloch and M. Kirschnick, "Pepsy-qns - performance evaluation and prediction system for queueing networks," University of Erlangen-Nuremberg, Institut für Mathematische Maschinen und Datenverarbeitung IV, Tech. Rep., June 1994. [Online]. Available: http://www4.informatik.uni-erlangen.de/Projects/PEPSY/en/pepsy.html

[BK02] F. Bause and P. S. Kritzinger, *Stochastic Petri nets - an introduction to the theory (2. ed.).* Vieweg, 2002.

[Bry77] R. E. Bryant, "Simulation of packet communication architecture computer systems," Cambridge, MA, USA, Tech. Rep., 1977.

[CF93] G. Chiola and A. Ferscha, "Distributed simulation of timed petri nets: Exploiting the net structure to obtain efficiency," in *Application and Theory of Petri Nets 1993*, ser. Lecture Notes in Computer Science, M. Ajmone Marsan, Ed. Springer Berlin Heidelberg, 1993, vol. 691, pp. 146–165. [Online]. Available: http://dx.doi.org/10.1007/3-540-56863-8_45

[CGKS05] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, 2005, pp. 340–351.

[CM78] K. Chandy and J. Misra, "Distributed simulation: A case study in design and verification of distributed programs," *Software Engineering, IEEE Transactions on*, vol. SE-5, no. 5, pp. 440–452, September 1978.

[CM81] K. M. Chandy and J. Misra, "Asynchronous distributed simulation via a sequence of parallel computations," *Commun. ACM*, vol. 24, no. 4, pp. 198–206, Apr. 1981. [Online]. Available: http://doi.acm.org/10.1145/358598.358613

[CNM11] G. Chalkidis, M. Nagasaki, and S. Miyano, "High performance hybrid functional petri net simulations of biological pathway models on cuda," *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, vol. 8, no. 6, pp. 1545–1556, Nov.-Dec. 2011.

[CS89] K. M. Chandy and R. Sherman, "The conditional-event approach to distributed simulation," DTIC Document, Tech. Rep., 1989.

[DBN89] E. J. Derrick, O. Balci, and R. E. Nance, "A comparison of selected conceptual frameworks for simulation modeling," in *Proceedings of the 21st conference on Winter simulation*, ser. WSC '89. New York, NY, USA: ACM, 1989, pp. 711–718. [Online]. Available: http://doi.acm.org/10.1145/76738.76829

[Dij59] E. Dijkstra, "A note on two problems in connection with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[EHU$^+$06] R. Ewald, J. Himmelspach, A. Uhrmacher, D. Chen, and G. Theodoropoulos, "A simulation approach to facilitate parallel and distributed discrete-event simulator development," in *Distributed Simulation and Real-Time Applications, 2006. DS-RT'06. Tenth IEEE International Symposium on*, 2006, pp. 209–218.

[FDP$^+$97] R. M. Fujimoto, S. R. Das, K. S. Panesar, M. Hybinette, and C. Carothers, *Georgia Tech Time Warp (GTW Version 3.1) Programmer's Manual for Distributed Network of Workstations*, 1997.

[Fer94] A. Ferscha, "Concurrent execution of timed petri nets," in *Simulation Conference Proceedings, 1994. Winter*, dec. 1994, pp. 229 – 236.

[Fer99] ——, "Adaptive time warp simulation of timed petri nets," *Software Engineering, IEEE Transactions on*, vol. 25, no. 2, pp. 237–257, Mar/Apr 1999.

[Fly72] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans. Comput.*, vol. 21, no. 9, pp. 948–960, Sep. 1972. [Online]. Available: http://dx.doi.org/10.1109/TC.1972.5009071

[Fre12] A. Freeth, "Cosc460 honours report: A sequential steady-state detection method for quantitative discrete-event simulation," 2012.

[Fuj88] R. M. Fujimoto, "Lookahead in parallel discrete event simulation," DTIC Document, Tech. Rep., 1988.

[Fuj89a] ——, "Parallel discrete event simulation," in *Proceedings of the 21st conference on Winter simulation*, ser. WSC '89. New York, NY, USA: ACM, 1989, pp. 19–28. [Online]. Available: http://doi.acm.org/10.1145/76738.76741

[Fuj89b] ——, "Time warp on a shared memory multiprocessor," DTIC Document, Tech. Rep., 1989.

[Fuj93a] ——, "Parallel and distributed discrete event simulation: algorithms and applications," in *Proceedings of the 25th conference on Winter simulation*, ser. WSC '93. New York, NY, USA: ACM, 1993, pp. 106–114. [Online]. Available: http://doi.acm.org/10.1145/256563.256596

[Fuj93b] ——, "Parallel discrete event simulation: Will the field survive?" *ORSA Journal on Computing*, vol. 5, no. 3, pp. 213–230, 1993.

[Fuj99] R. Fujimoto, "Parallel and distributed simulation," in *Simulation Conference Proceedings, 1999 Winter*, vol. 1, 1999, pp. 122 –131 vol.1.

[Fuj00a] R. M. Fujimoto, "Tutorial: Parallel & distributed simulation systems: from chandy/misra to the high level architecture and beyond," 2000, college of Computing Georgia Institute of Technology Atlanta, GA 30332-0280.

[Fuj00b] ——, *Parallel and Distribution Simulation Systems*, 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 2000.

[FW08] W. Franklin and K. P. White Jr, "Stationarity tests and mser-5: Exploring the intuition behind mean-squared-error-reduction in detecting and correcting initialization bias," in *Simulation Conference, 2008. WSC 2008. Winter*, dec. 2008, pp. 541 –546.

[FXY07] X. Fang, Z. Xu, and Z. Yin, "Distributed processing based on timed petri nets," in *Proceedings of the Third International Conference on Natural Computation - Volume 05*, ser. ICNC '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 287–291. [Online]. Available: http://dx.doi.org/10.1109/ICNC.2007.335

[GHSW05] R. Geist, J. Hicks, M. Smotherman, and J. Westall, "Parallel simulation of petri nets on desktop pc hardware," in *Simulation Conference, 2005 Proceedings of the Winter*, dec. 2005, p. 10 pp.

[Gor69] G. Gordon, *System simulation*, ser. Prentice-Hall series in automatic computation. Prentice-Hall, 1969. [Online]. Available: http://books.google.de/books?id=acEmAAAAMAAJ

[HELU10] J. Himmelspach, R. Ewald, S. Leye, and A. Uhrmacher, "Enhancing the scalability of simulations by embracing multiple levels of parallelization," in *Parallel and Distributed Methods in Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop on*, 2010, pp. 57–66.

[HRD08] K. Hoad, S. Robinson, and R. Davies, "Automating warm-up length estimation," in *Proceedings of the 40th Conference on Winter Simulation*, ser. WSC '08. Winter Simulation Conference, 2008, pp. 532–540. [Online]. Available: http://dl.acm.org/citation.cfm?id=1516744.1516846

[Jef85] D. R. Jefferson, "Virtual time," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 3, pp. 404–425, Jul. 1985. [Online]. Available: http://doi.acm.org/10.1145/3916.3988

[Jen81] K. Jensen, "Coloured Petri Nets and the invariant-method," *Theoretical Computer Science*, vol. 14, pp. 317–336, 1981.

[Jür97] T. Jürgens, "Verteilte simulation von hqpns auf einem netzwerk von workstations," Master's thesis, Universität Dortmund, Fachbereich Informatik, 1997.

[Kau87] F. J. Kaudel, "A literature survey on distributed discrete event simulation," *SIGSIM Simul. Dig.*, vol. 18, no. 2, pp. 11–21, Jun. 1987. [Online]. Available: http://doi.acm.org/10.1145/29497.29499

[KB03] S. Kounev and A. Buchmann, "Performance modeling of distributed e-business applications using queueing petri nets," in *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2003), Austin, Texas, USA, March 6-8, 2003.* Washington, DC, USA: IEEE Computer Society, 2003, pp. 143–155, best-Paper-Award at ISPASS-2003. [Online]. Available: http://www.ispass.org/ispass2003/

[KB06] ——, "SimQPN - a tool and methodology for analyzing queueing Petri net models by means of simulation," *Performance Evaluation*, vol. 63, no. 4-5, pp. 364–394, May 2006. [Online]. Available: http://www.elsevier.com/wps/find/journaldescription.cws_home/505618/description

[KB07] ——, "On the Use of Queueing Petri Nets for Modeling and Performance Analysis of Distributed Systems," in *Petri Net, Theory and Application*, V. Kordic, Ed. Vienna, Austria: Advanced Robotic Systems International, I-Tech Education and Publishing, February 2007. [Online]. Available: http://www.intechopen.com/books/export/citation/BibTex/petri_net_theory_and_applications/on_the_use_of_queueing_petri_nets_for_modeling_and_performance_analysis_of_distributed_systems

[KD07] S. Kounev and C. Dutz, *QPME 1.0 Queueing Petri net Modeling Environment User's Guide*, 2007.

[Ken53] D. G. Kendall, "Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain," *The Annals of Mathematical Statistics*, vol. 24, no. 3, pp. 338–354, 1953. [Online]. Available: http://dx.doi.org/10.2307/2236285

[KL83] W. D. Kelton and A. M. Law, "A new approach for dealing with the startup problem in discrete event simulation," *Naval Research Logistics Quarterly*, vol. 30, Issue 4, pp. 641–658, 1983.

[KNT07] S. Kounev, R. Nou, and J. Torres, "Autonomic qos-aware resource management in grid computing using online performance models," in *Proceedings of the 2nd international conference on Performance evaluation methodologies and tools*, ser. ValueTools '07. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007, pp. 48:1–48:10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1345263.1345325

[Kou05] S. Kounev, *Performance Engineering of Distributed Component-Based Systems - Benchmarking, Modeling and Performance Prediction.* Aachen, Germany: Shaker Verlag, Ph.D. Thesis, Technische Universität Darmstadt, Germany, December 2005, best Dissertation Award from the "Vereinigung von Freunden der Technischen Universität zu Darmstadt e.V.". [Online]. Available: http://www.amazon.de/exec/obidos/ASIN/3832247130/302-7474121-6584807

[Kou06] ——, “Performance modeling and evaluation of distributed component-based systems using queueing petri nets,” *Software Engineering, IEEE Transactions on*, vol. 32, no. 7, pp. 486 –502, july 2006.

[KS09] S. Kounev and K. Sachs, “Benchmarking and Performance Modeling of Event-Based Systems,” *it - Information Technology*, vol. 51, no. 5, September 2009.

[KS11] S. Kounev and S. Spinner, *QPME 2.0 User’s Guide*, 2011.

[KSBB08] S. Kounev, K. Sachs, J. Bacon, and A. Buchmann, “A Methodology for Performance Modeling of Distributed Event-Based Systems,” in *Proceedings of the 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC 2008), Orlando, Florida, USA, May 5-7, 2008.* Washington, DC, USA: IEEE Computer Society, 2008, pp. 13–22, acceptance Rate (Full Paper): 30% Best-Paper-Award-Nomination.

[LAH99] M. C. Lowry, P. J. Ashenden, and K. A. Hawick, “Distributed high-performance simulation using time warp and java,” University of Adelaide, Australia, Tech. Rep., 1999.

[Law06] A. Law, *Simulation Modeling and Analysis (McGraw-Hill Series in Industrial Engineering and Management).* McGraw-Hill Science/Engineering/Math, 2006.

[LL90] Y.-B. Lin and E. Lazowska, “Exploiting lookahead in parallel simulation,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 1, no. 4, pp. 457–469, 1990.

[LNT01] J. Liu, D. M. Nicol, and K. Tan, “Lock-free scheduling of logical processes in parallel simulation,” in *Proceedings of the fifteenth workshop on Parallel and distributed simulation*, ser. PADS ’01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 22–31. [Online]. Available: http://dl.acm.org/citation.cfm?id=375658.375661

[LR12] J. Liu and R. Rong, “Hierarchical composite synchronization,” in *Principles of Advanced and Distributed Simulation (PADS), 2012 ACM/IEEE/SCS 26th Workshop on*, 2012, pp. 3–12.

[LSCD04] J. Lemeire, B. Smets, P. Cara, and E. F. Dirkx, “Exploiting symmetry for partitioning models in parallel discrete event simulation,” in *PADS*, 2004, pp. 189–194.

[Lub89] B. D. Lubachevsky, “Efficient distributed event-driven simulations of multiple-loop networks,” *Commun. ACM*, vol. 32, no. 1, pp. 111–123, 1989.

[Mar90] M. Marsan, “Stochastic petri nets: An elementary introduction,” in *Advances in Petri Nets 1989*, ser. Lecture Notes in Computer Science, G. Rozenberg, Ed. Springer Berlin Heidelberg, 1990, vol. 424, pp. 1–29. [Online]. Available: http://dx.doi.org/10.1007/3-540-52494-0_23

[McC90] M. A. McClarnon, “Detection of steady state in discrete event dynamic systems: An analysis of heuristics.” Master’s thesis, School of Engineering and Applied Science, University of Virginia, Charlottesville, Virginia., Available from University of Viginia Library, lib-lend@virginia.edu, 434-982-3094., 1990.

[Mer11] P. Merkle, “Comparing process- and event-oriented software performance simulation,” Master’s thesis, Karlsruhe Institute of Technology (KIT), Germany, 2011.

[MI04] P. Mahajan and R. Ingalls, “Evaluation of methods used to detect warm-up period in steady state simulation,” in *Simulation Conference, 2004. Proceedings of the 2004 Winter*, vol. 1, dec. 2004, pp. 2 vol. (xliv+2164).

[MN98] M. Matsumoto and T. Nishimura, “Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator,” *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, pp. 3–30, Jan. 1998. [Online]. Available: http://doi.acm.org/10.1145/272991.272995

[MTA$^+$03] H. Matsuno, Y. Tanaka, H. Aoshima, A. Doi, M. Matsui, and S. Miyano, “Biopathways representation and simulation on hybrid functional petri net,” *In Silico Biology*, vol. 3, p. 32, 2003.

[Mur89] T. Murata, “Petri nets: Properties, analysis and applications,” *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541 –580, apr 1989.

[Nan81] R. E. Nance, “The time and state relationships in simulation modeling,” *Commun. ACM*, vol. 24, no. 4, pp. 173–179, Apr. 1981. [Online]. Available: http://doi.acm.org/10.1145/358598.358601

[Nel92] B. L. Nelson, “Statistical analysis of simulation results,” in *Handbook of Industrial Engineering*, G. Salvendy, Ed. New York, NY: Wiley, 1992, ch. 102, pp. 2567–2593.

[Nic88] D. M. Nicol, “Parallel discrete-event simulation of fcfs stochastic queueing networks,” in *Proceedings of the ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems*, ser. PPEALS ’88. New York, NY, USA: ACM, 1988, pp. 124–137. [Online]. Available: http://doi.acm.org/10.1145/62115.62128

[Nic93] ——, “The cost of conservative synchronization in parallel discrete event simulations,” *J. ACM*, vol. 40, no. 2, pp. 304–333, Apr. 1993. [Online]. Available: http://doi.acm.org/10.1145/151261.151266

[Nie99] P. O. Nierstrasz. (1999) Concurrent programming. Lecture. Bern. [Online]. Available: http://scg.unibe.ch/archive/lectures/CP-ConcurrentProgramming/CP-W99.pdf

[NK01] A. Nketsa and N. B. Khalifa, “Timed petri nets and prediction to improve the chandy-misra conservative-distributed simulation,” *Applied Mathematics and Computation*, vol. 120, no. 1-3, pp. 235 – 254, 2001, <ce:title>The Bellman Continuum</ce:title>. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0096300399002441

[NKJT09] R. Nou, S. Kounev, F. Julia, and J. Torres, “Autonomic QoS control in enterprise Grid environments using online simulation,” *Journal of Systems and Software*, vol. 82, no. 3, pp. 486–502, March 2009. [Online]. Available: http://www.sciencedirect.com/science/journal/01641212

[NR91] D. M. Nicol and S. Roy, “Parallel simulation of timed petri-nets,” in *Proceedings of the 23rd conference on Winter simulation*, ser. WSC ’91. Washington, DC, USA: IEEE Computer Society, 1991, pp. 574–583. [Online]. Available: http://dl.acm.org/citation.cfm?id=304238.304325

[Pag95] E. H. Page, “Simulation modeling methodology: principles and etiology of decision support,” Ph.D. dissertation, 1995. [Online]. Available: http://portal.acm.org/citation.cfm?id=923593

[Peg10] C. D. Pegden, "Advanced tutorial: overview of simulation world views," in *Proceedings of the Winter Simulation Conference*, ser. WSC '10. Winter Simulation Conference, 2010, pp. 210–215. [Online]. Available: http://dl.acm.org/citation.cfm?id=2433508.2433531

[PR94] H. Praehofer and G. Reisinger, "Distributed simulation of devs-based multiformalism models," in *In Proc. of AI, Simulation and Planning in High-Autonomy Systems.* IEEE, 1994, pp. 150–156.

[PS10] R. Pasupathy and B. Schmeiser, "The initial transient in steady-state point estimation: Contexts, a bibliography, the mse criterion, and the mser statistic," in *Simulation Conference (WSC), Proceedings of the 2010 Winter*, dec. 2010, pp. 184 –197.

[PT08] V. Pankratius and W. F. Tichy, "International workshop on multicore software engineering (iwmse 2008)," in *ICSE Companion*, 2008, pp. 1051–1052.

[PVM09] P. Peschlow, A. Voss, and P. Martini, "Good news for parallel wireless network simulations," in *Proceedings of the 12th ACM international conference on Modeling, analysis and simulation of wireless and mobile systems*, ser. MSWiM '09. New York, NY, USA: ACM, 2009, pp. 134–142. [Online]. Available: http://doi.acm.org/10.1145/1641804.1641828

[PYM94] K. Pawlikowski, V. Yau, and D. McNickle, "Distributed stochastic discrete-event simulation in parallel time streams," in *Proceedings of the 26th conference on Winter simulation.* Society for Computer Simulation International, 1994, pp. 723–730.

[RMM88] D. A. Reed, A. D. Malony, and B. McCredie, "Parallel discrete event simulation using shared memory," *IEEE Trans. Softw. Eng.*, vol. 14, no. 4, pp. 541–553, Apr. 1988. [Online]. Available: http://dx.doi.org/10.1109/32.4677

[Rob04] S. Robinson, *Simulation: The Practice of Model Development and Use.* John Wiley & Sons, 2004.

[Sac10] K. Sachs, "Performance modeling and benchmarking of event-based systems," Ph.D. dissertation, TU Darmstadt, 2010, sPEC Distinguished Dissertation Award 2011.

[Sch82] L. W. Schruben, "Detecting initialization bias in simulation output," *Operations Research*, vol. 30, pp. 569–590, 1982.

[SKB12] K. Sachs, S. Kounev, and A. Buchmann, "Performance modeling and analysis of message-oriented event-driven systems," *Journal of Software and Systems Modeling (SoSyM)*, pp. 1–25, February 2012. [Online]. Available: http://dx.doi.org/10.1007/s10270-012-0228-1

[SKM12] S. Spinner, S. Kounev, and P. Meier, "Stochastic modeling and analysis using qpme: Queueing petri net modeling environment v2.0," in *Proceedings of the 33rd International Conference on Application and Theory of Petri Nets and Concurrency (Petri Nets 2012)*, ser. Lecture Notes in Computer Science (LNCS), S. Haddad and L. Pomello, Eds., vol. 7347. Springer-Verlag, 6 2012, pp. 388–397.

[Ste91] J. S. Steinman, "SPEEDES: Synchronous Parallel Environment for Emulation and Discrete Event Simulation," in *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, Jan. 1991, pp. 95–101.

[TM06] N. Tiwari and P. Mynampati, “Experiences of using lqn and qpn tools for performance modeling of a j2ee application,” in *CMG-CONFERENCE-*, vol. 1. Computer Measurement Group; 1997, 2006, p. 537.

[vdA96] W. van der Aalst, “Parallel computation of reachable dead states in a free-choice petri net,” 1996.

[VJ03] C. Vidrascu and T. Jucan, “Concurrency-degrees for p/t - nets,” *Sci. Ann. Cuza Univ.*, vol. 13, pp. 91–104, 2003.

[WCS00] K. P. White Jr, M. Cobb, and S. Spratt, “A comparison of five steady-state truncation heuristics for simulation,” in *Simulation Conference, 2000. Proceedings. Winter*, vol. 1, 2000, pp. 755 –760 vol.1.

[Web09] T. Webber, “Reducing the Impact of State Space Explosion in Stochastic Automata Networks,” Ph.D. dissertation, Pontifícia Universidade Católica do Rio Grande do Sul, mar 2009.

[Wel81] P. D. Welch, “On the problem of the initial transient in steady-state simulation,” IBM Watson Research Center, Yorktown Heights, New York, Tech. Rep., 1981.

[Wel83] ——, “The statistical analysis of simulation results,” in *The Computer Performance Modeling Handbook*, 1983.

[WL89] D. B. Wagner and E. D. Lazowska, “Parallel simulation of queueing networks: limitations and potentials,” in *Proceedings of the 1989 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, ser. SIGMETRICS ’89. New York, NY, USA: ACM, 1989, pp. 146–155. [Online]. Available: http://doi.acm.org/10.1145/75108.75388

[WLB88] D. B. Wagner, E. D. Lazowska, and B. N. Bershad, “Techniques for efficient shared-memory parallel simulation,” Dept of Computer Science, Univ. of Washington, Seattle, WA 98195, Tech. Rep. TR-88-04-05, Aug. 1988.

[XUSC99] Z. Xiao, B. Unger, R. Simmonds, and J. Cleary, “Scheduling critical channels in conservative parallel discrete event simulation,” in *Parallel and Distributed Simulation, 1999. Proceedings. Thirteenth Workshop on*, 1999, pp. 20–28.

Acronyms

QN	Queueing Network
PN	Petri Net
PN	Petri Net
CPN	Colored Petri Net
SPN	Stochastic Petri Net
GSPN	Generalized Stochastic Petri Net
CGSPN	Colored Generalized Stochastic Petri Net
HFPN	Hybrid Functional Petri Net
HPN	Hybrid Petri Net
TPN	Timed (Transition) Petri Net
QPN	Queueing Petri Net
HQPN	Hierarchical Queueing Petri Net
FCFS	First-Come-First-Served scheduling strategy
RR	Round Robin scheduling strategy
PS	Processor Sharing scheduling strategy
IS	Infinite Server scheduling strategy
PRIO	Priority Scheduling strategy
RANDOM	Random Scheduling strategy
SRIP	Single Replication In Parallel
MRIP	Multi Replication In Parallel
LP	Logical Process
TWLP	Time Warp Logical Process
LVT	Local Virtual Time
GVT	Global Virtual Time used in optimistic Timewarp algorithm
CMB	Chandy/Misra/Bryant Algoritm
YAWNS	Yet Another Windowing Network Simulator YAWNS protocol is a barrier-based conservative synchronization algorithm introduced by Nicol et al. [Nic93]
CCT	Critical Channel Traversing CCT is an extension to the CMB algorithm

QPME	Queueing Petri Net Modeling Environment
SimQPN	SimQPN is a simulation engine for QPNs which is integrated into QPME
CERN	European Organization for Nuclear Research
CPU	Central Processing Unit
GPU	Graphics Processing Unit
SIMD	Single Instruction Multiple Data
MTW	Moving Time Window
BTB	Breathing Time Buckets
GTW	Georgia Tech Time Warp
LBTS	Lower Bound on the Time Stamp
CMR	Confidence Maximization Rule
MSER	Marginal Standard Error Rule
MSER-5	Marginal Standard Error Rule-5
ASD	Algorithm for a Static Dataset
ADD	Algorithm for a Dynamic Dataset
BPM	Business Process Management
PEPSY-QNS	Performance Evaluation and Prediction SYstem for Queueing NetworkS