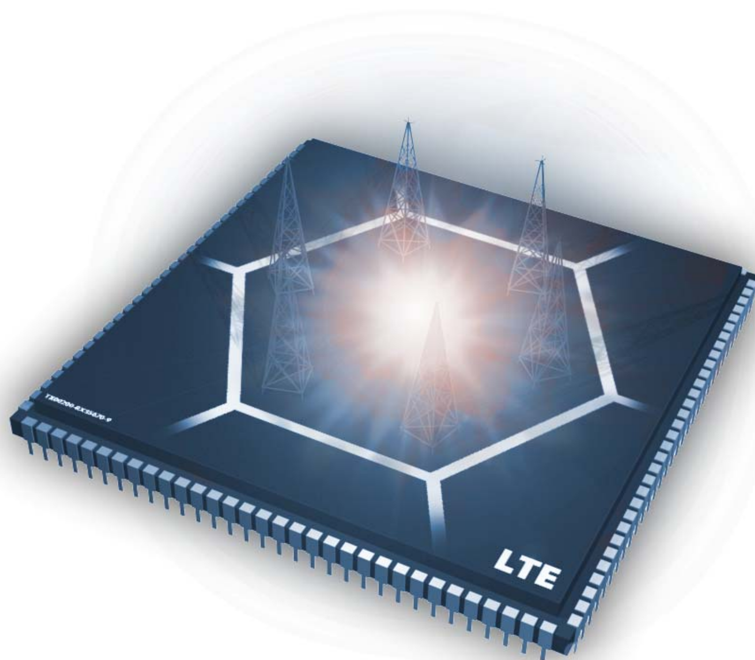
Mid Sweden University
The Department of Information Technology and Media (ITM)
Author: Håkan Andersson
E-mail address: haan0400@student.miun.se
Study programme: M. Sc. in engineering - computer engineering, 300 ECTS
Examiner: Tingting Zhang, tingting.zhang@miun.se
Tutors: Rahim Rahmani, rahim.rahmani@miun.se
Niclas Wiberg, niclas.wiberg@ericsson.com
Scope: 25622 words inclusive of appendices
Date: 2010-05-09



M.Sc. Thesis within
Computer Engineering AV
30 ECTS

Parallel simulation

Parallel computing for high performance LTE radio network simulations

Håkan Andersson

Abstract

Radio access technologies for cellular mobile networks are continuously being evolved to meet the future demands for higher data rates, and lower end-to-end delays. In the research and development of LTE, radio network simulations play an essential role. The evolution of parallel processing hardware makes it desirable to exploit the potential gains of parallelizing LTE radio network simulations using multithreading techniques in contrast to distributing experiments over processors as independent simulation job processes. There is a hypothesis that parallel speedup gain diminishes when running many parallel simulation jobs concurrently on the same machine due to the increased memory requirements. A proposed multithreaded prototype of the Ericsson LTE simulator has been constructed, encapsulating scheduling, execution and synchronization of asynchronous physical layer computations. In order to provide implementation transparency, an algorithm has been proposed to sort and synchronize log events enabling a sequential logging model on top of non-deterministic execution. In order to evaluate and compare multithreading techniques to parallel simulation job distribution, a large number of experiments have been carried out for four very diverse simulation scenarios. The evaluation of the results from these experiments involved analysis of average measured execution times and comparison with ideal estimates derived from Amdahl's law in order to analyze overhead. It has been shown that the proposed multithreaded task-oriented framework provides a convenient way to execute LTE physical layer models asynchronously on multi-core processors, still providing deterministic results that are equivalent to the results of a sequential simulator. However, it has been indicated that distributing parallel independent jobs over processors is currently more efficient than multithreading techniques, even though the achieved speedup is far from ideal. This conclusion is based on the observation that the overhead caused by increased memory requirements, memory access and system bus congestion is currently smaller than the thread management and synchronization overhead of the proposed multithreaded Java prototype.

Keywords: Parallel Simulation, PDES, LTE, Radio Network Simulation, Multithreading, Java, Concurrency.

Acknowledgements

I would like to thank Gunnar Bark (Manager at Ericsson Research for radio network algorithms and performance) and Niclas Wiberg (Expert within radio modelling and simulation at Ericsson Research), for giving me the opportunity to do my thesis at the Ericsson Research site in Linköping. It has been a very interesting and developing experience to work with cutting edge LTE research and contributing to the development of one of Ericsson's most important simulators designated for LTE radio network simulation.

I would also like to thank my supervisors at Ericsson, Niclas Wiberg and research engineer Kristina Jersenius, for your wide knowledge and enthusiasm within the field of simulation, parallel programming and Java. Our regular meetings and discussions regarding the simulation models, design and associated technology have really helped to improve the quality of this work. I also really appreciate that Kristina supplied me with scenario parameters in order to improve the quality and credability of experimental results.

I thank Rahim Rahmani, my supervisor at Mid Sweden University, for the time you have spent reviewing my thesis and supplying me with feedback and comments considering the contents and layout of this report.

I also thank everyone who answered the questionnaire related to task-oriented framework usability and transparency, both inside and outside of Ericsson.

Finally, I would like to express my gratitude to everyone at Ericsson Research in Linköping who have contributed with valuable discussions and positive attitudes regarding this work. Also huge thanks to family and friends who have supported me throughout my whole education and this thesis work.

Table of Contents

Abstract ii

Acknowledgements iii

Terminology vii

1 Introduction 1
1.1 Background and problem motivation 1
1.2 Overall aim 2
1.3 Scope 3
1.4 Concrete and verifiable goals 4
1.5 Outline 5
1.6 Contributions 7

2 3G long-time evolution (LTE) 8
2.1 Overview of LTE technology 8
2.2 Orhogonal frequency division multiplexing (OFDM) 9
2.2.1 LTE OFDM in the downlink 11
2.2.2 LTE SC-FDMA in the uplink 11
2.3 Multiple antenna techniques 11
2.4 LTE duplex schemes 12
2.5 LTE frame and sub-frame structure 13
2.6 LTE channels 15
2.6.1 Logical channels 15
2.6.2 Transport channels 15
2.6.3 Physical channels 15

3 Simulation model 18
3.1 Related simulation platforms and technologies 18
3.2 Ericsson Research LTE simulation platform 19
3.2.1 Events and timers 21
3.2.2 LTE physical layer models 22
3.2.3 Simulation output 23
3.2.4 Previous profiling results of the simulation environment 24

4 Parallel Computing 25
4.1 Processor evolution and parallel architectures 25
4.2 Fundamental components of parallel processing 26

4.3 Performance metrics for parallel computing 28
4.3.1 Speedup 28
4.3.2 Efficiency 29
4.3.3 Amdahl's law 29
4.4 Parallel programming 30
4.4.1 Multithreading 31
4.4.2 Synchronization 31
4.5 Algorithm analysis and design 32
4.5.1 Data and control dependency 32
4.5.2 Granularity and regularity 33
4.6 Design patterns for parallel computing 34
4.7 Java technologies and frameworks for parallel computing 35
4.7.1 Java concurrency API 35
4.7.2 Parallel Java (PJ) 36
4.7.3 Java Parallel Processing Framework (JPPF) 36
4.7.4 JOMP 36
4.7.5 MANTA compiler 36
4.7.6 Javab compiler 37
4.8 Alternative technologies for parallel and concurrent computations 37

5 Research in parallel simulations 39
5.1 Parallel simulation approaches 39
5.2 Evolution of parallel discrete-event simulation 42

6 Methodology 44
6.1 Experimental methodology 44
6.2 Performance experiments and evaluation criteria 45
6.3 Simulation scenarios 47
6.4 Environment and physical resources 48
6.5 Verification of program correctness 48
6.6 Evaluation of software design transparency and usability. 49

7 Design 50
7.1 Analysis of requirements and design considerations 50
7.2 Performance bottlenecks in current design 51
7.3 Analysis of the LTE physical layer model 53
7.4 Data and control dependency analysis 54
7.5 Task-oriented concurrency framework 55
7.6 Task management and asynchronous execution 57
7.7 Orchestration of tasks and data in LTE physical layer 60
7.8 Preserving deterministic behavior 62

7.8.1 Definition of deterministic simulation 62
7.8.2 Synchronization and ordering of log events 62
7.8.3 Verification of deterministic behavior......................... 65

8 Results 68
8.1 Performance gain of multithreaded simulation.................... 68
8.2 Execution time per simulation job when executing multiple simulation jobs.. 70
8.3 Comparison of parallel jobs and an ideal multithreaded simulator... 72
8.4 Implementation transparency and task-oriented framework usability 74
8.5 Verification of deterministic behavior................................... 75

9 Conclusions 76
9.1 Software design evaluation.. 76
9.1.1 Strengths and weaknesses of the task-oriented framework 76
9.1.2 Evaluation of questionnaire considering implementation readability and transparency 77
9.1.3 Evaluation of deterministic behavior 77
9.2 Evaluation of multithreaded prototype performance.......... 78
9.2.1 Anomalies between systems.. 78
9.2.2 Multithreading performance and multithreading overhead 78
9.3 Comparison of job parallelization and multithreading 80
9.4 Recommendations for future work 81

References.. 83

Appendix A: System specifications.. 89

Appendix B: Questionnaire for evaluation of readability and usability of parallel constructs.. 90

Appendix C: UML class diagram for task-oriented framework 93

Appendix D: Summary of questionnaire results..................................... 94

Terminology

Acronyms

2G	Second Generation
3G	Third Generation
3GPP	Third Generation Partnership Project
AMPS	American Mobile Phone Service
API	Application Programming Interface
BLEP	Block Error Probability
CDMA	Code Division Multiple Access
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DFTS-FDMA	Discrete Fourier Transform Spread Frequency Division Multiple Access
eNodeB	Enhanced Base Station.
ETSI	European Telecommunication Standards Institute
FCFS	First-Come-First-Served
FFD	Frequency Division Duplex
FDMA	Frequency Division Multiple Access
FIFO	First-In-First-Out
GPRS	General Packet Radio Services
GSM	Global System for Mobile communications
HARQ	Hybrid Automatic Repeat Request

ISI	Inter-Symbol Interference
JDK	Java Development Kit
JRE	Java Run-time Environment
LTE	Long Time Evolution
MAC	Media Access Control
MIMO	Multiple-Input Multiple-Output
NMT	Nordic Mobile Telephony
OFMD	Orthogonal Frequency Division Multiplexing
OFDMA	Orthogonal Frequency Division Multiple Access
PBCH	Physical Broadcast Channel
PCFICH	Physical Control Format Indicator Channel
PDCCH	Physical Downlink Control Channel
PDES	Parallel Discrete Event Simulation
PDSCH	Physical Downlink Shared Channel
PE	Processing Element
PHICH	Physical Hybrid-ARQ Indicator Channel
PMCH	Physical Multicast Channel
PRACH	Physical Random Access Channel
PUCCH	Physical Uplink Control Channel
PUSCH	Physical Uplink Shared Channel
RCR	Run-time length of task to communication overhead ratio
RLC	Radio Link Control

SC-FDMA	Single Carrier Frequency Division Multiple Access
SIMO	Single-Input Multiple-Output
SINR	Signal-to-Interference-plus-Noise Ratio
SIR	Signal-to-Interference Ratio
SMP	Shared-memory-multiprocessor
TDD	Time Division Duplex
TDMA	Time Division Multiple Access
TTI	Transmission Time Interval
UE	User Equipment
UoE	Unit of Execution
UMTS	Universal Mobile Telecommunications System
UTRA	Universal Terrestrial Radio Access
WCDMA	Wideband Code Division Multiple Access

Mathematical notation

Symbol	Description
S_N	Speedup of a parallel homogenous system with N processors.
T_1	The execution time for a program on a single processor.
T_N	The execution time of a program on N processors.
E_N	Efficiency of a parallel homogenous system given in percent.
η	Fraction of an algorithm or program to be executed sequentially.
$(1-\eta)$	Fraction of an algorithm or program to be executed in parallel.
R	Actual computation time when executing a task.
C	Amount of execution time due to communication overhead when executing a task.
ε	Error detecting code or checksum.
$CRC_N(\varepsilon, x)$	N-polynomial cyclic redundancy checksum function that computes a checksum from an accumulated checksum ε and data x.
e_n	The n:th event in a sequence of ordered events (e_1 , e_2 , e_3 , …) where e_n and e_{n+1} corresponds to events at logical time t_n and t_{n+1} respectively, such that $t_n \leq t_{n+1}$.

1 Introduction

Mobile communication systems have, since they were first introduced in 1946, evolved into global systems, enabling not only traditional telephony, but also advanced data communication services. Mobile communication systems now form part of the everyday life for almost half of the world's population. Developing mobile technologies has also emerged from being a regional or national concern to becoming a complex task undertaken by global standards-developing organizations such as *Third Generation Partnership Project* (3GPP) [1]. A continuously growing demand on mobile services places higher demands on future research and technical development within the area of cellular communication.

1.1 Background and problem motivation

Radio access technologies for cellular mobile networks are continuously being evolved to meet the future demands for higher data rates and lower end-to-end delays. Currently, evolutions of the *third generation* (3G) systems, so-called 3G *Long Term Evolution* (LTE) cellular systems, are being developed by Ericsson and others and will be commercially available in 2010 [1].

In the research and development of LTE, radio network simulations play an essential role in estimating the system and user performance of entire systems or specific radio network functions. The higher bandwidths, larger number of users and more advanced signal processing of LTE requires more extensive simulations, which takes both time and computer resources. Since the computer processing trend is heading towards parallel processing techniques due to the parallel nature of modern desktop-computer multi-core processors [2] [3], it is of paramount interest to exploit the potential gains of parallelizing radio network simulations.

This master thesis study focuses on multithreaded parallel simulation rather than distributed parallel computing models. The reason for this is that most simulation studies consist of many independent simulation jobs, which makes it fairly easy to use distributed computing

environments, simply by running independent jobs on different machines. Attempts have been made at Ericsson to run several independent simulation jobs concurrently on the same machine [4], thus in the optimal case dedicating one simulation process to each core. However it is possible that the parallel gain diminishes if several jobs are executed in this way, due to limitations of processor cache, extensive memory access, limitations of the system bus bandwidth and race conditions between processes. Identifying tasks and algorithms in the existing model that can be run concurrently is considered a convenient first step towards introducing parallel computing concepts to the LTE simulation platform.

In order to ease the future work of the LTE simulator developers at Ericsson, it is also desirable to investigate the possibilities of introducing parallelism as transparently as possible, thus preserving the current system architecture and hiding parallel implementation details.

1.2 Overall aim

The overall aim of this thesis is to obtain an indication as to whether parallelization of the current simulator platform by means of multithreading technology is possible and to determine what gains in performance and thereby reduction in execution time such modifications may have. A successful implementation would result in shorter simulation times due to more efficient utilization of the client system's processing capacity as well as increased scalability. Shorter simulation times will in turn improve the efficiency of radio communication research considering LTE simulation experiments and algorithm evaluation. This would also make it possible to compute more accurate and complex models involving increased number of entities considering a fixed simulation time frame. As long as the development of multi-core *central processing units* (CPUs) is still going in the direction of increasing the number of processor cores [2] [3], the software will also be well-adapted for more sophisticated parallel processing hardware architectures in the foreseeable future.

Independently to the outcome of the prototype implementation, this thesis is likely to contribute with valuable information and conclusions regarding the difficulties, drawbacks and limitations regarding parallelization of existing sequential, event-driven, deterministic

simulation applications in general and parallelization of user-centric radio network simulations in particular.

1.3 Scope

The theoretical part within this report covering 3G evolution and 3GPP LTE technologies is restricted to only introducing the reader within the field of telecommunications and radio communication technology related to 3GPP LTE. There is absolutely no intention to create a survey covering all aspects of LTE technology. For readers who require more exploratory descriptions regarding this subject there are other more illustrative resources that cover the technology and concepts of 3G and LTE such as *3G Evolution* by E. Dahlman et al. [1]. Instead, this part of the thesis presents an overview of the technologies that are vital to understand in order to follow the reasoning in this thesis and understand the simulation model.

Theory about parallel computing in this report is restricted to only clarifying the fundamentals of parallel processing and in describing simple methods regarding how to estimate performance gains of parallel processing. The diversity of parallel hardware architectures, processing networks and their specific features will not be covered within this report. Only performance metrics associated to homogenous *multiple-instruction, multiple-data* (MIMD) parallel architectures with shared memory will be considered as the vast majority of multi-core processors available today have symmetric cores [5].

The simulation platform developed by Ericsson Research for simulating multi-cell radio networks is built for deterministic event-driven simulation. The simulation model includes multiple cells, users, base stations and antennas. It also contains complex algorithms modeling data communication, protocol layers, radio wave propagation and interference [6]. The complexity and detail of the simulation model addresses the need to restrict this work to only focussing on key parts of the simulation model and the Java™ [7] simulator application.

Earlier results obtained when profiling and optimizing the LTE simulation platform have indicated that the major portion of the total computation-time for multi-cell LTE is spent within the physical layer models when simulating detailed and highly accurate models, as stated

by L. Zhang [4]. The LTE physical layer model involves transmission between base stations and mobile user equipment, modeling signal modulation, propagation and interference. This work will be restricted to primarily investigating the possibility to modify and adapt the physical layer model design for parallel multithreaded execution to work as a prototype for evaluation. The proposed design in this thesis will focus on the simulator environment constructs, data dependencies and data flow rather than an analysis and evaluation of the mathematical models currently used at Ericsson for modeling physical entities and protocols.

This work is also restricted to only evaluating and using technologies intended for parallel computing on shared-memory multi-core processors. Distributed solutions such as grid-computing or specialized multi-processor systems such as super-computers will not be considered within the scope of this thesis, they will merely be mentioned.

Profiling measurements performed within the scope of this work and the results obtained from these will be restricted to being based on the output from one profiling tool only, supplied by Ericsson Research. However, a renowned, well-tested and publicly accepted tool for Java profiling will be used.

1.4 Concrete and verifiable goals

The difficulties within the problem domain comprehending and analyzing a very large and complex sequential application and modifying it to suit parallel computations, while still preserving the correctness and reliability of the original software will be dealt with. A sequentially executed deterministic simulation model is not easily converted in order to perform parallel calculations as the next simulation state is derived from the current state. Hence, the order of interaction, communication and results from parallel calculations must be synchronized and ordered so as to be aligned with the simulation time (logical time). This can be verified by comparing the system output from the current verified platform release and the implemented parallel prototype.

The minimum requirements for the theoretical part of this work are that the following research areas and techniques are analyzed:

- Fundamentals of 3GPP Long Time Evolution (LTE) radio network technology, primarily physical layer technology. A survey covering 3G evolution and LTE should be created to give a basic theoretical background of LTE simulation.

- Briefly describe the physical layer simulation models used at Ericsson and provide detailed information regarding mechanisms that are vital to understand this work.

- Current research within the area of parallel simulation and parallel computing, that can be related to the problem domain of this thesis.

- Standards, frameworks, external libraries and tools for parallel programming in Java™.

The minimum requirements for the practical part of this work are that the following are fulfilled:

- Implement a multithreaded prototype of the LTE physical layer model that is capable of utilizing desktop multiprocessor architectures. Maximize transparency for developers considering implementation and algorithm complexity.

- Verify that the implemented prototype is deterministic and produce the same result as the non-parallel simulation platform for the same input parameters.

- Carry out performance measurements in order to be able to compare the performance gains of the multithreaded prototype in contrast to the sequential version.

- Carry out performance measurements in order to be able to compare the gains of multithreading compared to distributing work on several independent processes executing concurrently.

1.5 Outline

This report contains a theoretical part which is composed of chapters 2, 3, 4 and 5: “3G and Long-Time Evolution (LTE)”, “Simulation model”, “Parallel computing” and “Research in parallel simulations” respectively.

Chapter 2 aims at providing the reader with a brief introduction to the evolution of wireless radio network communication and mobile communication, in particular 3G and *Long-Time Evolution* (LTE) standards and related technologies. If the reader is already familiar with these concepts this chapter may be missed out. For others it may provide som clarification regarding the technology, algorithms and physical concepts that are part of the simulation model.

Chapter 3 aims at clarifying why simulation is an important part of modern mobile communication research and presents a basic description regarding the manner in which how the simulation model and platform have been designed. This is required in order to understand the more detailed analysis in chapter 4. This chapter also summarizes current trends and breaktroughs within the field of parallel simulation research.

In chapter 4, a survey on parallel computing and associated technologies is presented. Java technologies have been particulary considered. This chapter intends to provide the reader with an overview of possible technologies that may be used to solve the problem at hand, but also to illustrate which frameworks, tools and techniques were considered before this work was conducted. Theoretical tools to analyze and evaluate parallel algorithms are also presented in this chapter.

In chapter 5, a summary of research within the field of parallel simulation, particularly *parallel discrete event-driven simulation* (PDES) is described.

In chapter 6, the evaluation metrics and methods used to evaluate the performance gains of multithreading techniques in LTE simulation is presented as well as descriptions of the scenarios used for experimental evaluation.

The second part of this thesis consists of a practical part including chapter 7, “Design” and chapter 8, “Results”.

In chapter 7 the approach to introduce parallelism in LTE radio network simulation is presented. This chapter contains an analysis of the technical requirements and elaborates on structural design and parallel

programming concepts used in order to determine algorithms and design patterns that fit the stated requirements.

Chapter 8 presents the results obtained by profiling the multithreaded prototype implementation and comparing it to the performance of the current release of the simulation platform.

Chapter 9, "Conclusions" presents an evaluation of the work conducted in this thesis in addition to personal comments and analytic observations. A recommendation for future improvements and research within parallel computing for event-driven user-centric radio network simulations concludes this chapter.

1.6 Contributions

The sequential simulation platform, simulation model and its structural design, algorithms and source code was contributed by and is the property of Ericsson Research. This project has contributed to the software design by reconstructing and adding functionality to an already existing simulator environment by introducing parallel programming design concepts and Java associated implementations through a transparent task-oriented framework. The task-oriented framework is independent from the simulator and usages outside the area of simulation might be found for this framework in the future.

This work has contributed to Ericsson Research by serving as a pre-study with regards to how to utilize the computational power of modern multi-core systems in the most efficient way. Hopefully, the outcome of this work may serve as an aid in decision making when considering redesign or development of new parallel discrete event simulators.

This work has also contributed to research within the field of parallel programming and parallel discrete event simulation as a case study of the strengths, weaknesses in addition to actual speedups achievied when accommodating a sequential event-driven simulator for multi-processor execution.

2 3G long-time evolution (LTE)

The cellular technologies specified by the *Third Generation Partnership Project* (3GPP) are the most widely developed in the world. These technologies are commonly divided into generations, ranging from the first generation of communication systems including the analog *Nordic Mobile Telephony* (NMT) targeting only voice services, to *second generation* (2G) technologies such as the *Global System for Mobile communications* (GSM) and *General Packet Radio Services* (GPRS), to modern *third generation* (3G) systems offering higher bandwidth services through a higher-bandwidth radio interface called *Universal Terrestrial Radio Access* (UTRA). 3G mobile telecommunication is based on the *wideband code division multiple access* (WCDMA) air interface and packet data in 3G is handled by technologies known as *enhanced uplink* and *High-Speed Downlink Packet Access* (HSPDA) technology (jointly referred to as HSPA, short for *High-Speed Packet Access*). When 3G was developed, internationalization of cellular standardization also became a reality and 3G is now handled in 3GPP.

The latest step within the development of 3GPP is an evolution of 3G into an evolved radio access referred to as *Long-Term Evolution* (LTE) and evolved packet access core network architecture in the *System Architecture Evolution* (SAE). LTE and SAE are planned to be widely deployed in 2010 [1].

2.1 Overview of LTE technology

The research and development of LTE is driven by an increasing demand for higher end-user data transfer rates and the importance of low delay, in addition to the normal capacity and peak data rate requirements. Spectrum flexibility and maximum commonality between *Frequency Division Duplex* (FDD) and *Time Division Duplex* (TDD) solutions were also identified as high priority requirements. To achieve these goals LTE has introduced a number of new technologies when compared to previous cellular systems.

There is no requirement for the LTE radio interface to be backward compatible with WCDMA and HSPA, which makes it possible to design

the LTE radio interface from scratch, purely optimized for IP-transmissions. However, LTE has to support spectrum flexibility as operators obtain more and more scattered spectrums, spread over different bands with different contiguous bandwidths. LTE has to be able to operate in all these bands and with the bandwidths that are available to the operator. However, due to costly filter designs, LTE is targeted to operate in spectrum allocations from roughly 1 to 20 MHz.

The physical layer of LTE conveys both data and control information between an *enhanced base station* (eNodeB) and mobile *user equipment* (UE). The LTE physical layer employs some advanced technologies that are new to cellular applications. These include *Orthogonal Frequency Division Multiplexing* (OFDM), described in chapter 2.2 and *Multiple Input Multiple Output* (MIMO) data transmission which is described in chapter 2.3.

LTE is introduced in resemblance with an evolved core network known as *System Architecture Evolution* (SAE) in order to enable the improved performance to be achieved. System functions such as: user charging systems, authentification, service setup etc. are not really part of the radio access network functions, but are required by the radio access technology. These functions are usually jointly referred to as the *core network functions* primarily used by the operator. The SAE offers many advantages over previous topologies and systems used for cellular core networks, see E. Dahlman et al. for details [1].

2.2 Orhogonal frequency division multiplexing (OFDM)

Orthogonal Frequency Division Multiplexing (OFDM) has been adopted as the signal bearer technology for LTE [1]. In addition, two associated access schemes are used: *Orthogonal Frequency Division Multiple Access* (OFDMA) used on the downlink and *single carrier DFT-spread OFDM* (DFTS-OFDM) also known as *Single Carrirer Frequency Division Multiple Access* (SC-FDMA) on the uplink [8].

Previous cellular systems have used single carrier modulation schemes almost exclusively. Transmission by means of OFDM can, instead, be viewed as a kind of multi-carrier transmission which breaks the available bandwidth into many narrower sub-carriers and transmits the data in parallel streams [8]. OFDM transmission uses a large number of

these close spaced sub-carriers that are modulated using low rate data modulation, for example *quadrature amplitude modulation* (QAM). Normally these signals would be expected to interfere with each other, but this is avoided by making the signals orthogonal to each other by having the carrier spacing equal to the reciprocal of the symbol period. The result of this is that there is no mutual interference between the different signals. When the signals are demodulated they will have a whole number of cycles in the symbol period and their contribution will sum to zero. In other words there is no interference contribution [9].

The data transmitted is split across all the carriers and if some of the carriers are lost due to multi-path distortion effects, the data can be reconstructed by using error correction techniques. Having data carried at a low rate across all carriers also means that the effects of reflections and inter-symbol interference can be overcome [1].

The actual implementation of the OFDM technology is different between the downlink (i.e. from *eNodeB* to UE) and the uplink (i.e. from UE to *eNodeB*) as a result of the different requirements between the two directions and the equipment at either end. However OFDM was chosen as the signal bearer format for LTE as it enables high data bandwidths to be transmitted efficiently while still providing a high degree of resilience to reflections and interference. In addition, OFDM can be used in both *frequency division duplex* (FDD) and *time division duplex* (TDD) formats which are key concepts for the LTE standard. This becomes an additional advantage of OFDM as a modulation technique [1].

The choice of bandwidth for LTE is tightly coupled with OFDM as its influences a variety of system design decisions, including the number of carriers that can be accommodated in the OFDM signal and in turn this influences other elements including, for example, the symbol length.

OFDM provides resilience to multi-path delays and spread. However it is still necessary to implement methods of adding resilience to the system in order to overcome *inter-symbol interference* (ISI). In areas where ISI is expected, this is avoided by inserting a guard period into the timing at the beginning of each data symbol. This makes it possible to copy a section from the end of the symbol to the beginning. This is known as the *cyclic prefix* (CP), see E. Dahlman et. al for details [1].

2.2.1 LTE OFDM in the downlink

The OFDM signal used in LTE consists of a maximum of 2 048 different sub-carriers that are spaced 15 kHz apart. Although it is mandatory for the mobile user equipment to have a capability to be able to receive all sub-carriers, not all are required to be transmitted by the *eNodeB* which only must be able to support transmission of 72 sub-carriers. By this means, all mobiles will be able to talk to any *eNodeB*. Within the OFDM signal it is possible to choose between three types of QAM modulation: *phase-shift keying* (QPSK) which is able to represent 2 bits per symbol, 16QAM which is able to represent 4 bits per symbol and 64QAM which is able to represent 6 bits per symbol. QPSK is the slowest modulation method in relation to data transfer rate, but does not require such a large *signal-to-interference-and-noise ratio* (SINR). Only when there is a sufficient SINR can the higher modulation formats be used.

In the downlink, the sub-carriers are split into resource blocks. This enables the system to be able to divide the data across a fixed number of sub-carriers. Resource blocks utilize 12 sub-carriers, regardless of the overall LTE signal bandwidth and cover one slot in the LTE time frame, further described in section 2.5. This actually means that different LTE signal bandwidths will have different numbers of resource blocks [1].

2.2.2 LTE SC-FDMA in the uplink

For the LTE uplink, another OFDM-based technology is used, called *single-carrier frequency division multiple access* (SC-FDMA) or *single carrier DFT-spread OFDM* (DFTS-OFDM). The reason for this is that the RF power amplifier that transmits the radio frequency signal from the UE via the antenna to the *eNodeB* is the highest power consuming item within the mobile device. Hence, it is necessary that it operates in as efficient mode as possible to maximize battery life-time, which can be significantly affected by the form of radio frequency modulation and the signal format. SC-FDMA is a hybrid format that combines the low peak-to-average power ratio offered by single-carrier systems, with the multi-path interference resilience and flexible sub-carrier frequency allocation that OFDM provides [1].

2.3 Multiple antenna techniques

One of the main problems that previous telecommunications systems have faced is that of multiple signals arising from the many reflections

that are encountered. By using multiple antennas and *Multiple Input Multiple Output* (MIMO) antenna processing, also known as *spatial multiplexing*, these additional signal paths can, instead, be used to achieve improved system performance, improved system capacity (more users) and improved coverage (possibility of larger cells) as well as improved service provisioning, for example higher per-user data rates [1]. Multiple antennas may also be used to provide additional diversity against fading on the radio channel or shape the overall antenna beam in a certain way, for example to maximize the overall antenna gain in the direction of the target receiver/transmitter or to suppress specific dominant interfering signals (also known as beam-forming).

Using multiple antennas at both the transmitter and the receiver can be seen as a tool to further improve the SINR and/or achieve additional diversity against fading. In the general case of N_T transmit antennas and N_R receive antennas, the receiver SNR can be made to increase in proportion to the product $N_T \times N_R$. This enables a corresponding increase in the achievable data rates, assuming that data rates are power limited rather than bandwidth limited. In the bandwidth-limited case, MIMO techniques can, instead, increase the data rates by means of *spatial multiplexing*, where multiple parallel data streams are sent between a transmitter and a receiver.

MIMO schemes using 2×2, 4×2 and 4×4 antenna matrices are considered for LTE. While it is relatively easy to add further antennas to a base station, the same is not true for mobile handsets, where the dimensions of the user equipment limit the number of antennas which should be placed at least a half wavelength apart [1].

2.4 LTE duplex schemes

There are two forms of duplex schemes in LTE which enables uplink and downlink transmission: *frequency division duplex* (FDD) and *time division duplex* (TDD) [1]. FDD uses two channels, one for the transmitter and one for the receiver and enables simultaneous uplink and downlink transmission. TDD uses one frequency or channel, but allocates different time slots for transmission and reception.

LTE has been defined to accommodate both a paired spectrum for *frequency division duplex* (FDD) and an unpaired spectrum for *time division duplex* (TDD). It is anticipated that both LTE FDD and LTE TDD will be widely deployed, as each form of the LTE standard has its own advantages and disadvantages from which decisions can be made regarding which format to adopt dependent upon the particular application. LTE FDD is anticipated to form the migration path for current 3G services, most of which use FDD paired spectrums. However, there has been an additional emphasis on including TDD LTE using unpaired spectrums. TDD LTE is seen as providing the evolution or upgrade path for TD-SCDMA. In view of the increased level of importance being placed upon LTE TDD, it is planned that user equipments will be designed to accommodate both FDD and TDD modes [1].

2.5 LTE frame and sub-frame structure

To maintain synchronization and for the LTE system to manage the different types of information that must be carried between the base station and the user equipment, an LTE time domain structure has been defined. Figure 1 illustrates the high-level time-domain structure for LTE transmission which consists of 10 ms radio frames that in turn consists of ten equally sized sub-frames of length 1 ms.

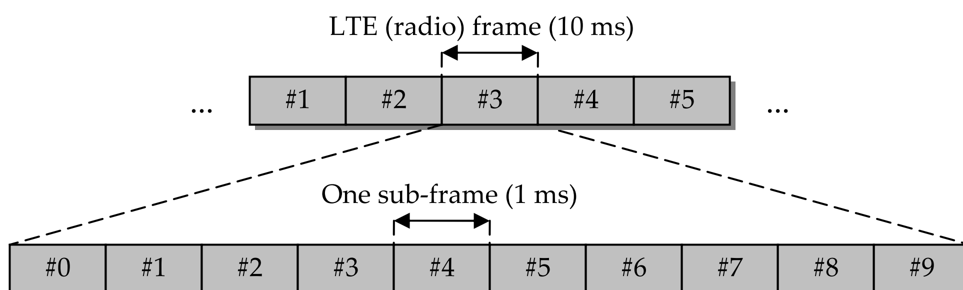


Figure 1: LTE generic high-level time domain structure [1].

Within one carrier, the different sub-frames of an LTE radio frame can be used either for downlink transmission or for uplink transmission. For FDD, this implies an operation in a paired spectrum and that all sub-frames of a carrier are either used for downlink transmission or uplink transmission as illustrated in Figure 2.

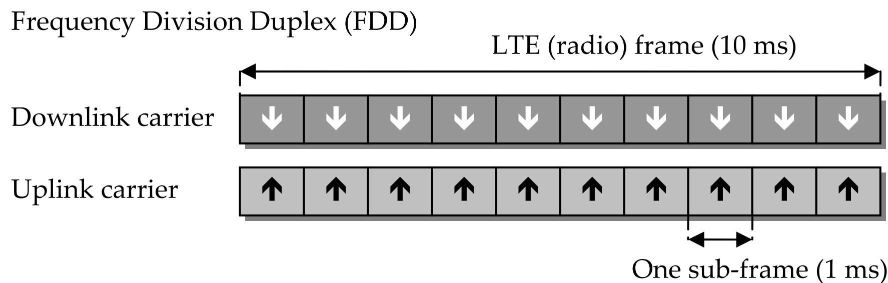


Figure 2: Generic LTE frame structure, also known as *Type 1* for either FDD or TDD duplex modes [1].

In the case of TDD operation in an unpaired spectrum, the first and sixth sub-frames of each frame are always assigned for downlink transmission while the remaining sub-frames can be flexibly assigned either for downlink or uplink transmission. The motivation behind this predefined assignment is that these sub-frames include the LTE synchronization signals that are used for cell-search and neighbor-cell search. Flexible assignment of sub-frames in the case of TDD allows for different asymmetries in terms of the amount of sub-frames assigned for downlink and uplink transmission respectively, as illustrated in Figure 3 [1].

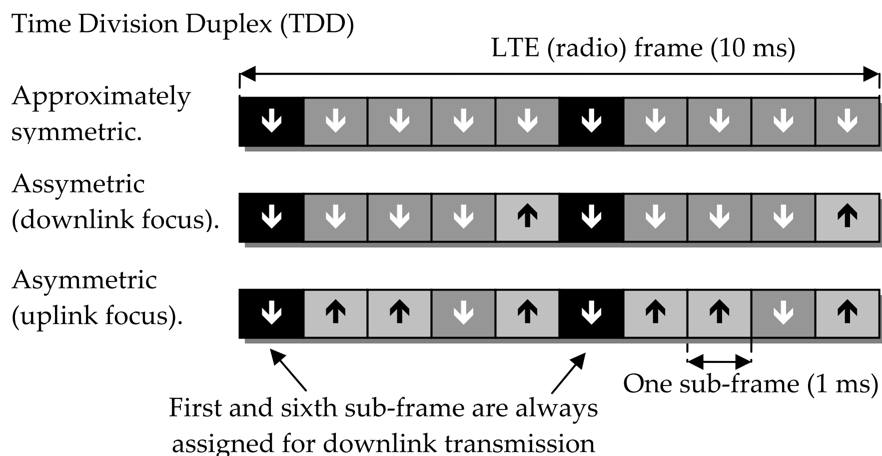


Figure 3: Examples of downlink/uplink assignment using TDD and LTE frame structure *Type 2*. Note that TDD also can be used for *Type 1* frames [1].

2.6 LTE channels

To transport data across the LTE radio interface, various channels are used to segregate the different types of data and allow them to be transported across the radio access network in an orderly fashion. There are three main categories into which the various data channels may be grouped: logical channels, transport channels and physical channels [1].

2.6.1 Logical channels

The *medium access control* (MAC) layer handles logical-channel multiplexing, *hybrid automatic repeat requrest* (HARQ) retransmissions and uplink and downlink scheduling. The MAC offers services to the *radio link control* (RLC) in the form of logical channels. A logical-channel is defined by the type of information that is carried by the channel and is generally classified as a control channel, used for transmission of control and configuration information, or as a traffic channel used for the user data [1].

2.6.2 Transport channels

From the physical layer, the MAC layer uses services in the form of transport channels which are defined by how and with what characteristics the information is transmitted over the radio interface. Data on a transport channel is organized into transport blocks and in each *transmission time interval* (TTI), at most one transport block of a certain size is transmitted over the radio interface. However, using spatial multiplexing, there can be up to two transport blocks per TTI. Each transport block is associated with a transport format that specifies how the transport block is to be transmitted over the radio interface: transport block size, modulation scheme, antenna mapping etc.

Part of the MAC functionality is the multiplexing of logical channels and mapping of the logical channels to the appropriate transport channels. The *downlink shared channel* (DL-SCH) and uplink shared channel (UL-SCH) are the main downlink and uplink transport channels [1].

2.6.3 Physical channels

The *physical layer* (PHY) is responsible for coding, physical-layer hybrid-ARQ processing (retransmission), modulation, multi-antenna processing and mapping of the signal to the appropriate physical time-frequency

resources. The physical layer also handles mapping of transport channels to physical channels. Figure 4 and Figure 5 shows examples of how logical channels are mapped to transport channels and how transport channels in turn are mapped to physical channels for the downlink and uplink respectively [1].

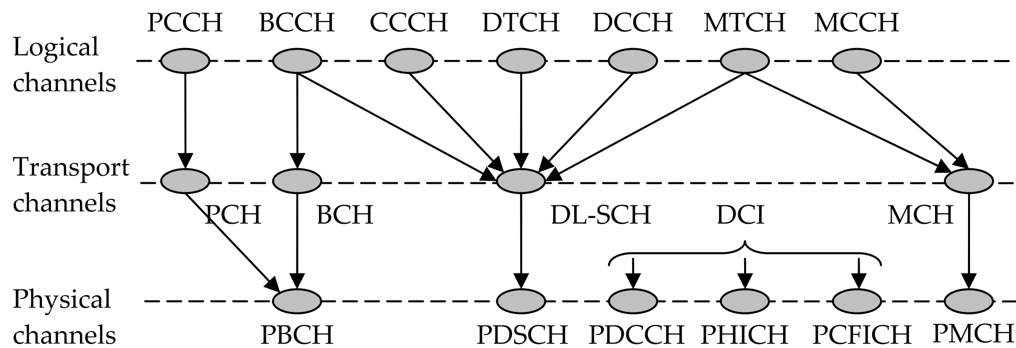


Figure 4: Downlink channel mapping [1].

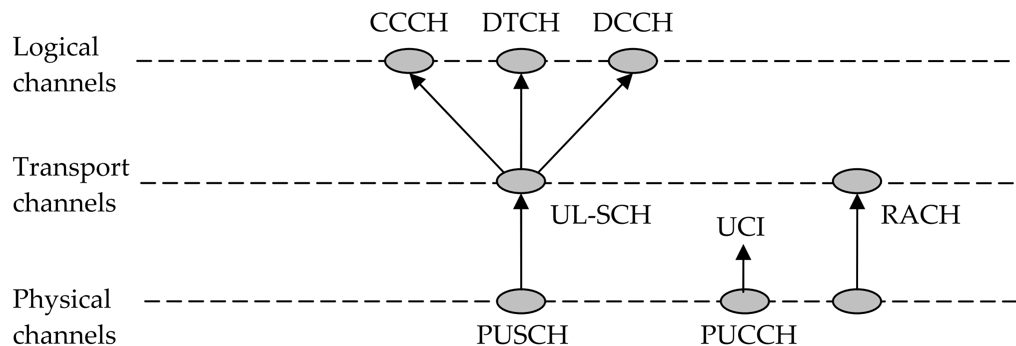


Figure 5: Uplink channel mapping [1].

The physical channel types defined in LTE include the following:

- *Physical downlink shared channel* (PDSCH) – The main physical channel used for unicast transmission and transmission of paging information.

- *Physical broadcast channel* (PBCH) – System information that is required by the terminal in order to access the network is transmitted on this channel.

- *Physical multicast channel* (PMCH) – This channel is used for *multi-media broadcast over a single frequency network* (MBSFN).

- *Physical downlink control channel* (PDCCH) – Used for downlink control information, mainly scheduling decisions that are required for reception of PDSCH and for scheduling grants enabling transmission on the PUSCH.

- *Physical hybrid-ARQ indicator channel* (PHICH) – This channel carries hybrid-ARQ acknowledgement to indicate to the terminal whether a transport block should be retransmitted or not.

- *Physical control format indicator channel* (PCFICH) – This channel provides the terminals with information necessary to decode the set of PDCCHs.

- *Physical uplink shared channel* (PUSCH) – The main physical channel used for uplink transmission, i.e the counterpart to the PDSCH.

- *Physical uplink control channel* (PUCCH) – Used by the terminal to send hybrid-ARQ acknowledgements indicating retransmission of downlink transport block(s) to the eNodeB, to send channel status reports for downlink channel-dependent scheduling and for requesting resources to transmit uplink data upon.

- *Physical random access channel* (PRACH) – Is used for random access.

Note that there is only one PCFICH in each cell and only one PUSCH and PUCCH for each terminal [1].

3 Simulation model

Simulation allows experimentation, although computer simulation mostly requires complex programs and is time consuming. However, computer simulation has several advantages compared to direct experimentation or mathematical models. Some of the most primary advantages of using simulations are that it is possible to experiment with different scenarios, repeating scenarios to find cause-and-effect relationships and the possibility to take risks and explore possibilities without thinking about cost as stated by A. E. Sheikh et al. [10]. Time-flow handling in simulations may be managed using time-slices (move forward in equal time intervals) or event-driven (eliminates unnecessary processing). The behaviour of the system can be *deterministic* or *stochastic*: deterministic systems have a behaviour that is entirely predictable, whereas stochastic systems cannot be predicted, but some statements can be made about how likely certain events are to occur [10].

At Ericsson Research, simulation plays an important role in the research and development of LTE. This chapter describes briefly the simulator environment and model in addition to related platforms and technologies.

3.1 Related simulation platforms and technologies

The simulation of LTE radio networks at Ericsson Research are achieved through a Java simulation platform developed exclusively by Ericsson Research [6]. However, simulation is nothing new within the research and development of telecommunications, since it has been extensively used as an engineering tool for design, implementation and optimization of radio networks for a very long time. Hence, a diverse range of simulation software and frameworks exist, both free and commercial, which are able to model complex wireless network systems.

One of the better renowned network simulators is *OPNET* [11], which is a software suite containing simulation technologies for network and wireless network simulation modeling. OPNET also offers data

visualization, GUI-supported modeling, result prediction, monitoring and application optimization. OPNET arrives with a commercial license and requires some detailed implementations to be implemented in C/C++ programming languages [11].

Another publicly available network simulator is *Ns-2* [12], which is a discrete-event simulator maintained as an open-source project, originating from UC Berkely. Ns-2 provides support for the simulation of TCP, routing and multicast protocols over wired and wireless networks and is primarily targeted for UNIX systems, even though it may be built and run on Microsoft Windows with Cygwin support [12].

WinProp Software Suite [13] is a commercial software suite, containing tools for radio network planning and mobile radio wave propagation simulations, supporting detailed models of indoor and outdoor environments with different infrastructures. It supports several network standards such as 2G, 3G, wireless LANs and WiMAX [13].

WarnSim [14], is a simulator for circuit-switched wide area radio networks such as *Land Mobile Radio System* (LMR), *Personal Communication System* (PCS) and *Public Safety Wireless Network* (PSWN). The simulator is developed in C# .NET and hence only currently runs on Microsoft Windows platforms with the .NET framework installed [14].

The computation and visualization software suite MATLAB [15] is another application extensively used for simulator implementations in radio network simulation. Several publicly available LTE technology related simulators developed for MATLAB also exists, such as the LTE simulator developed at the Vienna University of Technology [16].

3.2 Ericsson Research LTE simulation platform

3G *long-time-evolution* (LTE) networks are simulated at Ericsson Research to evaluate performance in terms of coverage, capacity and quality in a multi-cell system [17]. The LTE simulator is built using an event-driven approach and an object-oriented hierarchical deterministic simulation model. The platform provides implementations of entities and physical models important to a radio network simulator, for example: user generators, radio network, transport network, Internet, deployment and propagation models. Additionally, the platform provides detailed models of the radio network, including multi-cell

interference (slow and fast fading), protocols (MAC, RLC, TCP/IP), physical layer (OFDM) and traffic models (web, VoIP, streaming). The simulation models, modelling both physical objects and logical objects include the following (see Figure 6) [17]:

- *Mobility*, *deployment* and *propagation* models are used to specify the movement for mobile users, specify their distribution and define typical path gains, shadow fading and multi-path fading for different scenarios.
- *Physical layer models* involve receivers, transmitters, decoders, modulation and demodulation as well as physical level communication protocols.
- *Radio protocol models* are used to model protocol stacks and protocol operations involving protocol specific data structures, buffers, transmission and retransmission.
- *Application traffic models* and *Internet protocol models* operate on the highest level and involve user traffic models, including for example, voice-over-IP (VoIP) or web traffic as well as Internet access.
- *Radio resource management (RRM)* models handle link adoption, scheduling, power control, quality measurements, hand-over etc.

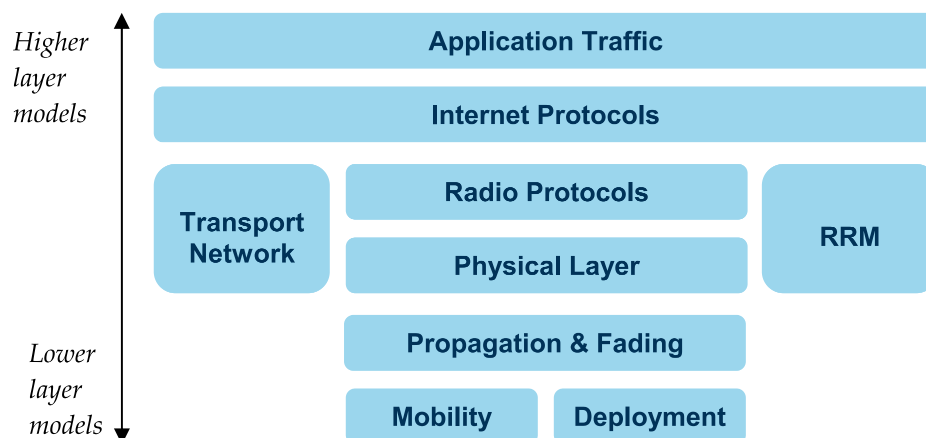


Figure 6: Conceptual model of high and low level models of physical and logical entities in the LTE simulator [17].

It should be noted that there are also models of physical entities, including for example mobile user equipment, *eNodeB* base stations and physical antennas which are not illustrated in Figure 6 [17].

The simulator environment is implemented in Java and runs on Sun's standard *Java Virtual Machine* (JVM) [6] which enables effortless cross-platform interoperability for operating systems (OS) which have a JVM implementation [18].

3.2.1 Events and timers

The LTE simulator platform is event-driven and hence all processing is handled using an event queue containing events that are scheduled to be executed in the future. The main loop of the simulator pops events from the queue, invokes them and updates the simulation logical time to the event time. When the event-queue is empty simulation stops. This is illustrated in Figure 7, where events are scheduled ahead of time and are executed sequentially as the current simulation time is advanced.

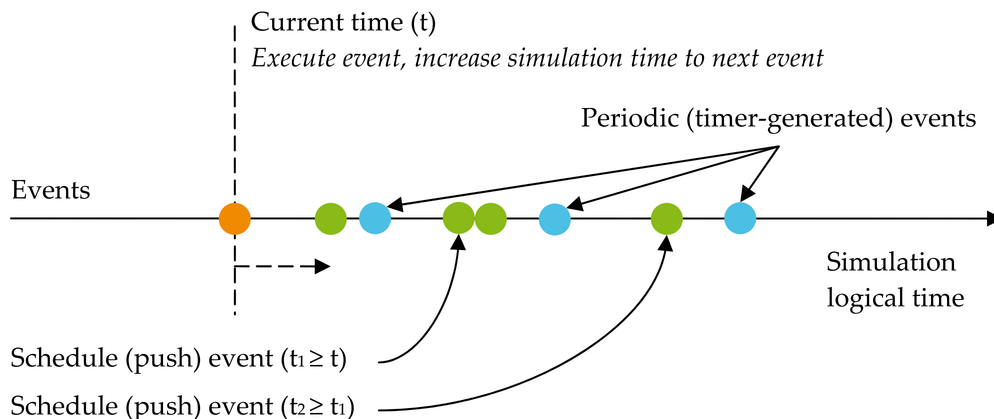


Figure 7: Scheduled events are executed according to logical simulation and time is then advanced to the next event. [6]

Events in the simulation platform are low-level objects that are used to control the timing and order of execution. Event objects are used once and then thrown away. Event objects implement the Java *Runnable* interface and hence may contain arbitrary Java code that may invoke methods on other objects. Another convenient means of controlling execution within the simulator platform involves logical timers that perform a desired operation periodically. This makes timers very suitable to execute operations associated to the periodic behaviour of

LTE sub frames such as the reception of physical channels, scheduling of transmissions and updating of radio propagation models [6].

3.2.2 LTE physical layer models

When user data is available for transmission from the higher layers, the LTE MAC layer typically determines a transport format based on the current channel quality and the amount of data to transmit. Then, the LTE physical layer models set a suitable transmit power, modulation and code rate. This transmit power is used to calculate the received power and the interference by the propagation and interference models. Finally, the physical layer models notify the higher layers using OK flags to indicate successful reception or not. The relations and typical interactions between higher layers, the physical layer as well as propagation and interference models are conceptually illustrated in Figure 8 [18].

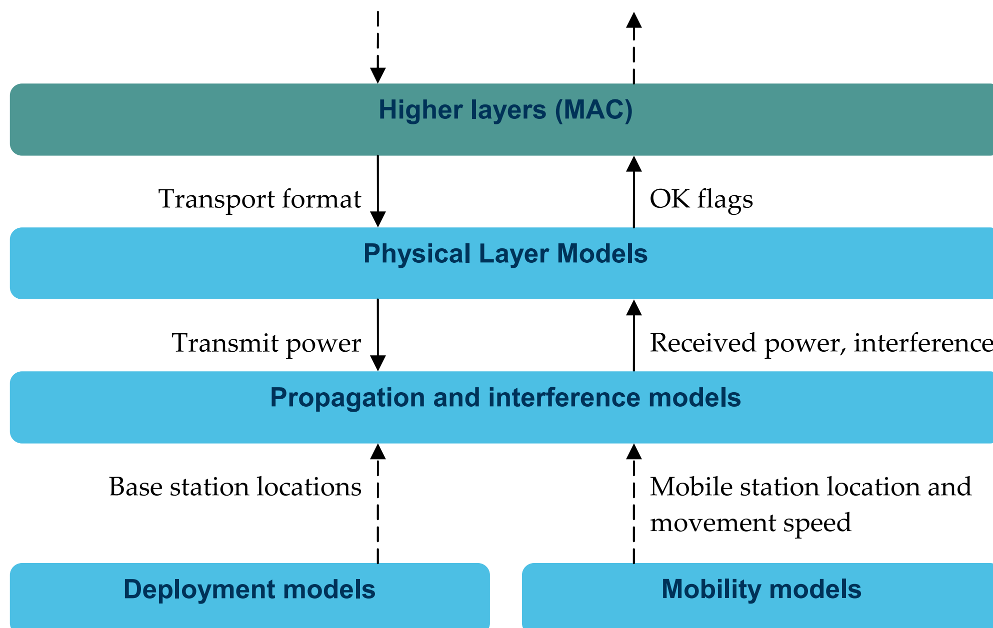


Figure 8: Conceptual overview of interaction and relations between higher layers and physical layers as well as propagation and interference models. [18]

The physical layer model used in LTE is an OFDM channel model used for the downlink and a single-carrier channel model for the uplink. The OFDM channel model contains sub-models that model a physical channel receiver which calculates the *Signal-to-Interference Ratio* (SIR) for each sub-band. A receiver model then combines the SIR values,

modulation format, and number of symbols, and computes the *block error probability* (BLEP) value that is combined with a random number to set the OK-flag sent to the MAC as illustrated in Figure 8 [18].

The uplink single-carrier channel has a corresponding functionality with a consecutive set of sub-bands, a common modulation, and an evenly distributed power per sub-band. Similar to the downlink, a SIR value is first computed and is then mapped to a BLEP value, from which the OK flag is obtained using a random number [18].

3.2.3 Simulation output

The LTE simulator environment uses an event-based logging system for simulation output, supporting new log event handlers (derived from the abstract class *LogHandler*) to be registered dynamically. Output parameters to be logged are defined as data fields (*LogField*) in the different simulation models and are registered in a centralized log manager (*Logger*). The output parameters to be logged in a particular scenario are registered before the simulation starts. During simulation run-time *LogItems* generate events (*LogEvents*) as illustrated in Figure 9 when data field values are updated. Log events are then distributed to all registered event handlers (*LogHandlers*) for the particular *LogItem*. Hence, every log handler may define its own means of interpreting and processing simulation output data through class specialization.

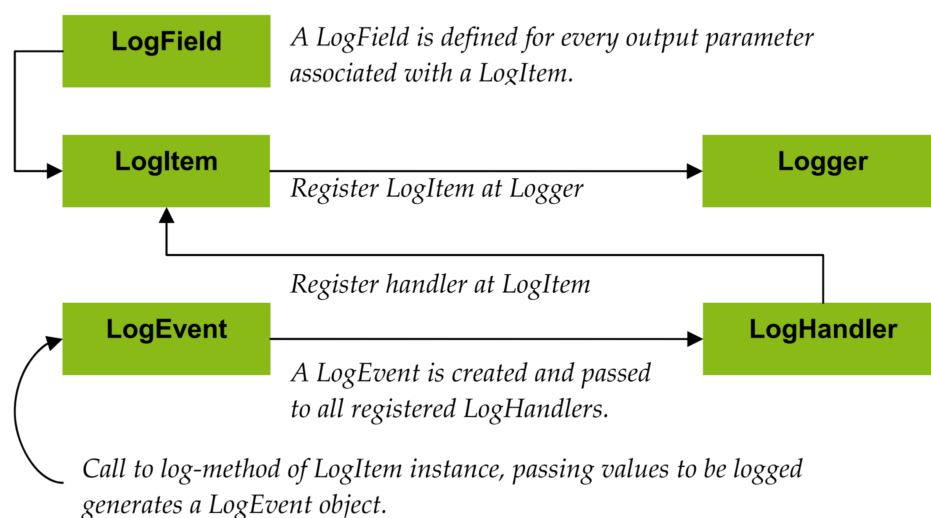


Figure 9: Conceptual overview of the LTE simulator logging system. [18]

3.2.4 Previous profiling results of the simulation environment

Previous profiling and optimization work on the Ericsson Research LTE simulator has been conducted in 2008 [4], improving run-time performance and decreasing memory consumption. The results obtained are useful within this work, but since the simulator application is an ongoing project, the simulator has evolved and several changes have been made. Hence, the results presented by L. Zhang [4] may not be completely credible or relevant for the current implementation.

4 Parallel Computing

One of the mechanisms generally adopted to increase performance, efficiency and smooth running of a system is parallel processing. The concept of parallel processing can simply be described as completing a large task both quicker and more efficiently by dividing it into several smaller subtasks and executing them simultaneously using more resources. However, while applying parallel processing to a process, a number of factors must be considered, for example: whether the task can be performed in parallel, its cost effectiveness, synchronization of the sub-tasks and communication among the resources [19].

4.1 Processor evolution and parallel architectures

Parallel hardware architectures for parallel computing have been the subject of research and development for scientific purposes for several decades as stated by M. O. Tokhi et al. [19], but now shared memory multi-core processor architectures have been developed and manufactured for use in standard desktop computers. This introduces new software requirements in order to utilize multi-processor hardware efficiently in contrast to the traditional processors which enabled higher performance primarily through *instruction level parallelism* (ILP) techniques such as: pipelining, caches, superscalar execution, out-of-order execution etc. [20].

Modern research programs such as the *Tera-scale Computing Research Program* initiated by *Intel Corporation* [21], has increased efforts to advance computing technology by scaling multi-core architectures to 10s or 100s of cores and embracing a shift to parallel programming with improved performance and increased energy-efficiency. This is merely another indication that parallel processing and parallel programming is becoming the future paradigm of computing and that software developers and programmers have to adapt to the associated idioms to create efficient and high performance applications.

4.2 Fundamental components of parallel processing

The concept of parallel processing includes some basic terms and fundamental characteristics. To understand parallel processing at the implementation level it is vital to at least be familiar with these concepts.

The smallest unit of a program that is executed by one processor is usually referred to as a *task* and consists of a sequence of instructions that operate together as a group. Concurrency among processors is exploited only among tasks and in order to be executed a task must be mapped to a *unit of execution* (UoE, usually denoted UE, but UoE is chosen as an acronym within the context of this thesis to avoid confusion with *user equipment* which is denoted UE). The constitution of an UoE may be a *process* or *thread*. Processes hold a collection of resources such as a runtime stack, signal handlers, I/O descriptors etc. that enables the execution of program instructions. A process can be defined as "heavyweight" and has its own address space, while a thread may be considered "light-weight", shares resources and belongs to a process.

A parallel program can generally be said to be composed of a number of UoEs each executing a subset of tasks. UoEs usually require synchronization and communication between them when they are executed on a physical processor, more generically described as a *processing element* (PE). Using the term PE is a convenient way to describe parallel system processing hardware as the environment may involve different types of processing hardware or may consist of a distributed processing network of several machines [5].

If the number of UoEs in a program is more than the number of PEs, more than one UoE is assigned to some or all of the PEs. On the other hand, if the number of UoEs is less than the number of PEs, some PEs are not assigned UoEs and remain idle during the execution of the program or are assigned to a PE when it becomes available depending on the scheduling algorithm [5].

There are several levels of applicable parallelism at various processing levels in a parallel program, mainly on the basis of computational grain size, usually referred to as *granularity*. M. O. Tokhi et al. [19] defines five levels of parallelism: job (program) level, subprogram level, procedure

level, loop level, instruction (expression) level and bit level as illustrated in Figure 10 [19].

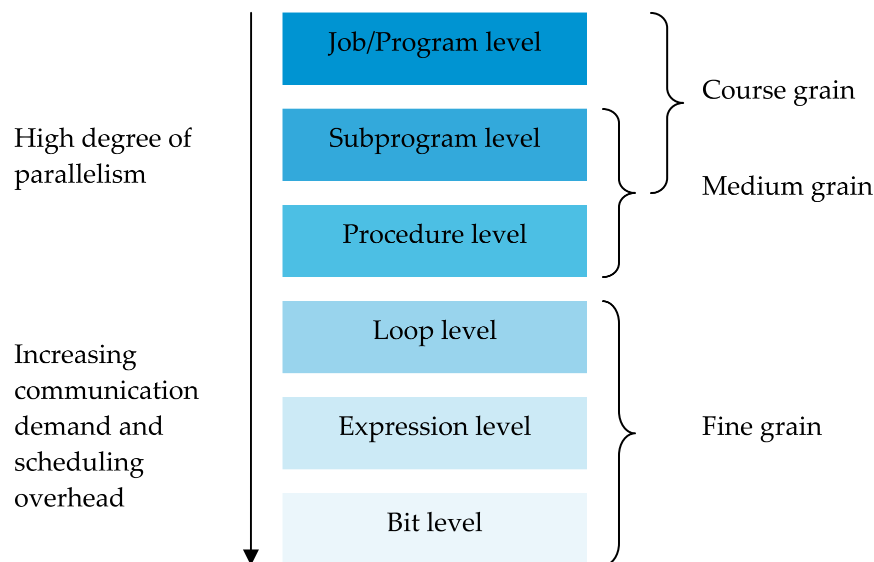


Figure 10: Levels of parallelism according to taxonomy of M. O. Tokhi et al. [19].

The highest level of parallelism occurs if multiple jobs/programs are executed concurrently, whereas the grain size can be as high as tens of thousands of instructions in a single program. However, the lower the granularity, the higher the demand regarding communication and scheduling. This generates considerable overhead for very fine-grained parallelism.

Most algorithms are described as sequential algorithms. Hence these algorithms must be parallelized to be executed on a parallel architecture. The goal of parallelization is to obtain high performance and increased speed over the best sequential program that solves the same problem. This requires efficient load balancing among Pes and a reduction in communication overhead and synchronization. The steps involved in parallelizing a sequential algorithm are as stated by [19]:

- *Decomposition* of the total computation into an appropriate number of tasks. Decomposition has a substantial impact on the performance of a parallel process and to ensure better performance all processes are required to be kept busy as much of time as possible through efficient decomposition.

- *Assigning tasks to UoEs,* in such a way that workload is balanced among UoEs (load balancing) and interprocess communication and runtime overhead are minimized. Load balancing primarily concerns data access, computation and communication while reduced interprocess communication is very important in order to achieve better performance especially when a number of PEs are involved in the parallel execution of an algorithm.

- *Orchestration,* involves synchronization and specification of data exchange among processes through data structure organization and task scheduling. The main objective with this step is to reduce the cost of communication and synchronization, preserving the locality of data reference and reducing the overhead of parallelism management.

- *Mapping UEs to PEs for execution* can be done manually by the programmer or with the help of an appropriate scheduling algorithm and operating system.

4.3 Performance metrics for parallel computing

Parallel processing has several advantages over sequential processing: decreased execution time, better scalability and lower cost-to-performance ratio. Parallel processing is therefore able to handle larger tasks. However, these advantages are not easily attainable. The speedup of a program using multiple processors or multiple processor cores for parallel computing is limited by the execution time necessary for the most time-consuming sequential fraction of the program. Regardless of how many processors are devoted to a parallelized execution of a program, the minimal execution time cannot be less than the sequential execution of the part that cannot be parallelized. Several theories have therefore been proposed to estimate the performance gains of parallelizing parts of a program [19].

4.3.1 Speedup

The theoretical maximum parallel speedup factor (S_N) for a homogenous architecture is defined as the ratio of the execution time (T_1) on a single processor, to the execution time (T_N) on N processors as given by:

$$S_N = \frac{T_1}{T_N} \quad (4.1)$$

The maximum ideal speedup for a processor architecture with N identical processors is N. In practice the speedup is much less, since conflicts over memory access, communication delays, inefficiency in algorithms etc. may cause the processor to not perform in an ideal manner. However it is possible to achieve speedup above the ideal speedup, known as *super linear speedup*, due to anomalies in programming, compilation, architecture, cache usage etc. The possibility of achieving super linear speedup is highest on multiprocessor systems with sophisticated cache and registry management as highlighted by M. O. Tokhi et al. in [19].

4.3.2 Efficiency

The efficiency (E_N) of a homogeneous parallel system is defined as:

$$E_N = \frac{S_N}{N} \cdot 100\% = \frac{T_1}{N \cdot T_N} \cdot 100\% \quad (4.2)$$

where N is the number of processors, S_N is the speedup factor using N processors, T_1 is the execution time on a single processor and T_N is the execution time on N processors. Efficiency can be interpreted as a measurement of the average utilization of the N processors, expressed as a percentage. This measure also allows for a uniform comparison of the various speedups obtained from systems equipped with different numbers of processors. It has been illustrated that efficiency is related to the granularity of the system [19].

4.3.3 Amdahl's law

To determine the maximum expected improvement to an overall system when only part of the system is improved, *Amdahl's law* as stated by Gene Amdahl [22] defines a fundamental rule derived from equation (4.1). It is a model for the relationship between the expected speedup of a parallelized implementation of an algorithm relative to the corresponding sequential algorithm under the assumption that the problem size remains the same when parallelized.

Amdahl's law states that the maximum theoretical speedup factor S_{MAX} for an algorithm or program executed on a parallel architecture is given by:

$$S_{MAX} \leq \frac{T_1}{T_N} = \frac{T_1}{\eta T_1 + \frac{(1-\eta)T_1}{N}} = \frac{1}{\eta + \frac{(1-\eta)}{N}} = \frac{N}{\eta(N-1)+1} \qquad (4.3)$$

where $\eta \in [0,1]$ is the proportion of the program that is executed sequentially, $(1-\eta)$ is the proportion that can be executed in parallel and N ($N \geq 1$) is the number of processors that executes the parallel part. Note that for the limit when $N \to \infty$, it is evident that $S_{MAX} \to 1/\eta$. This indicates that the cost-to-performance ratio rapidly falls as N is increased, even for a small component of η. For this reason, parallel computing is only profitable for either small numbers of processors or problems with very high values of $(1-\eta)$, so-called *embarrassingly parallel problems* [19].

It is also noticeable that his analysis is rather primitive and neglects other crucial performance factors such as memory bandwidth, I/O bandwidth and only holds true for homogenous architectures with a fixed load. In practice, creating additional parallel tasks may increase the overhead and the chances for there to be contention regarding shared resources. On the other hand, if the original serial computation is limited by resources other than the availability of CPU cycles, the actual performance could be much better than Amdahl's law would predict [5].

4.4 Parallel programming

The method of *multitasking* has long been part of all modern operating systems in which multiple processes share common processing resources such as the CPU. In the case of a uniprocessor system this means that the processing unit has control transferred from one process to another causing their instructions to be interleaved at stages of their execution, also known as *context-switching* or *time-sharing*. In the case of a multiprocessor environment this means that more than one process can proceed independently with its execution process. On the other hand, within a multiprocessing environment, processes of a single program can be executed concurrently in a number of processors.

However, processes may interact and affect each other, based on the data dependencies and control dependencies of the process. This addresses the need to provide additional effort in relation to process mapping, scheduling and interprocess synchronization in order to execute a parallel program in a parallel architecture compared to executing a sequential program in a uniprocessor environment [19].

4.4.1 Multithreading

Multithreading for both uniprocessor and multiprocessor computing is a growing technology for generic parallel and distributed computing and is currently a subject of widespread interest among scientists and professional software developers. Threads reduce overheads by sharing fundamental parts, for example data, memory stack and file I/O. Context-switching between threads is also typically faster than context-switching between processes since memory access is reduced [23].

Dependency is one of the key issues in multithreading for high-performance computing. Data dependency between two blocks or statements requires memory access time and in practice increasing dependencies implies increasing access time which degrades performance. Thus it is essential to study data dependencies in an algorithm intended for a concurrent thread implementation. Detection of multithreading potentials within an application involves discovering sets of computations that can be performed simultaneously. The approach to parallel multithreading is thus based on the study of data dependencies [19].

4.4.2 Synchronization

Synchronization is another key issue of concurrent multithreading. The performance of the synchronization mechanism of a multiprocessor determines the granularity of parallelism that can be exploited on that machine. Communication between threads typically involves reading and writing to a shared resource and synchronization is required to guarantee that two processes or threads do not attempt to access the same resource simultaneously causing a system failure or producing incorrect results. It is also vital to design algorithms that minimize the requirement for synchronization so the threads can solve the actual problem instead of spending their time executing synchronization code, causing synchronization overhead.

In general there are two types of synchronization: synchronization for precedence and synchronization for mutual exclusion. The former method guarantees that one event does not begin until another event has finished, while the latter guarantees that only one process can access the critical section where the data are shared and must be manipulated. This is achieved using mechanism such as *semaphores*, which can be described as a locking mechanism to lock critical resources, *mutexes*/condition variables, event flags and message queues. For time critical systems it may also be required to perform thread scheduling to avoid situations including for example a high-priority thread waiting for a low-priority thread to release a lock on some critical resource [19].

4.5 Algorithm analysis and design

In practice more than one algorithm exists for solving a specific problem. The choice of the most efficient algorithm for a given problem and for a specific computer is a difficult task and depends on many factors, including data and control dependencies, granularity and regularity of the algorithm [19].

4.5.1 Data and control dependency

Data dependency is a key issue in algorithm analysis for real-time and time-critical high performance computing. Analyzing data dependency involves studying how to reduce block or statement dependencies, how to reduce memory access time and what the impact of data dependencies is on interprocess communication. The main classes of dependencies are data dependence and control dependence. Dependence indicates the order in which results must be calculated. The dependence also sets an upper bound on how much parallelism can possibly be exploited for a specific scenario.

Detection of parallelism in an application involves determining sets of computations that can be performed simultaneously. The approach to parallelism is based on the study of data dependencies. The presence of dependence between two computations implies that they cannot be performed in parallel. In general it can be stated that the fewer the dependencies, the greater the parallelism. Many algorithms have regular data dependencies that repeat throughout the set of computations in the algorithm. For such algorithms, dependencies can be concisely described mathematically and be easily manipulated. However, there

are algorithms for which dependencies vary from one computation to another and these algorithms are more difficult to analyse. When two or more algorithms have similar dependencies, it means that they exhibit similar parallel properties.

A control dependency, on the other hand, determines the ordering of an instruction with respect to a branch instruction so that the instruction is executed in the correct program order and only at the appropriate time. Control dependency is preserved by two properties in a simple sequential computing:

- Instructions executed in program order, ensuring that an instruction that occurs before branching is executed before the branch control point.

- The detection of control ensures that an instruction that is control dependent on a branch is not executed until the branch direction is known.

The presence of dependencies indicates complexity in an algorithm and in turn communication overhead in a parallel processing context [19].

4.5.2 Granularity and regularity

Granularity and *regularity* are two important issues of algorithm analysis and design for high-performance sequential and parallel computing. In particular the study of parallelism includes interprocess communication, issues of granularity of the algorithm and of the hardware in addition to the regularity of the algorithm. *Hardware granularity* is defined as the ratio of computational performance over the communication performance of each processor within the architecture, according to:

$$Hardware\ granularity = \frac{Runtime\ length\ of\ task}{Communication\ overhead} = \frac{R}{C} \qquad (4.4)$$

where R is the actual computation time of a task and C is the amount of time due to communication overhead during the execution of the corresponding task. When the hardware granularity is very small, it is unprofitable to use parallelism. On the other hand, when the hardware granularity is very large, parallelism is potentially profitable. A

characteristic of fine-grained processors is that they have fast inter-processor communication and can therefore tolerate small task sizes and still maintain a satisfactorily high hardware granularity or R/C ratio. However, medium-grain or course-grain processors with slower inter-processor communication will produce correspondingly smaller hardware granularity (R/C ratio) if their task sizes are also small.

Task granularity can be defined as the ratio of computational demand, i.e. the time required to execute the task without any communication overhead over the communication demand (actual communication time) during execution of the corresponding task. Typically a high compute/communication ratio is desirable. The concept of task granularity can also be viewed in terms of compute time per task. When this is large it is a course-grain task implementation. When it is small it is a fine-grain task implementation. Although course-granularity may ignore potential parallelism, partitioning a problem into the finest possible granularity does not necessarily lead to the fastest solution, as maximum parallelism also has a maximum overhead, particularly due to increased data dependencies. Therefore it is essential to choose an algorithm granularity that balances useful parallel computation against communication and other overheads considering the system at hand.

Generally algorithms can be classified on the basis of their characteristics as regular, irregular and mixed. *Regularity* is a term used to describe the degree of uniformity in the execution thread of the computation. In addition to their own characteristics, the regularity or the irregularity of algorithms also depends on how the code is developed for implementation. Thus, it is essential to explore the characteristics of an algorithm, hardware and coding style for computation [19].

4.6 Design patterns for parallel computing

In the book *Patterns for parallel programming* [5], T. G. Mattson et al. propose a design pattern for algorithm analysis and dealing with parallelization of problems by introducing a pattern language. Design patterns have long been used in system development and programming as a means of tackling problems in a unified and standardized way, hence increasing both the readability and writeability of application source code. Using design patterns also decreases the chance of

performing common mistakes and solves problems using proven methods.

The patterns introduced by Timothy G. Mattson et al. [5] involve finding concurrency, algorithm structure, supporting structures and implementation mechanisms. Explaining useful generic patterns falls out the scope of this thesis, but design patterns have proved to be a convenient way of introducing parallel programming and concurrent algorithm design to traditional programmers in order to reduce the chance of errors and to producing consistent code [5].

4.7 Java technologies and frameworks for parallel computing

Multithreaded applications, parallel computing and distributed computing becomes a possibility in Java both through the native Java concurrency API [24] as well as third-party frameworks and libraries, both free and commercial. There are also attempts to introduce parallelism through specialized compilers and *Java Virtual Machine* (JVM) implementations.

4.7.1 Java concurrency API

The Java 2 platform includes a package of *concurrency utilities* (CU) [24] since the release of the Java version 1.5. This package contains classes which are designed to be used as building blocks when building multithreaded applications. For example, the Java CU includes several thread-pool implementations, a framework for asynchronous execution of tasks, a host of collection classes optimized for concurrent access and synchronization utilities such as counting semaphores, atomic variables, locks and condition variables. The *Task Scheduling Framework* of Java CU also contains the *Executor* framework providing a generalized interface for invocation, scheduling, execution and control of asynchronous tasks. The Java CU API also contains a wide range of thread-safe data structures to ease synchronization issues such as atomic variables and blocking queues.

The current Java CU API can be seen as supporting course-grain parallelism within Java applications. However, B Goetz [25] states that when JDK 7 is released it will contain a *fork-join* framework with support for fine-grained parallelism through optimized worker-thread implementations [25].

4.7.2 Parallel Java (PJ)

Parallel Java (PJ) [26] is a free API and middleware for parallel programming Java on *shared memory multiprocessor* (SMP) parallel computers, cluster parallel computers and hybrid SMP cluster parallel computers. PJ is licensed under the terms of the *GNU General Public License* (GPL) and was primarily developed by Prof. Alan Kaminsky, Rochester Institute of Technology for educational purposes. The API includes class support for common patterns in scientific parallel computing including parallel for-loops, parallel matrix operations, concurrent task/job execution and scheduling. The framework also contains support for interprocess-communication vital for parallel computer cluster processing [26].

4.7.3 Java Parallel Processing Framework (JPPF)

Java Parallel Processing Framework (JPPF) [27] is an open source grid computing platform written in Java that aims at simplifying parallel execution of applications. The framework is task-oriented and supports transparent, abstracted execution of tasks/jobs over a computer cluster. Framework configuration files are used to specify host and client configurations. Remote execution is achieved through specific drivers and node packages running on an Apache Ant application server which must be installed on all systems of the processing cluster [27].

4.7.4 JOMP

JOMP [28] is a research project whose goal is to define and implement a set of directives and library routines for shared memory parallel programming in Java as close as possible to the C/C++ *OpenMP* syntax. JOMP is a prototype reference implementation that consists of a compiler and a runtime library. The compiler translates Java source code with directive calls to the JOMP runtime library, which in turn uses Java threads to implement parallelism. JOMP is pure Java and hence can be run on any JVM [28].

4.7.5 MANTA compiler

Manta is a native Java compiler that compiles Java source codes to x86 machine-code executables. The aim of the compiler is to beat the performance of all current Java implementations and support high performance parallel distributed computing. Currently it contains a

Remote Method Invocation (RMI) implementation that is currently about 30 times faster than standard implementations. Manta supports the complete Java 1.1 language (not 1.2, 1.3 etc.) including exceptions, garbage collection and dynamic class loading. Manta also supports some Java extensions, such as *JavaParty* programming model, replicated objects and efficient divide and conquer parallelism. A *distributed shared memory* (DSM) model is built upon Manta, called *Jackal*. However there is currently no *JAR*-package support and no *Java Swing* support [29].

4.7.6 Javab compiler

In *Javab Manual* [30] A. J.C. Bik and D. B. Gannon present a byte-code parallelization approach for automatic loop parallelization in Java. Implicit parallelism is made explicit by standard JVM multithreading mechanisms. A prototype implementation of a compiler written in C, based on this technique is available online [31]. However, the compiler has only been tested and verified for simple loop algorithms with clear data dependencies. For the particular array-based algorithm examples [30], speed-up gains in range 2 to 3 have been obtained for high numbers of iteration [30].

4.8 Alternative technologies for parallel and concurrent computations

Apart from multithreading, a number of traditional and emerging technologies exist for parallel computing that require to be mentioned. *Symmetric multiprocessing* (SMP) is a computer system with multiple identical processors that share memory and are connected via a bus. Bus contention prevents bus architectures from scaling. As a result SMPs generally do not comprise more than 32 processors, but are very cost effective [20].

Distributed computers or distributed memory multiprocessor systems are systems in which the processing elements are connected by a network. Distributed computers are highly scalable. Cluster computing is a form of distributed computer where loosely coupled computers work together as standalone machines interconnected by a network. However, load balancing is a critical problem in cluster computing [32].

A *massively parallel processor* (MPP) is a single computer with many networked processors. MPPs have many of the same characteristics as

clusters, but MPPs have specialized interconnected networks. MPPs also tend to be larger than clusters, typically having more than 100 processors. In MPP each CPU contains its own memory and copy of the operating system and application [20].

Grid computing is the most distributed form of parallel computing and makes use of different computers communicating over the Internet to work on a given problem. Because of the low bandwidth and extremely high latency, grid computing typically deals only with so called embarrassingly parallel problems. Most grid computing applications use middleware software to manage network resources [33].

Specialized *digital signal processors* (DPSs) are specialized microprocessors with optimized architecture for fast operational needs of digital signal processing. DSPs are typically used for a specific application or may be integrated in general purpose computers to run in parallel with the CPU in order to unload the CPU for certain types of computations [34].

The processing unit of video cards, usually referred to as *graphics processing unit* (GPU) are usually specialized SIMD (according to Flynn's taxonomy) architecture processors where each processing unit executes the same instructions at any given clock cycle. However each processing unit can operate on different data elements. This is particularly useful when processing graphics and images as each pixel can be processed in the same manner and computations usually involve matrix operations. [32] As processing power of parallel GPUs has increased so called *general purpose computing* on the GPU (GPGPU) has started to compete with CPU processing. NVIDIA has introduced the API extension *Compute Unified Device Architecture* (CUDA) [35] framework to allow specialized C functions to run on the GPUs stream processors and *Open Computing Languge* (OpenCL) [36] is an open source initiative by the *Kronos Group* (also OpenGL, OpenML) aimed at abstracting CPU and GPU computations.

CUDA currently is targeted for C/C++ languages but Java bindings are available in jCUDA [37] API which uses JBI to communicate with a C library.

5 Research in parallel simulations

Parallel discrete-event simulation (PDES) has gained much attention during the last two decades and a reflection of this is the number of researchers who have used either conservative or optimistic synchronization to successfully simulate their applications. With the extensive refinement in these methods, synchronization has emerged to be the single largest overhead in distributed simulation [38].

5.1 Parallel simulation approaches

A. Hind [39] addresses the problems of parallel multiprocessor simulation in contrast to distributed parallel simulation. He states that excessive processing and huge memory requirements forces simulation to be executed in distributed memory computing clusters, but points out that multiprocessor systems may be very well suited for parallel simulation. Instead of message passing algorithms and distributed memory, shared memory structures address the data access synchronization and cache coherency problems. Furthermore, A. Hind [39] categorizes simulation model decomposition to include five different approaches: parallelizing compilers, distributed experiments, distributed functions, distributed events and distributed model components, but also points out the problems associated with each of these approaches.

Automated parallel compilers seldom offer much gain from parallelization as the problem is modelled according to a sequential model in a sequential language and compiled to run on multiprocessor hardware. The biggest advantage of this approach is that it is transparent to the user.

Distributed experiments may be conducted by running several separate simulations on separate processors in parallel. This is particularly efficient for stochastic simulations, as results can be averaged at the end of the run. This approach is extremely efficient as no co-ordination is required between processors, with the exception of results averaging and presentation. Hence, for N processors this may give close to ideal speed-up of N. The only overhead is loading the model into each

processor. However in terms of hardware, distributed experiments may not be possible due to the high memory requirements.

Distributed functions involve different tasks of a simulation being placed on separate processors. For instance, processors may be dedicated to random number generation, event list processing, etc. The processor may be identical or may be tailored to each individual function. The advantages associated with this model are its freedom from the possibility of deadlock and its potential scalability. The architecture may also be made transparent to the user as each function code can be divided and placed within each processor fairly easy. The disadvantage is the communication overhead between functional processors, which may become the limiting factor in relation to performance. This approach also fails to exploit any parallelism in the actual system or physical entities being modelled. The law of diminishing returns sets in at an early stage using this approach.

The *distributed events* approach uses a global event list, as in sequential simulation, to schedule available processors to process the next event on the list. The difficulty is maintaining consistency in the simulation as the next event available on the list may be pre-empted by other events currently being processed by other processors. The need for global simulation control points significantly towards the use of shared memory multiprocessor architectures so that all processors can have access to the global event list. The results for this approach appear to indicate that it is reasonable if there are only a small number of processes required and a large amount of global information used by the components of the system.

The most popular method for parallel simulation is *model decomposition*. Decomposing a simulator model means that the simulation model is divided into a number of sub-components that are then assigned to processes. One or many processes can then be assigned to execute on each processor. Model decomposition usually follows the logical structure of the real system being simulated. Therefore this approach can take advantage of any parallelism inherent in the system to be modelled, so it appears to promise significant speed-up on a multiprocessor system. However, this only holds true if the simulation does not require a significant amount of global information and control. The major overhead is communication between processes executing on

different processors. In a shared memory environment this can be handled by global shared variables or message passing. The other major problem associated with this method is the synchronization of events during simulation.

Synchronization schemes in discrete-event parallel simulation can be divided to two groups: conservative approaches and optimistic approaches. In the conservative approaches causality problems are avoided by relying on some strategy of determining which events are safe to process. Generally conservative approaches can provide good performance with sparsely connected systems which have less opportunity for deadlocks and/or an application which contains good look-ahead properties. The worst case for a conservative synchronization approach is to be forced into almost sequential operation coupled with the synchronization mechanism overheads.

Optimistic approaches allow causality errors rather than avoiding them, but when they are detected, a roll-back mechanism is employed in order to recover which is achieved by re-simulating from the time of the error. Therefore optimistic approaches do not require to determine whether or not it is safe to proceed, they only need to detect the error and recover. The advantage of this is that the simulator can exploit the parallelism fully in applications which may produce causality errors, but in reality rarely do. Obviously, the greater the amount of causality errors that a simulation produces, the greater the synchronization overhead.

Roll-back is accomplished by undoing all the effects of all events that have been processed prematurely and is accomplished by returning to the old correct state which is taken from a store of previous states. In addition, previously sent messages must be unsent, typically by sending anti-messages that cancel the effect of the original. Time warp approaches such as the one described in this case do not lend themselves to fine-grained parallelism due to the memory overheads required. Each process requires substantial memory capacity as well as the mechanism for restoring to the simulation state. Also, it is unproven that a continuous cycle of roll-backs may be possible for a particular simulation [39].

5.2 Evolution of parallel discrete-event simulation

Different kinds of relaxation to causality (or synchronization) have been proposed such as the *NoTime*-simulator [38] as an approach to make optimistic, conservative simulations more efficient by reducing synchronization. Most research focuses on distributed grid or cluster computing with high communication costs instead of considering simulation on a multi- or many-core processor. However, many problems are common as the problem solving algorithms are distributed among threads in a multiprocessor environment and hence require synchronization.

Object-oriented approaches for distributed parallel simulations have been proposed, introducing concepts such as *Abstract Parallel Machines* [40] to promote key software engineering qualities including reusability and extendibility while still fulfilling parallel processing requirements such as scalability, portability and performance [40].

A conservative implementation of a very basic model for high performance parallel simulation of telecommunication networks is presented in [41]. The performance of a distributed simulation program on a multiprocessor machine simulating a packet-oriented *CCIT Signalling System No. 7* (SS7) system was examined and implemented through a queuing network. The results indicated that the benefit of parallelism increased as the simulation time was increased and converged to the number of processors within the system [41].

Efficiency of multiprocessor systems is directly dependent on the load balance, as addressed in 1997 by Azzedine Boukerche and Sajal K. Das in *A Dynamic Load Balancing Algorithm for Conservative Parallel Simulations* [42]. They proposed an algorithm for dynamic load balancing for conservative parallel simulations, based on null-messages to avoid deadlocks and to increase parallelism of the simulation. The primary goal of this work was to minimize synchronization overhead, which resembles the work carried out in the *NoTime*-simulator [38], but achieving this through efficient utilization of hardware. They state in their work that their dynamic load balancing algorithm considerably reduces synchronization overhead.

P. Heidelberger and D. Nicol address in *Building Parallel Simulations From Serial Simulators* [43] that for PDES to have a practical impact,

PDES tools must hide the complexities of time synchronization from users and provide capabilities approaching those of industrial quality serial simulators. They successfully implemented a message-based distributed MPI-based parallel simulation of a mobile cellular network model, with a speedup linear to the number of processors using a high degree of look-ahead [43].

A framework called SWiMNet is presented in *Partitioning Parallel Simulation of Wireless Networks* [44], which is a simulation framework for wireless and mobile telecommunication systems, that uses a two stage parallel simulation: conservative scheme at stage 1 and time warp (optimistic approach) at stage 2. In the first stage, the simulation model uses a mobile host partition and in the second stage a cell-based partitioning. Furthermore the simulation uses optimistic approaches by pre-calculating all possible events and using roll-backs. Their work indicates that careful partitioning can have a significant impact on performance, but addresses the critical importance of dynamic load balancing using a round-robin algorithm [44].

In *Exploring the Effects of Hyper-Threading on Parallel Simulation* [45] L. Bononi et al. explore the effects of hyper-threading on parallel simulation middleware. The conclusion in this work states that Intel Hyper-Threading techniques successfully reduced execution-time, but also increased system efficiency and scalability [45].

To conclude this chapter, it should be stated that the most successful examples of parallel simulation models were developed for parallel execution from the beginning, but in *Case Study: Parallelizing a Sequential Simulation Model* [46] a case study of parallelizing a sequential simulation model has been presented. The approach in this work was communication topology simplification, look-ahead specification and modelling changes to eliminate performance bottlenecks which turned out to be a successful solution for the particular problem [46].

6 Methodology

To evaluate the gains of multithreaded LTE simulations executing on multi-processor hardware, a multithreaded prototype must be designed and implemented. It is vital to pay great attention to data and control dependencies when re-designing the current sequential simulator in order to preserve deterministic behavior. Extensive experiments are then required to evaluate the performance of the prototype implementation. The methodology used to consider the evaluation of the parallel LTE simulator prototype is described in this chapter.

6.1 Experimental methodology

In order to determine task granularity in terms of execution time and to find CPU bottlenecks in the simulator application, a profiling tool should be used to collect profiling data during the design phase of a multithreaded simulator. Application profiling is carried out with *JProfiler* from *ej-technologies* [47], which is a convenient tool for identifying CPU "hot spots", call trees and memory usage, but may produce incorrect results if used to measure actual execution time in experimental evaluation. The reason for this is that for sequential simulation one processor may be dedicated to the simulator and one to the profiler application. However, for parallel simulation all processors may be used for executing simulations and hence the profiler and simulator will compete for CPU resources. A better approach is to use high performance timers to perform execution time measurements, which minimizes profiling overhead. Actual time measurements should be normalized to reduce system hardware dependency from the achieved results.

It is vital for the credibility of the profiling results that several different experiments with different simulation scenarios are executed and that experiments are repeated to indicate variations in execution time. The motivation to use different scenarios is to trigger different computational models in the simulator or in terms of parallel processing result in different task granularities.

Experiments should also be performed on different systems with different hardware configurations so as to indicate anomalies in performance that are related to different processors. Differences in architecture, clock frequency and cache size may result in different *hardware granularity* as defined by equation (4.4). Three different systems will be used for the experiments (see Appendix A for details):

- System A – Equipped with an Intel® Core™2 6600 @ 2.40 GHz processor, this gives a total of 2 processor cores. Linux 64-bit OS.
- System B – Equipped with an Intel® Core™2 Extreme CPU X9650 @ 3.00 GHz, this gives a total of 4 processor cores. Windows Vista 32-bit OS.
- System C – Equipped with two Intel® Xeon™ CPU E5440 @ 3.00 GHz, this gives a total of 8 processor cores. Linux 64-bit OS.

6.2 Performance experiments and evaluation criteria

To evaluate the performance of multithreaded LTE simulation as compared to sequential simulation and parallel jobs (distributed experiments), a number of evaluation metrics according to the parallel processing metrics described by M. O. Tokhi [19] are used.

To measure and evaluate the speedup of a parallel implementation versus sequential implementation, execution time as a function of the number of threads (or utilized processors) should be measured. The number of threads should not exceed the number of available processors N of the system since this will only generate additional overhead. Only a single simulation process should be run in each case. This will provide an indication of how the parallel simulator prototype performance is related to the number of available processors. The parallel speedup S_N as defined by equation 4.1 is useful to clearly illustrate the relation between the parallel and sequential versions as more processors (threads) are added. Hence, this experiment also requires that the execution time for the sequential simulator is also measured for the same scenario.

To minimize run-time length of current LTE simulations at Ericsson Research, N independent sequential simulation processes are executed concurrently on a machine with N processors. However, this approach may lead to system bandwidth problems, considering system bus

bandwidth and system memory bandwidth. To evaluate run-time length efficiency, *N* concurrent processes should be started, executing *N* independent sequential simulators. The sum of the execution times of the *N* processes should be measured and compared to the execution time involved in running *N* multithreaded simulations in sequence. The result from such an experiment will indicate whether the parallel prototype executes faster than the sequential version. This experiment is important since it will indicate whether the multithreaded prototype is useful for use in LTE research simulation work or whether it requires further development.

In order to analyze the overhead of multithreaded simulation as well as parallel jobs, the actual measured execution times should be compared to an ideal estimate derived from Amdahl's law. This will make it possible to determine the ratio between the CPU-time used for the actual computations and the CPU-time caused by the overhead: memory access, synchronization, race conditions etc.

To illustrate a summary of the experiments to be performed, the experiments presented in Table 1 should be executed on systems A, B, C (see Appendix A) respectively.

Table 1: The set of experiments to perform on systems A, B and C respectively. The asterisk (*) means that the experiment should be performed for both parallel and sequential implementations.

Number of concurrently executing threads per process. \ Number of concurrent processes	1	2	3	4	5	6	7	8
1	ABC*	ABC		BC		C		C
2	ABC	B		C				
3	B							
4	BC	C						
5								
6	C							
7								
8	C							

In Table 1 only the row with a single thread implies that a sequential simulator should be used. For all other experiments the parallel prototype simulator should be used. A special case is ABC* which indicates that the experiment should be performed for all systems and both parallel and sequential implementations. The intention with

regards to this is to be able to measure the overhead of the parallel implementation executing as a sequential simulator. All simulation experiments have a simulation time (logical time) of 10.0 seconds. The reason for this is to maintain the execution time at a feasible level, while still minimizing the impact of start-up and memory allocation routines.

6.3 Simulation scenarios

Profiling results and analysis as well as experimental results will be based on four different simulation scenarios, further referred to as *scenario I – detailed radio model (DRM)*, *scenario II - file upload*, *scenario III - file download* and *scenario IV - voice-over-IP (VoIP)*. The scenarios are designed to use a great deal of resources and trigger computation intensive models and functions. The reason for this is that the possible gain of parallelization is particularly interesting for very computation intensive scenarios as these require the longest execution times due to a large task granularity. All scenarios involve seven base stations (*eNodeB*) and have three cells per base station, which gives a total number of 21 cells. Each cell has a radius of 166.66 meters. A regular deployment propagation model is used.

The simulation scenarios have an initial number of users generated when simulation starts and that number is then fixed. Users are configured to have two antennas. Downlink and uplink frequency bands are set to 9 MHz if not stated otherwise with 50 sub-bands using a carrier frequency of 2000 MHz.

The differences between the scenarios are stated below:

- *Scenario I – Detailed radio model (DRM)* – This scenario has initially 630 users which will generate fixed size FTP downlink traffic. This scenario is modeled using a very detailed radio model and hence is very computationally expensive.

- *Scenario II - File upload* – This scenario has initially 630 users which will generate fixed size FTP uplink traffic and is based on a simpler radio model than the DRM scenario.

- *Scenario III - File download* – This scenario has initially 630 users which will generate fixed size FTP downlink traffic and is based on a simpler radio model than the DRM scenario.

- *Scenario IV - VoIP* – This scenario has initially 3150 users which will generate voice-over-IP (VoIP) data traffic, which typically consists of small, but many data packets and traffic bursts. The radio frequency bandwidth is set to 4.5 MHz for both the downlink and uplink bands which use 25 sub-bands each. This scenario is based on a simpler radio model than the DRM scenario.

6.4 Environment and physical resources

Java and the *Java Development Kit* (JDK) 1.6 will be used as the language of implementation (including JVM 1.6 for executing experiments) and *Eclipse 3.5.0* will be used as the primary *integrated development environment* (IDE) for development. *Subversion* 1.6 for *Eclipse* will be used for version control. System A used for experimental evaluation is also used as the system for development while systems B and C will be used remotely.

The thesis and associated work will be carried-out at Ericsson Research, Linköping, Sweden, which pursues research within the area of wireless access networks and radio communications. Hence expertise within the area of radio networks, radio network modeling and telecommunications will be available for consultancy throughout the project. Niclas Wiberg, Ph. D. in information theory and *Ericsson expert* in radio network and simulation, Kristina Jersenius *Ericsson research engineer* in radio networks and also Rahim Rahmani, Mid Sweden University will supervise this work. The project involves the handling of confidential information and resources vital to Ericsson, which makes it impossible to cover all details of the simulation platform in a public report. However the information published in this report should provide sufficient illustration in order to describe the simulator aspects that affect the parallelization task and provide the reader with sufficient information to understand this work.

6.5 Verification of program correctness

The LTE simulator currently used at Ericsson Research is deterministic [18]. This means that for a given input seed *X*, the same output value Y is always produced. This holds true if the simulator is executed repeatedly with the same input seed *X* on the same system, but

anomalies may be present between different systems. The variation between systems is platform and hardware dependent and is most likely to occur due to floating-point round-off anomalies between processors [48]. However, in this thesis deterministic simulation is defined as a reproducible result Y for a given input X on a specific system.

Deterministic simulation involves several problems when executed in a non-deterministic manner. For this reason, it is critical to determine a convenient method which guarantees an equal output from both parallel and sequential simulation, as well as verifying the deterministic behavior of the parallel prototype as stated in problem definition of this thesis. Since simulation output by definition is deterministic, sequential and parallel simulation outputs can be compared in order to verify program correctness. Hence, a mechanism or tool capable of detecting irregularities in the simulation output must be created, which is capable of detecting differences in both the output order and the actual values in order to verify program correctness.

6.6 Evaluation of software design transparency and usability

As stated in the section for the initial requirements of this thesis (see section 1.4) the multithreaded prototype should be implemented as transparently as possible to the developer (user). This involves both changes to the current simulator design as well as changes to the ways of thinking when implementing simulation models that are targets for asynchronous execution. In order to evaluate whether the introduction of multithreading constructs into the current design is transparent, the proposed software constructs should be reviewed by a small group of developers who work with the simulator on a daily basis as well as a small group of non-developers who work with software development outside of Ericsson. These groups of people will be consulted in order to answer a questionnaire covering their previous experience of parallel programming, questions considering their thoughts about the design as well as some complementary tests. The results obtained of such a questionnaire might provide a hint of the transparency and usability of the proposed solution. The questionnaire used for evaluation is presented in Appendix B.

7 Design

To investigate and evaluate the performance gains of multithreaded LTE simulation in contrast to sequential simulation or running concurrent independent parallel processes, a parallel LTE simulator prototype must be designed and implemented. This chapter describes the design considerations taken in order to parallelize algorithms in the LTE physical layer of the simulation model.

7.1 Analysis of requirements and design considerations

In the introduction of this report, section 1.4, several minimum requirements involving design issues are stated. These requirements include the following to be designed and implemented for a prototype simulator:

- Design and implement a multithreaded simulator that exploits parallelism in physical layer models.
- Provide transparent implementation of concurrency constructs.
- Design tools for the verification of deterministic behaviour and program correctness.

A wide range of APIs exist for multithreading and distributed parallelism in Java, as previously described in section 4.7. Auto-parallelizing compilers offer the least amount of work considering re-design and modifications to the current sequential simulator. However the auto-parallelizing compilers or byte code transformation tools available, such as *MANTA* [29] and *javab* [30], are prototype softwares which have originated from universities and have mainly been proven for small artificial code examples rather than real-world complex applications. These compilers also tend to support only a limited set, or early versions of the Java JDK. Other APIs for parallel Java include rich frameworks such as Parallel Java [26], JPPF [27] and JOMP [28] and are primarily targets for distributed computing in grid networks. However, building a customized light weight framework for parallel execution based on the Java concurrency API [24] is chosen as the design strategy

due to a higher level of control and to avoid dependencies on third party developers.

In order to provide multithreaded execution of physical layer models, it is required to analyze data dependencies and decompose models and algorithms in order to execute different logical parts concurrently. A concurrency framework designed for this purpose can provide transparent implementation through encapsulation of multithreading constructs, synchronization and scheduling according to requirements in section 1.4.

7.2 Performance bottlenecks in current design

Using a profiler tool it is possible to determine which part of the simulation model is the most dominant regarding computational time, i.e. in which methods and algorithms the CPU spends most time executing. As previously stated, the physical layer model involves the most CPU-intense calculations and this is clearly verified by the profiler results in Table 2, where the *beginSubframe()* and *endSubframe()* methods of the *BasicLteManager* class, which is the physical layer coordinating function has the highest CPU-time ratio among most scenarios.

Table 2: Total percent of execution time (CPU time) spend in physical layer sub frame computations. Data based on call-tree profiling results.

	% of total execution time		
Scenario	**beginSubframe()**	**endSubframe()**	**Other methods**
Detailed radio model	1.0	72.7	26.3
File download	2.3	63.3	34.4
File upload	7.3	42.9	49.8
VoIP	13.1	33.6	53.3

The data in Table 2 is calculated from results obtained by running *JProfiler* call tree profiling, which measures the execution time on a method basis and summarizes execution times hierarchically through the method call tree. This is further verified by running “hot spot” profiling which identifies which particular methods constitutes bottle necks in the system, whereas many methods relating to the physical layer models were the most dominant. No details regarding profiling

results can be presented here based on confidentiality reasons relating to Ericsson.

The differences between the scenarios in Table 2 depend on several factors including the number of users, traffic models and computational models. Hence, task granularity and algorithm regularity is highly dynamic and is a function of the scenario parameters which are passed to the simulator for configuration.

Based on the profiler data presented in Table 2, equation 4.3 can be used to calculate an estimate of the theoretical speedup achieved for N processors, assuming that the sub-frame computations can be performed in parallel without synchronization or overhead. However, this is impossible in reality, but the result can be used as a heuristic for the potential gains of parallelizing the physical layer of the LTE simulator. A theoretical estimate of ideal multithreaded LTE simulator speedup is illustrated in Figure 11, which is based on Amdahl's law and the profiler data presented in Table 2.

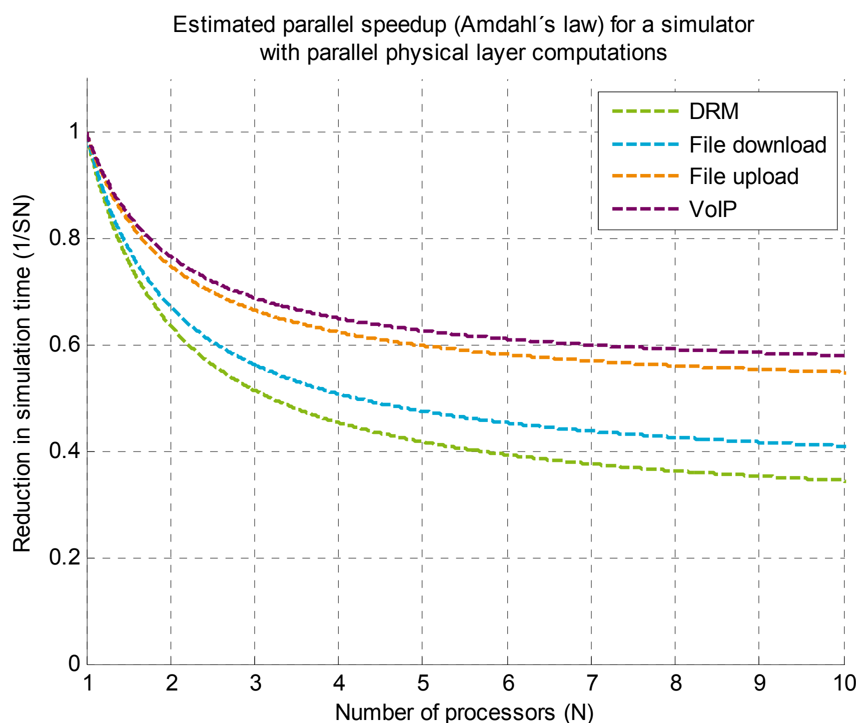


Figure 11: Estimated theoretical parallel speedup for a simulator with multithreaded physical layer computations for the different considered scenarios.

As can be seen in Figure 11 the greatest speedup is expected when executing the “Detailed radio model” and the “File download” scenarios since these scenarios spend more CPU-time executing physical layer computational models than the “File upload” or “VoIP” scenarios. It should also be noted how the speedup is limited by the sequential fraction of the program.

7.3 Analysis of the LTE physical layer model

The physical layer model provides higher layers with mapping of logical and transport channels to physical channels. Processing within the physical layer is initiated by an LTE sub-frame timer that starts a processing chain at each sub-frame. The physical layer model computations include initiation of measurements, control channel transmission and data transmission which is performed at the beginning of each LTE sub frame. The last processing stage at the beginning of a LTE sub frame is to update the uplink and downlink multi-path tables of each UE.

Each simulated LTE sub frame ends with a processing chain of measurements, control channel transmissions and data transmissions on the physical layer channels as illustrated in Figure 12. The measurements involve: sounding, downlink path loss computations, uplink interference calculations and downlink quality estimations. The results from these measurements are then used to update the state of each UE or base station.

After all the measurements have been completed, control channel transmission and reception are simulated. The PDCCH is used to transmit downlink control information, mainly scheduling decisions that are required for the reception of PDSCH and for scheduling grants enabling transmission on the PUSCH. Hence, the transmission on PUSCH and PDSCH cannot be conducted before the PDCCH processing is completed. Run-time tests with deterministic verification has also confirmed that data transmission and reception cannot be processed before quality measurements and control channel transmission and reception is completed.

The last step in the physical layer processing is to clear the transmission data from the current sub frame. Naturally this can not be performed

before all transmissions are completed, which is the reason for clearing uplink and downlink transmissions at the last processing stage.

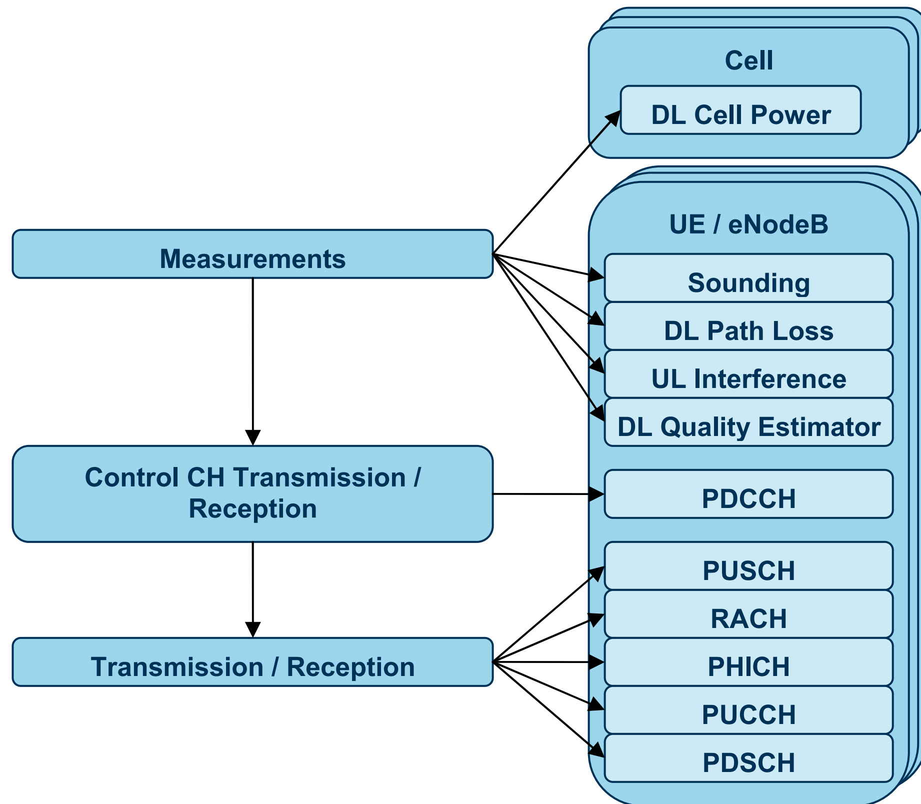


Figure 12: Physical layer processing steps and affected models for a LTE sub frame.

7.4 Data and control dependency analysis

The physical layer model is naturally decomposed to different measurement, control channel transmission and physical channel transmission tasks as illustrated in Figure 12 due to the embarrassingly parallel nature of the model. Each type of measurement involves no data dependencies between each other and each measurement algorithm may be run in parallel over several different cells, base stations or UEs, depending on the particular algorithm. Measurements and control channel transmissions are independent from each other in the current model and can be processed in parallel. It is, however, noticeable that control channel transmission is not very computation intensive compared to quality measurements.

Simulation of data transmission and reception is based on the results from control channel transmission and reception and hence control

channel transmission must be completed before data transmission is initiated. Since all UEs have their own channels for transmission and reception they can be computed in parallel. However, current sub frame transmissions can not be cleared before all transmissions are complete. This can be seen as a control dependency according to the taxonomy of M. O. Tokhi et al. [19], even through this relationship is rather obvious. Hence, it is vital to wait for all physical layer transmissions to be completed and cleared before continuing the processing of other events from the simulator event queue.

Most physical layer models calculate results such as SINR, MMSE, OK flags etc. to be provided to higher layers. This is currently conducted by sending data directly from measurement and channel models to higher layers of associated UEs as soon as results have been calculated. Higher layers might then in turn push events to the event queue of the simulator. In a parallel scenario with non-deterministic execution this may result in stochastic behavior (output) of the simulator. Hence results from physical layer models must be synchronized properly in order to provide deterministic results.

7.5 Task-oriented concurrency framework

To enable concurrent multithreaded execution of the program and hide implementation details in order to provide transparency for the developer, a task-oriented concurrency framework is proposed. A task in this case is defined as an arbitrary segment of code to be executed asynchronously. Tasks are further classified into two sub types:

- *Unordered task* – A segment of code that is executed concurrently on a separate thread and processes local data where the order of execution is not essential.
- *Ordered task* – A segment of code that is executed concurrently on a separate thread and processes non-shared data and returns a result to be processed sequentially (shared or local data).

This can be directly translated to an UML class description where *AbstractTask* is a generalization of *UnorderedTask* and *OrderedTask* as illustrated in Figure 13. A coordinating class *TaskManager*, is also created to encapsulate task scheduling, execution and synchronization among

asynchronous tasks. Each simulator has a task manager instance to handle its tasks.

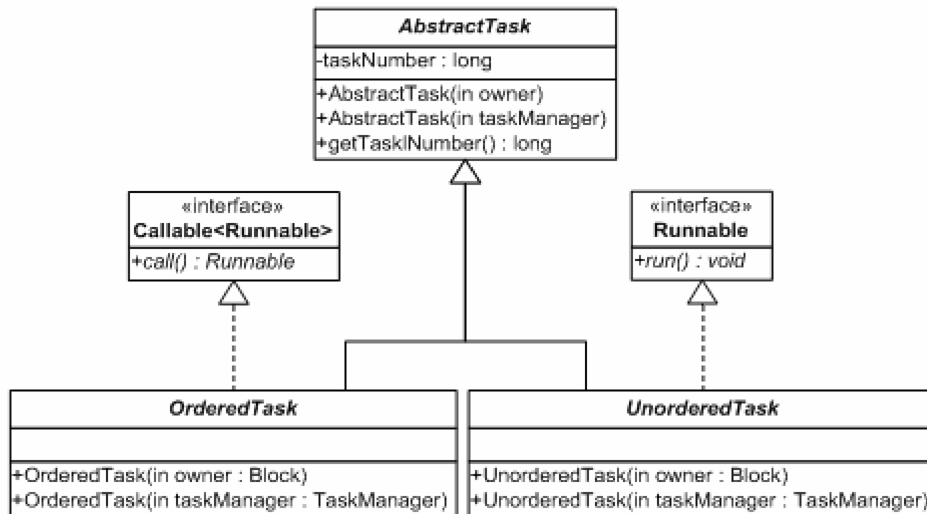


Figure 13: UML-class diagram of task class hierarchy.

The classes *UnorderedTask* and *OrderedTask* implement the *Runnable* and *Callable<Runnable>* interfaces of the Java concurrency API as abstract methods which makes it possible to use the constructors of the classes to create anonymous class instances with specialized *run()* or *call()* methods in order to provide arbitrary code to be executed asynchronously through class specialization. This is illustrated with source code examples in Figure 14 and Figure 15.

```
new UnorderedTask(this) {
   public void run() {
      async_operation();
   }
};
```

Figure 14: Java code example of *UnorderedTask* usage to encapsulate a code segment to be executed asynchronously on a separate thread.

In the constructor of the task classes, the object can submit itself for execution by calling the submit method of the coordinating *TaskManager* class. The task manager reference can be obtained from the object that created the task (derived from *Block*) through its reference to the simulator instance. Hence, the *this* parameter should be passed to the constructor when creating a task.

```
new OrderedTask(this) {
   public Runnable call() {
      final Result result = async_calc();
      return new Runnable() {
         public void run() {
            process(result);
         }
      };
   }
};
```

Figure 15: Java code example of *OrderedTask* usage to encapsulate a code segment to be executed asynchronously on a separate thread and return another code segment to be executed sequentially on the main thread.

OrderedTask objects return a *Runnable* object (see Figure 15) which makes it possible to return arbitrary code that should be executed sequentially. This allows a high degree of flexibility as asynchronous computations in the call method body can return an asynchronously computed result to be processed sequentially in logical order. The *Runnable* object returned by an ordered task is automatically executed in logical order by the task manager when asynchronous computations are performed by storing submitted tasks in a queue.

7.6 Task management and asynchronous execution

A task manager can be bound to a simulator instance and manage the scheduling and execution of arbitrary asynchronous tasks using a thread pool in order to minimize thread creation overhead. This is achieved by executing tasks on idle threads and reusing threads for queued tasks as soon as executing a previous task has been completed. The *ExecutorService* of the Java concurrency API [24] enables the management of a fixed size thread pool and manages task scheduling and execution of classes that implement the *Runnable* or *Callable* interfaces. The *Runnable* interface defines a method that may be executed by a thread and the *Callable* interface a method to be executed on a separate thread that returns a value. Tasks are scheduled (queued) in a *first-come-first-served* (FCFS) manner, which may not provide optimal load balance, but is better than a random scheduling as imbalance will last no longer than the length of one task. This is the standard scheduling algorithm as implemented in the *ExecutorService* class of the Java concurrency API 1.6. Figure 16 illustrates how tasks are submitted, queued at the task manager and executed by acquiring idle

threads from the thread pool. Blocking queues are used to store submitted jobs and non-blocking queues to store *Future* objects in order to be able to implement synchronization and execution of *Runnable* objects returned by ordered tasks.

The size of the thread pool is set equal to the number of processors of the system in order to minimize overhead due to context switching and to maximize hardware utilization. However, the thread size can easily be set to any number, which is advantageous for performing performance experiments with different number of threads and increased hardware utilization control.

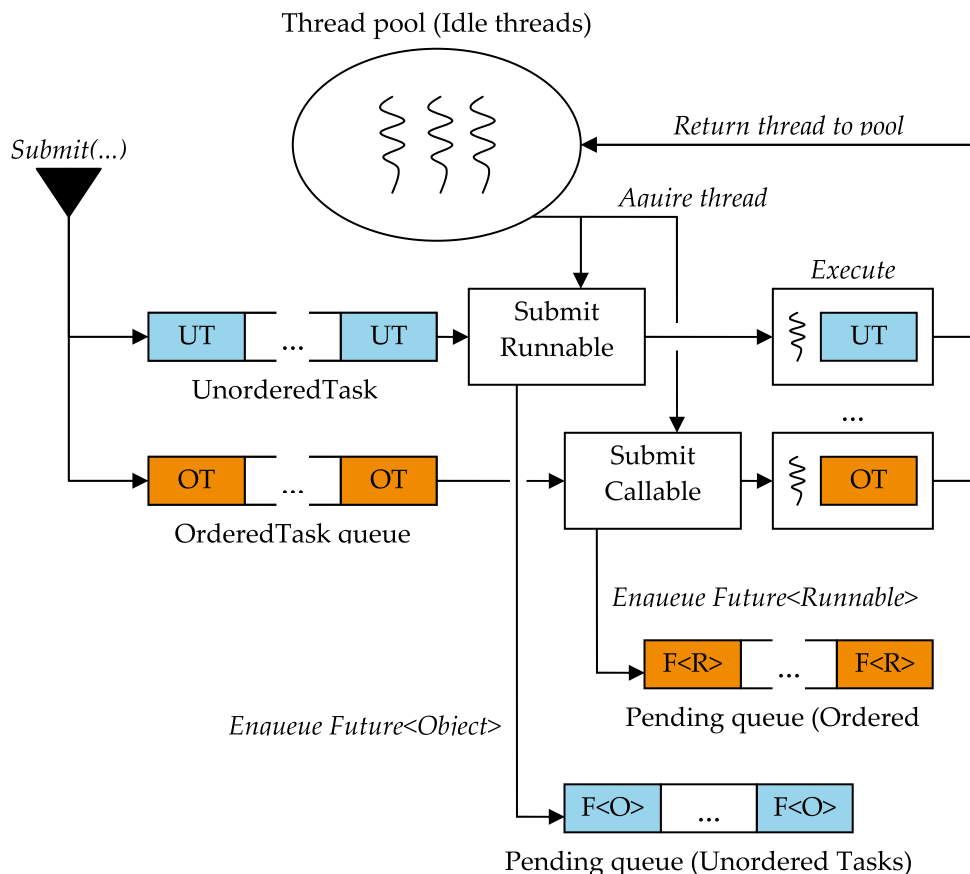


Figure 16: Block schema describing task submission and execution of tasks.

The task manager is heavily based upon the functionality provided by the Java concurrency API *ExecutorService*, but also supports coordination and ordering of tasks. The most important functionality of the task manager is the possibility of submitting tasks (*UnorderedTask* and

OrderedTask class specializations) for the execution and synchronization of these asynchronous operations. This enables the so called *fork-join design pattern* to be used in software design, as described by T. G. Mattson et al. [5]. The fork-join design pattern actually means that at a certain point in the program, work is decomposed so as to be executed on different processors of the system for asynchronous execution (fork). The program then requires to wait for the results from these asynchronous computations in order to be able to continue execution at a certain point where data dependency is present (join). The fork/join pattern is very useful if the number of tasks to be executed is dynamic and changes over time which is the case in LTE simulation. This behavior is conceptually illustrated in Figure 17.

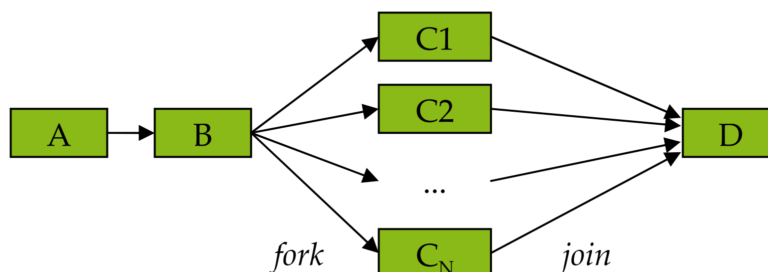


Figure 17: Conceptual illustration of fork/join pattern where task C is divided into several independent sub-tasks that need to be combined (joined) in order to execute task D.

The *fork* operation in this sense means that tasks are created by instantiating specializations of *UnorderedTask* and/or *OrderedTask* classes. The task manager will schedule these tasks for execution and run any returned *Runnable* objects sequentially on the main thread. However, in order to perform a join operation it must be verified that all asynchronous operations have completed their execution. As illustrated in Figure 16, a *Future*<*T*> class template instance is obtained when tasks are submitted for execution. These objects are part of the Java concurrency API and support the main thread to be blocked, waiting for tasks to complete. The join operation is therefore accomplished by simply queuing *Future*<Object> and *Future*<*Runnable*> objects and removing them from the queue of pending tasks as they are completed. When all pending task queues are empty there are no more asynchronously executing tasks and the program can safely proceed. A UML diagram illustrating the complete task-oriented framework design is presented in Appendix C, Figure 36.

7.7 Orchestration of tasks and data in LTE physical layer

Based on the results from the analysis of data and control dependencies in the LTE physical layer model (see section 7.3 and 7.4) it is possible to introduce a multithreaded execution model which resembles the task-oriented framework proposed in sections 7.5 and 7.6.

Currently each processing stage (measurements, control channel transmission, data transmission etc.) in the physical layer simulation model is implemented as loop constructs that iterate in sequence over a collection of objects with a common interface. This is the case for processing in both *beginSubframe()* and *endSubframe()* methods of the LTE physical layer. For this reasons all independent physical layer algorithms may be potential targets for loop-parallelism. However, the dynamic nature of algorithms and task granularity may cause thread management and synchronization overhead to be more expensive than the speedup gained by mapping algorithms, or tasks to several processors.

Using a common interface for all physical layer model entities is practical when designing and implementing software, but the abstraction provided by such an interface is an obstacle when attempting to implement multithreading concepts that are efficient in every scenario. Obviously, simple parallel loop constructs would not be an efficient solution.

In order to maximize efficiency by using multithreading techniques and minimize overhead caused by thread management it is more convenient to loop sequentially over physical layer models, but create asynchronous tasks in the methods of the physical layer models (classes) that are called from loop constructs. In this manner, complex algorithms can take advantage of multi processor hardware, while simpler constructs or quick-returns might benefit from executing on the main thread, thus avoiding the thread management overhead. Simple run-time tests indicated that this approach provides a higher speedup than parallel loop implementations. Determining which algorithms are suitable for being implemented as asynchronous tasks can be determined by profiling data.

As previously stated in section 7.4, most physical layer models, represented by Java classes, sends computed results to higher layers as

soon as they have been calculated, which may result in stochastic behavior of the simulator due to out-of-order event generation. The *OrderedTask* class is designed to solve this particular situation. By implementing the *call* method of the abstract *OrderedTask* and place calculations to be performed asynchronously in this method it is possible to queue the results by placing code that sends data to high layers in a *Runnable* object that is returned by the call method. In this manner the internal structure of the physical layer models do not require any dramatic changes in order to benefit from multithreading parallelism. Additional data that might be necessary to access can be declared as constant data (*final* in Java) in order to be accessed by these anonymous class instances. Since the *OrderedTask* instances are created from the main thread, queued result objects will be ordered in a logical order. These queued results will then be processed when the task manager is called to synchronize. Synchronization points in the program can be determined from the data and control dependency analysis in section 7.4.

All measurements and control channel transmissions have no data dependencies in the LTE physical layer model and can be executed in parallel. However, data transmission and reception on physical layer channels PDSCH, PUSCH etc. cannot be processed until all measurements and control channel transmission are complete. Hence, synchronization is required before data transmission and reception are initiated as illustrated in Figure 18.

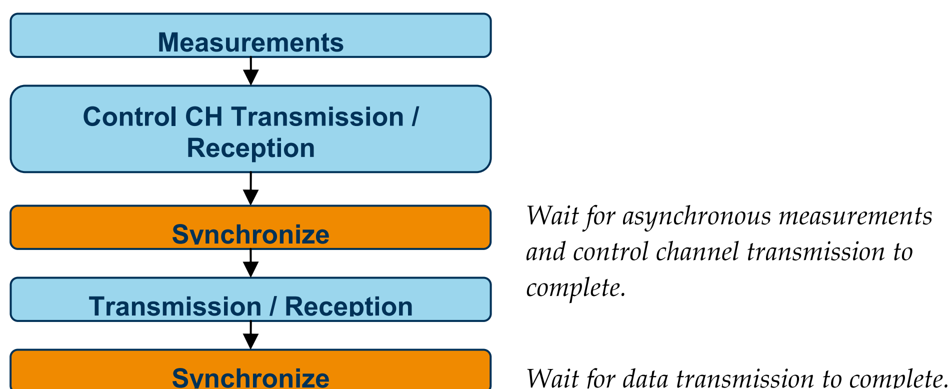


Figure 18: Conceptual model of parallel execution and synchronization of LTE physical layer models.

Synchronization is also necessary after all asynchronous data transmission and reception tasks have been scheduled in order to wait for the transmission to complete before processing other events from the simulator event queue.

Using this approach to provide multithreaded execution, parallelism is exploited over all cells and UE physical layer channels. It should be noted that some simpler operations such as methods called to clear transmissions or update states of physical layer entities that do not return any results to be reported to higher layers (not described here), can benefit from multithreading techniques through the *UnorderedTask* class which provides less overhead than the *OrderedTask* class.

7.8 Preserving deterministic behavior

7.8.1 Definition of deterministic simulation

Deterministic behavior can be defined as an unbroken chain of prior occurrences. A deterministic system in this sense is a system in which no randomness is involved in the development of future states of the system. Deterministic models thus produce the same output for a given starting condition. This implies that in event-driven simulation, the chain of events and their time stamps will be equivalent every time the simulation is executed, regardless of the hardware or state of the executing system. Hence, deterministic simulation implies that the results can be repeated.

This apparently becomes a problem in a parallel computing environment with non-deterministic execution as the order of object creation, function calls and order of events may alter from time to time depending on the operating system thread scheduling, the number of processors used and the current system load.

7.8.2 Synchronization and ordering of log events

The simulator output or simulator log is generated by log handler objects that are called when simulation data is logged as previously described in section 3.2.3. Since the simulator by definition is deterministic, at least when executing on the same system, two equal simulation logs imply two equal simulation states, assuming that sufficient information about the simulation state is written to the

simulation log. If a simulation state for parallel and sequential execution differs, simulation is not deterministic as this implies that the simulation state or result is dependent on the order of execution which is stochastic. This problem is illustrated by example in Figure 19, where two otherwise independent executing concurrent tasks send data to the logging system in invalid logical order.

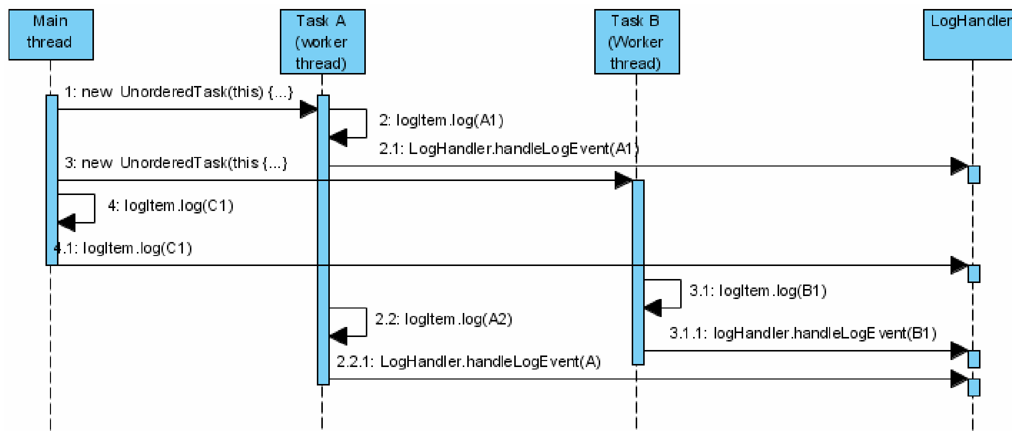
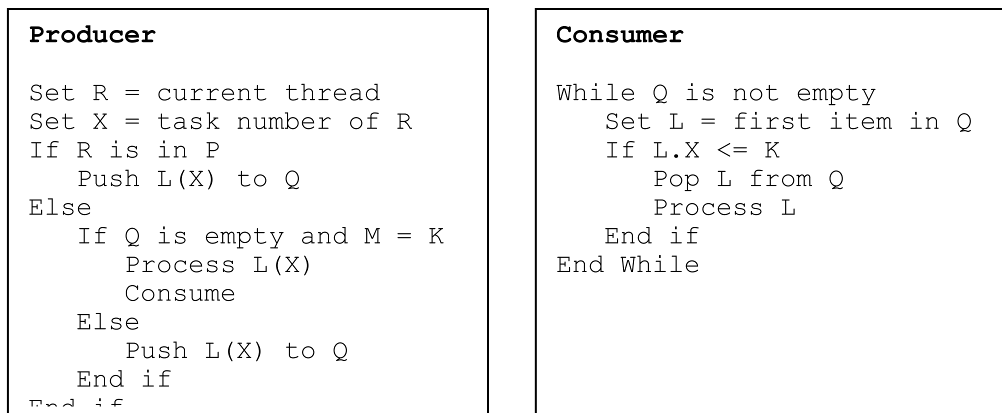


Figure 19: Sequence diagram illustrating logging out-of-sequence problem.

Since one of the primary design goals is to provide transparency of concurrency constructs to the developer, a synchronization algorithm for log events is required. If log events are automatically ordered, logging can be implemented in exactly the same manner as in a sequential program. When data is sent to a log item, a log event is generated and distributed to all registered log handlers. Hence, synchronization of log events means that log events should not be sent to any log handler if there is a logically preceding task executing that could potentially also generate a log event. Logical order can be determined by introducing a serial number that is assigned to each concurrent task when it is created. Concurrent tasks are spawned in sequential order on the main thread, which in turn will imply that the serial number is incremented for each generated task. By associating a scheduled concurrent task with the ID of a thread that is about to be executed in a hash table, it is possible to keep track of executing threads and the logical order in which the log events should be processed. To minimize overhead, log events are processed directly if there is no scheduled or pending task. However, if there are scheduled or pending tasks, log events are sorted using a specialized dynamic priority queue

that sorts log events based on the associated task serial number. For log events with equal task serial numbers, a first-in-first-out (FIFO) strategy is applied.

The algorithm to order log events can be divided into a producer and a consumer part as described by the following pseudo code:



```
Producer

Set R = current thread
Set X = task number of R
If R is in P
   Push L(X) to Q
Else
   If Q is empty and M = K
      Process L(X)
      Consume
   Else
      Push L(X) to Q
   End if
End if
```

```
Consumer

While Q is not empty
   Set L = first item in Q
   If L.X <= K
      Pop L from Q
      Process L
   End if
End While
```

Figure 20: Producer and consumer algorithms for log event processing.

where *R* is the current thread, *P* is the set of currently executing (registered) threads, *L(X)* is a log event with associated task number (priority) *X*, *M* is the total number of created tasks, *K* is the total number of completed tasks, *N* is the zero-based serial number of a task ($N \leq M$), and *Q* is a priority queue based on *X*. This makes it possible to consume log events on the main thread as stated in Figure 20, or in parallel by running the consumer part on a separate thread. This would imply that the producer thread never calls *consume*, instead the main thread can continue normal execution. However, parallel log event processing has only been prepared for and not implemented in the prototype design, since there is no current guarantee that all log handler implementations are thread-safe.

It is vital for the solution described above that the counters for created and completed tasks are implemented as atomic counters to provide cross-thread data integrity. However, the hash table used to register and unregister executing threads does not require to be atomic since registration can be handled by the task manager logic on the main thread. The priority queue manages the sorting of queued log events, which is illustrated by example in Figure 21 according to the sequence diagram in Figure 19.

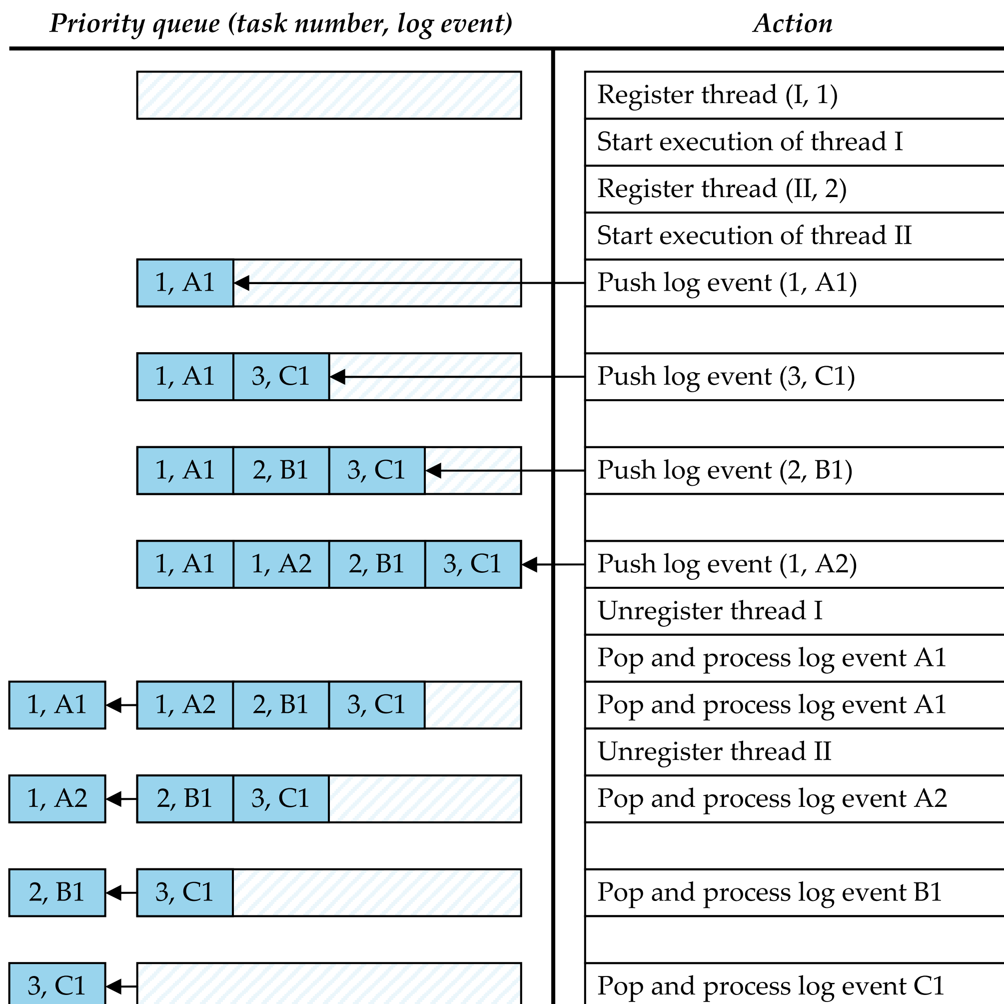


Figure 21: Example of how the priority queue is used to order log events according to the scenario illustrated in the sequence diagram in Figure 19.

7.8.3 Verification of deterministic behavior

In order to verify deterministic behavior, sequential and parallel simulation output using the same seed and scenario parameters must be equal. Simulation output is based on log data and in order to reflect the simulation state as accurately as possible all deterministic log items should be compared. However, this will result in thousands or millions of log events. Storing all this data in memory would require significant memory resources and hence it is more efficient to use a checksum approach in order to minimize memory requirements.

Java has built in support for 64-bit cyclic redundancy checksums (CRC32 class) [49], which are based on IETF RFC 1952 [50]. The CRC32

implementation is a polynomial checksum of order 32 that calculates a 64-bit fixed-length binary sequence from raw data and can be used as an error-detecting code. Using the CRC32 logic a specialized log handler can be created that accumulates a checksum ε for the current state based on the byte stream conversion of log event data according to:

$$\varepsilon_n = CRC_{32}(\varepsilon_{n-1}, x_n) \qquad (7.5)$$

where ε_n is a checksum containing the simulator state history until the n:th log event (state) derived from the previous checksum state ε_{n-1} and the last log event x_n. In order to verify that a multithreaded implementation is deterministic, this is simply accomplished by adding a CRC32 based log handler to a sequential and parallel simulator and comparing the output as illustrated in Figure 22.

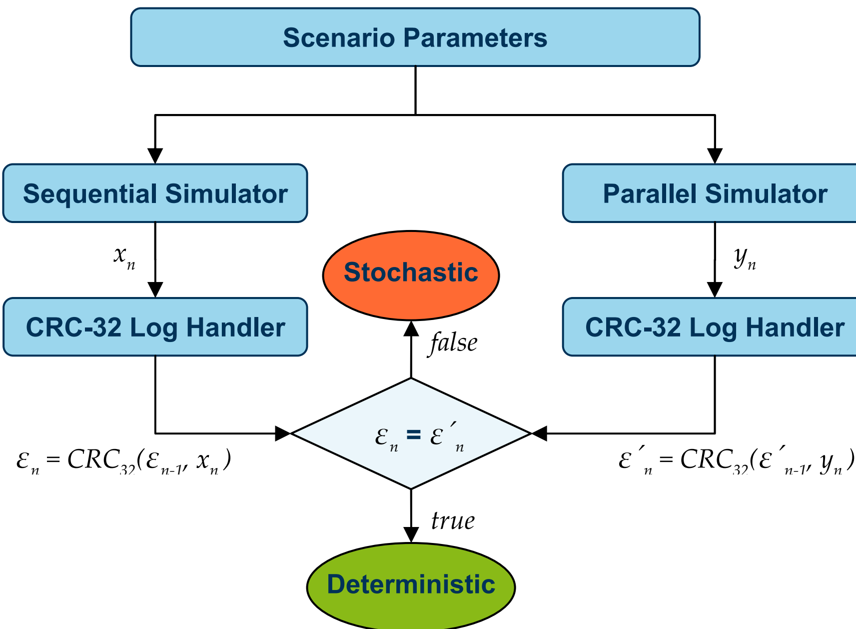


Figure 22: Conceptual model of checksum-based verification of deterministic behaviour.

It should be noted that each simulator sends data to the log handler every time output data is sent to the logging system. Hence it is also necessary to re-compute the CRC checksum according to equation 7.5 for every log event. If ε_n and ε'_n (see Figure 22) are compared every time n increases (a log event is generated) and equation 7.5 is recalculated, stochastic behavior can be detected at the log event level.

Simulation output, i.e. log events typically consists of values describing properties including transmission power, SINR, fading etc. for different model instances. Since most output data is calculated using mathematical models, errors in a parallel simulator are likely to propagate with time. Hence the probability of detecting stochastic behaviour increases as simulation time tends towards infinity which implies $n \rightarrow \infty$.

To automatically perform deterministic verification mechanisms similar to the mechanism described in Figure 22 unit tests prove to be very useful. At Ericsson *JUnit* is used to create unit tests to automatically verify the functionality of the simulator. It is thus possible to set up several scenarios in order to test the deterministic properties of parallel and sequential simulations automatically.

8 Results

The graphs presented in chapter 8 summarize the results of the performance measurements carried out according to the experimental methodology described in chapter 6.

8.1 Performance gain of multithreaded simulation

The graphs presented in Figure 23, Figure 24 and Figure 25 illustrate the reduction in average execution time T_N for the multithreaded prototype when executing the four different scenarios: I, II, III and IV (see section 6.3) for 10.0 seconds (simulation time) on system A, B and C respectively (see section 6.1). The dotted curve illustrates the expected ideal speedup, or actually estimated reduction in execution time (T_1 / S_N) according to Amdahl's law.

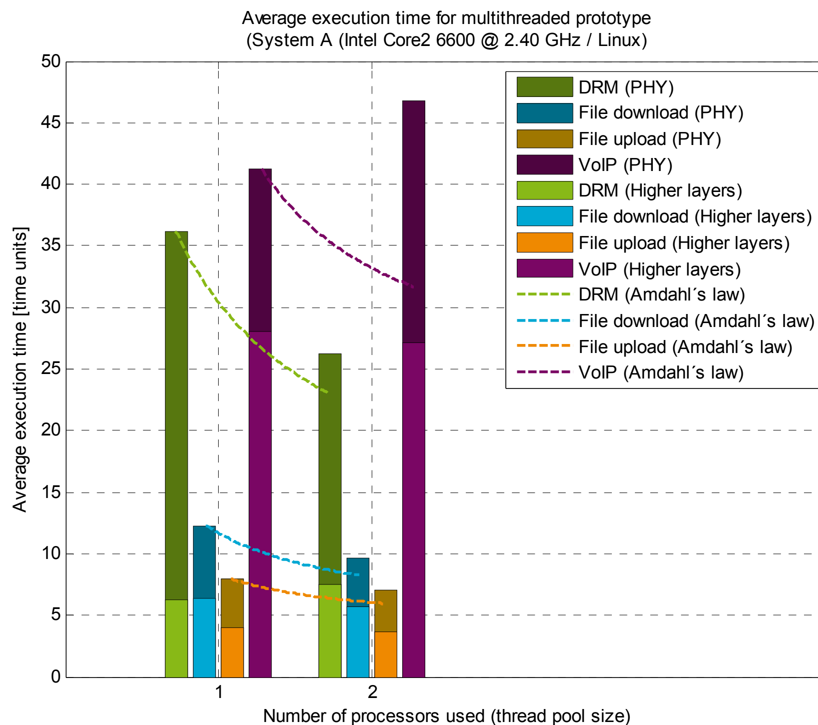


Figure 23: Average physical layer execution time for the parallel prototype implementation executing for 10.0 seconds (simulation time) on system A.

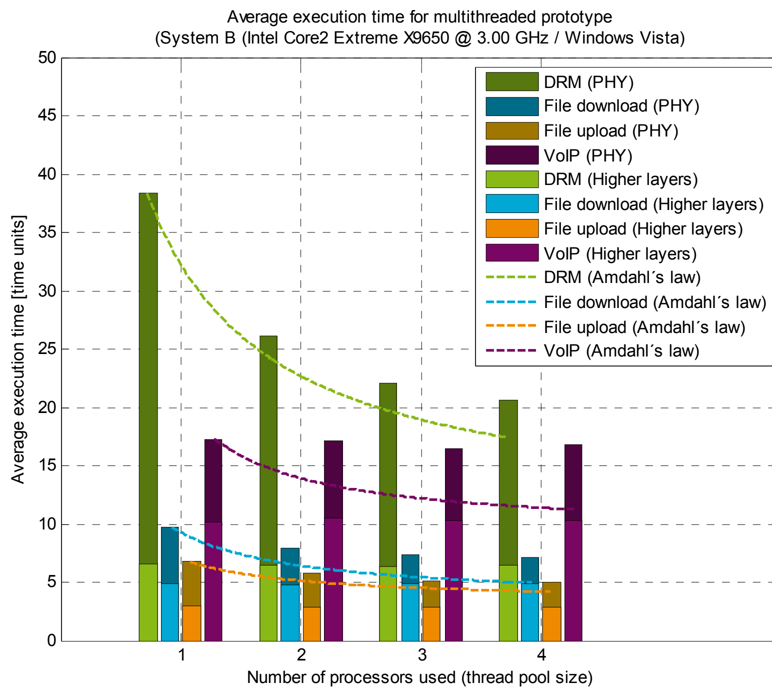


Figure 24: Average physical layer execution time for the parallel prototype implementation executing for 10.0 seconds (simulation time) on system B.

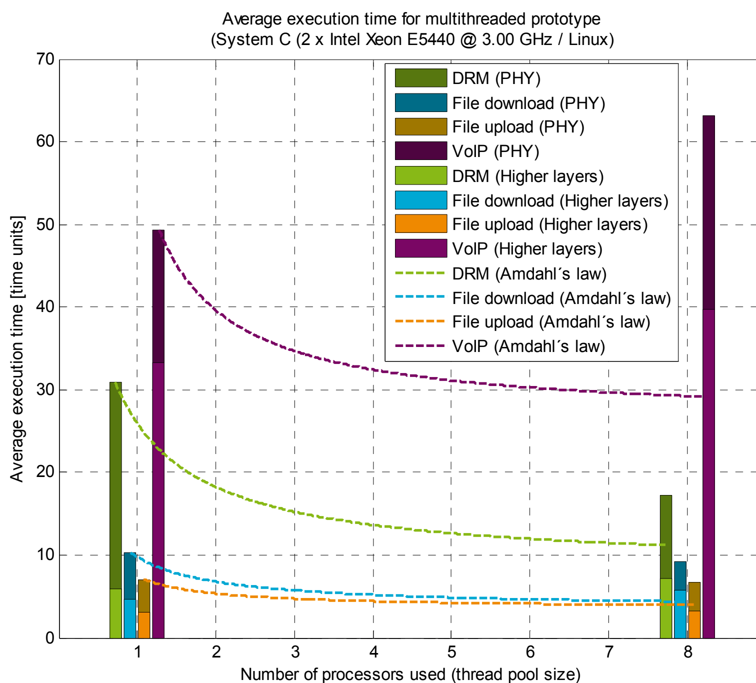


Figure 25: Average physical layer execution time for the parallel prototype implementation executing for 10.0 seconds (simulation time) on system C.

8.2 Execution time per simulation job when executing multiple simulation jobs

In order to compare simulation by parallel jobs (independent processes) and multithreading techniques Figure 26, Figure 27 and Figure 28 illustrate the average execution time per job on system A, B and C respectively. The leftmost bar group is the execution time of a single sequential job T_1. The second bar group is the theoretical ideal speedup T_1 / N (N defines number of processors on the system). The third bar group is the average execution time per single-threaded job, when running N such jobs in parallel (obtained by dividing the sum of all actual process execution times by N). The rightmost column is the average execution time T_N of a single job when executing the multithreaded prototype with a thread pool size of N.

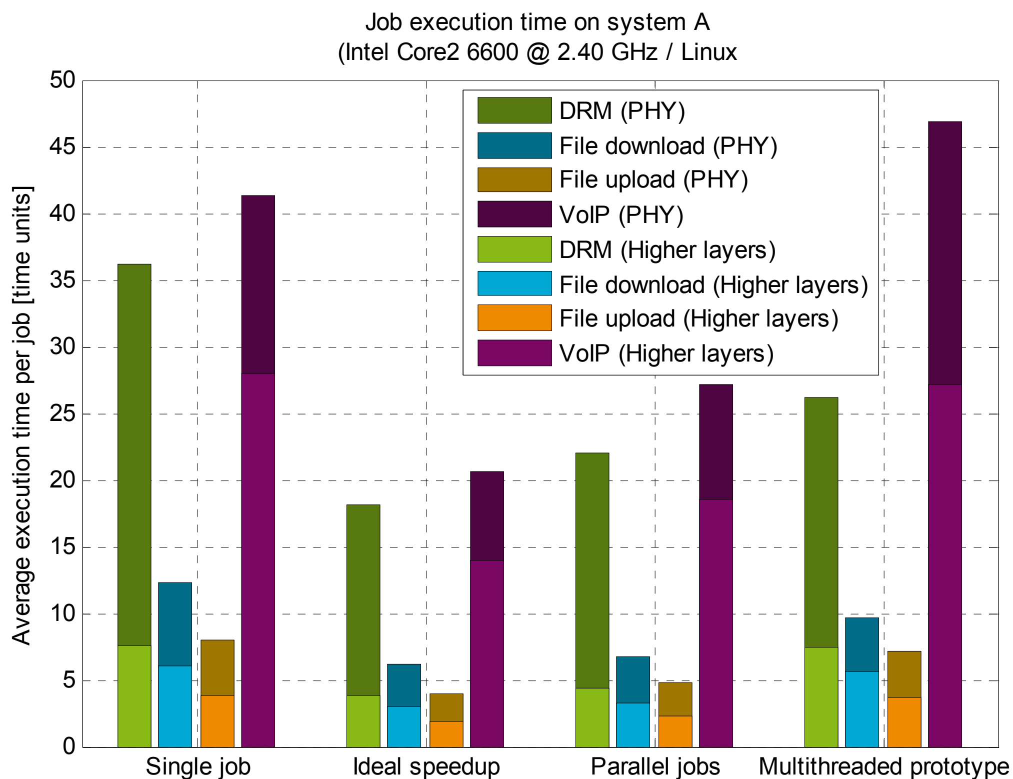


Figure 26: Average execution time per simulation job on system A, N = 2.

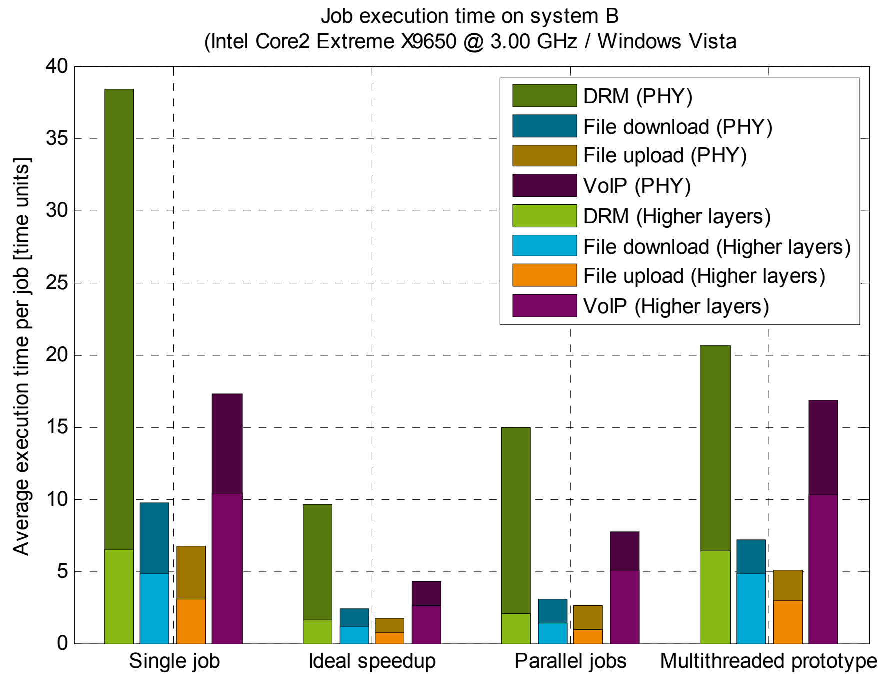


Figure 27: Average execution per simulation job on system B, number of processors N = 4.

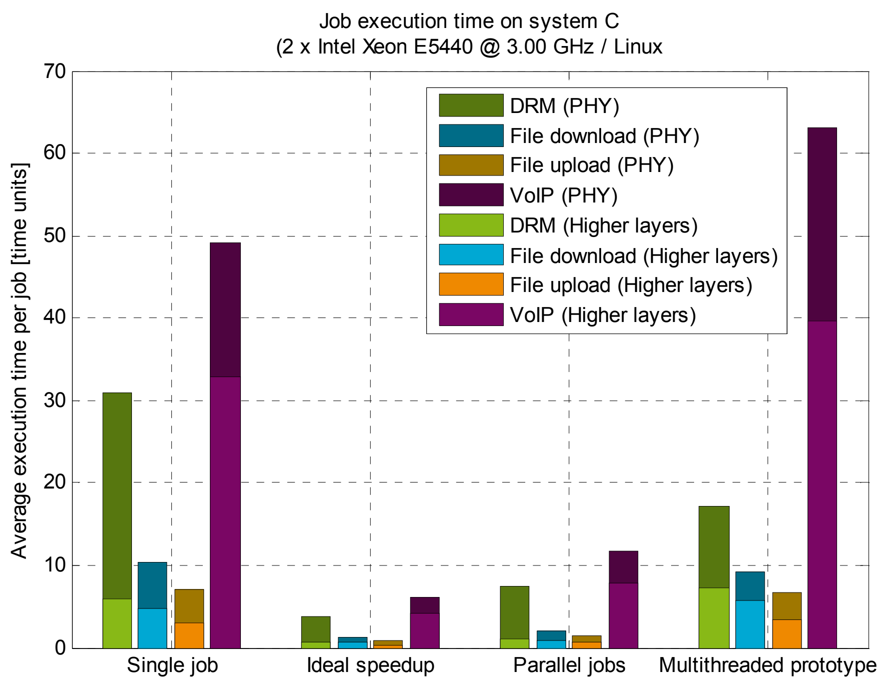


Figure 28: Average execution per simulation job on system C, number of processors N = 8.

8.3 Comparison of parallel jobs and an ideal multithreaded simulator

The graphs illustrated in Figure 29, Figure 30 and Figure 31 illustrate the measured average execution time per simulation job when running *N* such jobs in parallel on *N* processors (obtained by dividing the sum of all actual process execution times by *N*), compared to the expected ideal estimate of an ideal multithreaded simulator derived from Amdahl's law (T_1 / S_N). Two ideal estimates have been visualized: one speedup estimate for an ideal multithreaded simulator with a parallel physical layer (PHY) and one estimate for an ideal multithreaded simulator with both a parallel physical layer and a parallel *media access control* (MAC) layer, which would be the next natural extension in the further development of parallelising the simulator.

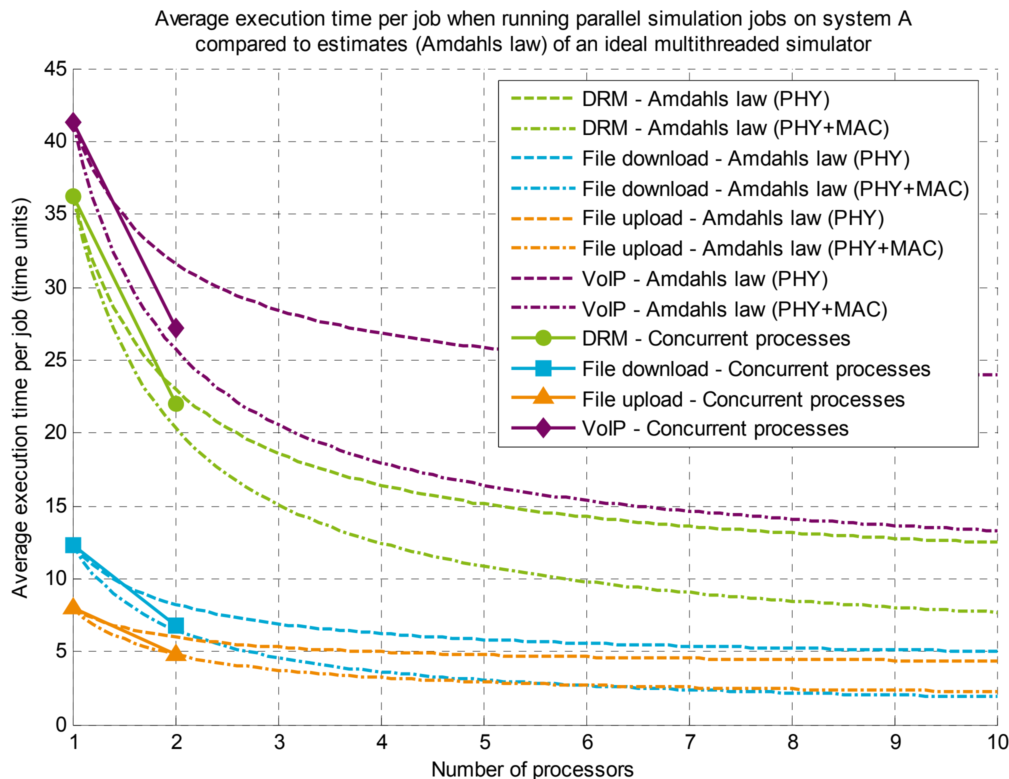


Figure 29: Average execution time of a single parallel simulation job executing on system A compared to ideal estimate of a multithreaded simulator.

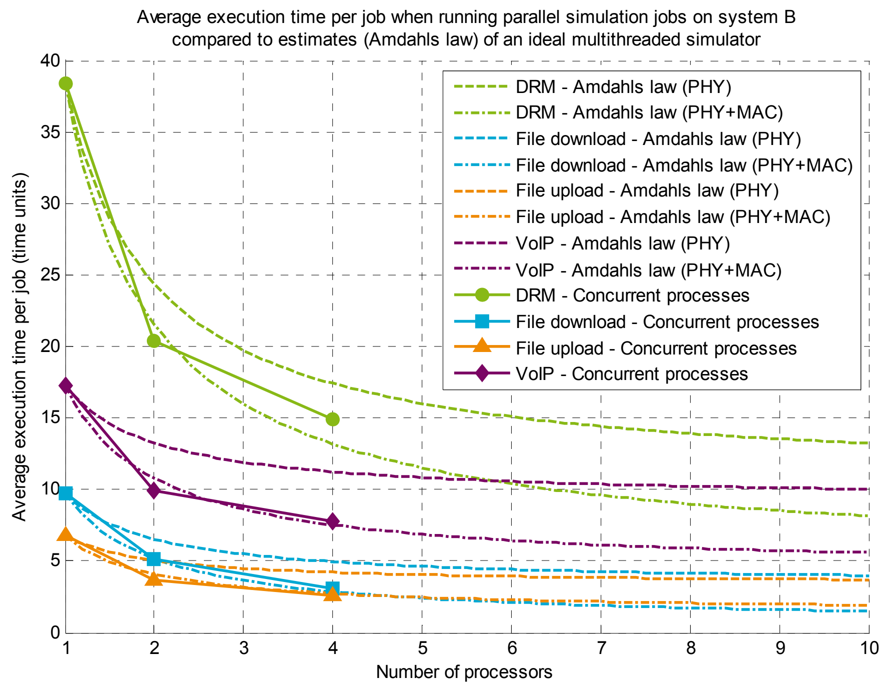


Figure 30: Average execution time of a single parallel simulation job executing on system B compared to ideal estimates of a multithreaded simulator.

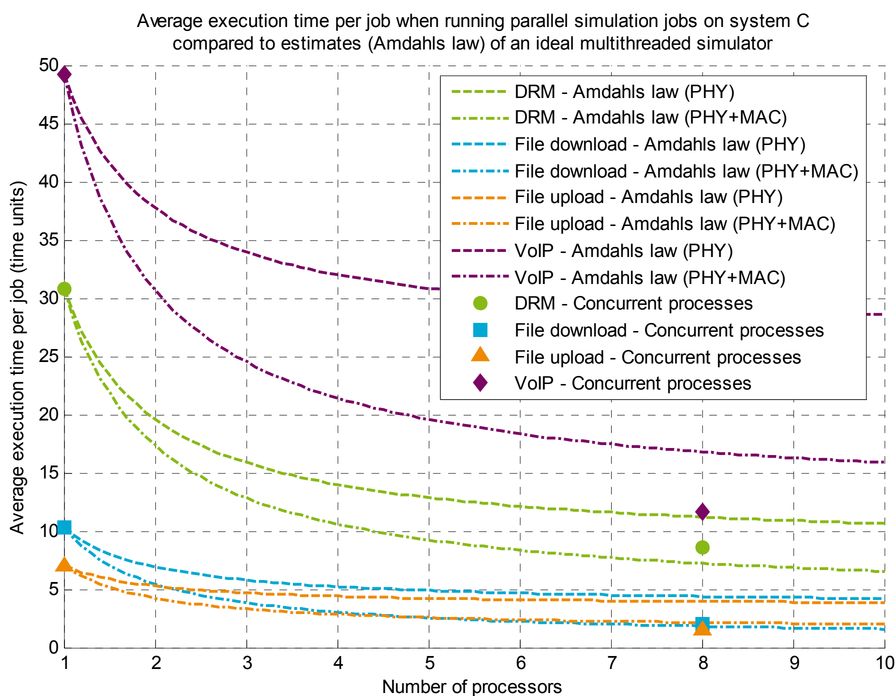


Figure 31: Average execution time of a single parallel simulation job executing on system C compared to ideal estimate of a multithreaded simulator.

8.4 Implementation transparency and task-oriented framework usability

This section is a summary of the received answers to the questionnaire in Appendix B which are presented as a whole in Appendix D. A total of 6 people answered the questionnaire. 50% of the consulted people are represented by research engineers at Ericsson and the other 50% is represented by people that are working with computer engineering outside of Ericsson.

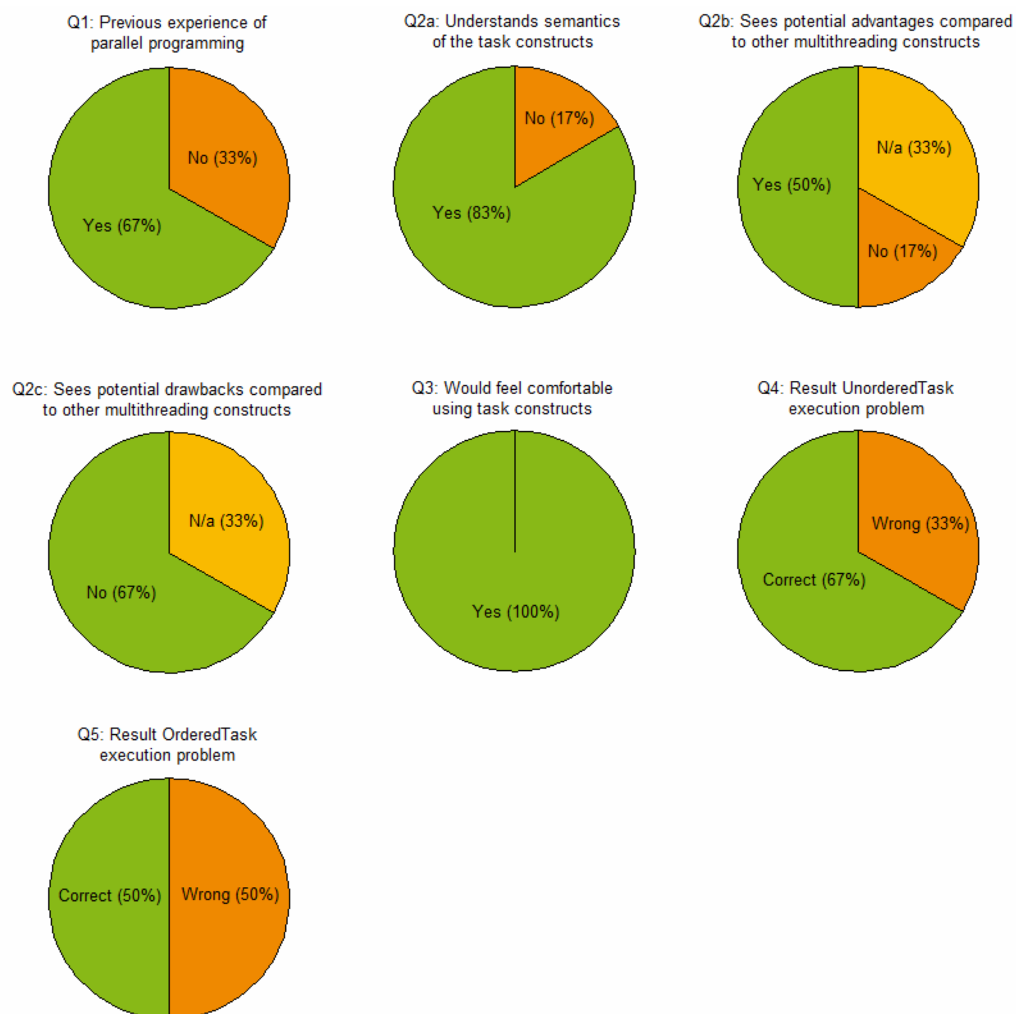


Figure 32: Summary of questionnaire results regarding task-oriented framework usability and readability.

As can be seen in Figure 32, 67% of the consulted people had previous experience of parallel programming and multithreading constructs from a variety of programming languages: C POSIX threads, Java, Microsoft

C# .NET. The results obtained for questions 2b and 2c had 33% of the results marked as *not applicable* (N/a). The reason for this is that these people did not have any previous experience of multithreading constructs to compare with. The results from question 4 and 5 have been quantified to be either completely correct or completely wrong.

The correct answer to question number 4 in Appendix B, which illustrates a code example for executing asynchronous operations using the *UnorderedTask* construct, is that nothing can be stated considering the order of execution. When the first *UnorderedTask* object is created it will submit itself for execution immediately and the same is true for the second task object that is created. Since the execution of the threads that executes these tasks and the main thread are non-deterministic and unsynchronized, the order of execution is impossible to determine.

The correct answer to question number 5 in Appendix B, which illustrates a code example for executing asynchronous operations using the *OrderedTask* construct is that "C" will be printed first, then "A" and lastly "B". The reason for this is that the print statements are placed within the *Runnable* object that is returned by the asynchronous operation. By definition of the task-oriented framework these objects will be executed in sequential order according to the order in which the task objects where created when the *completeTasks()* method of the *TaskManager* class is called. Since the print statement for "C" is called before *completeTasks()*, the order of execution of the print statements will be deterministic.

8.5 Verification of deterministic behavior

Automated CRC-32 based unit tests, as described in section 7.8.3, have been used to verify that the multithreaded prototype is deterministic and produces the same result as sequential simulation when running simulations for 15.0 seconds (simulation time) for all the four scenarios described in section 6.3. Since each LTE sub frame is 1.0 ms, this means that the checksums are calculated from all deterministic log messages generated during 15,000 LTE sub frames which approximately constitute data from 9×10^9 log messages.

9 Conclusions

9.1 Software design evaluation

The multithreaded prototype presented in this report enables multithreaded parallelism and computational concurrency over cells and UEs for LTE physical layer simulation. The proposed solution indicates that multithreading mechanisms suuch as: scheduling, execution and synchronization can be partly automated and provided through a minimal interface using concepts of class generalization, anonymous classes and standard Java concurrency interfaces, while still being scalable with hardware. The solution also shows that parallel computations in LTE simulations are possible by decomposing models and providing appropriate data synchronization.

9.1.1 Strengths and weaknesses of the task-oriented framework

The proposed task-oriented framework provides two abstract classes *OrderedTask* and *UnorderedTask* in order to trigger asynchronous computation. The possibility to use abstract and anonymous classes in Java makes it possible to create class specializations that encapsulate arbitrary code to be executed asynchronously which has resulted in a very flexible task-oriented framework. This framework is not limited to be used in simulation, even though the design of particulary the *OrderedTask* is somewhat based on how data is computed in the physical layer of the Ericsson LTE simulator. However, the synchronization mechanism of tasks is currently based on atomic counters and serial numbers associated to the task that prevents from creating sub-tasks in order to generate hierarchical execution trees. In order to increase flexibility of the task framework it might be of interest to evolve the synchronization algorithm to support hierarchical task relationships by introducing parent and child relationships between tasks in order to process critical parts of the code sequentially. However, this is better avoided since it might increase complexity of code and make debugging an even more difficult task.

Another disadvantage is that each created task object (*OrderedTask* or *UnorderedTask*) currently requires a direct or indirect reference

parameter passed to the constructor in order to submit the task for execution to the *TaskManager* class instance that is bound to the simulator. This could be avoided by allowing the *TaskManager* to be a singleton. However, this is a merely a question of design whether static instances should be tolerated or not.

9.1.2 Evaluation of questionnaire considering implementation readability and transparency

From the questionnaire result summaries in Figure 32, it is clear that the small group of consulted people in general have a positive attitude to how tasks are created and synchronized after only a brief introduction in the questionnaire description. This conclusion is based on the fact that 83% of the people considered the semantics of the constructs understandable and there were no inventions regarding potential drawbacks. However, considering the results from questions 4 and 5, which targets readability and to determine whether the semantics really are understandable, only 67% understood the underlying executional model of *UnorderedTasks* and only 50% understood the executional model of *OrderedTasks*. The results indicate that the brief introduction presented in the questionnaire in Appendix B is not sufficiently elaborative and that the task interface might require further clarification or re-design.

9.1.3 Evaluation of deterministic behavior

In order to provide deterministic output of the simulator during non-deterministic execution of physical layer models in the multithreaded prototype and still provide implementation transparency, a synchronization mechanism is implemented for log events. The mechanism provides automatic synchronization of log events to send to the log handlers in the same order they would be sent during sequential execution. The ordering of log events introduces additional overhead compared to allowing out-of-order presence of log output messages, caused by queuing and sorting of log events. However, this allows output to be compared from sequential and multithreaded simulation in a straight-forward manner which was an initial requirement for this work. However if only the content of each log message had to be equal, a method other than CRC checksums could have been used, for example XOR-operations between log messages. Using such an approach would have provided data of log messages to be compared, but would not

consider the order of log messages since they actually do not have anything to do with the simulator state. Simple XOR-operations would also have provided a faster method for verifying deterministic properties of the simulator.

9.2 Evaluation of multithreaded prototype performance

9.2.1 Anomalies between systems

When measuring the execution time of a multithreaded application and increasing the number of processors, a speedup is expected if the parallelism is scalable. As can be seen in Figure 23, Figure 24 and Figure 25 the speedup or reduction of simulator execution time increases as the number of processors *N* increases in most cases. This expectation is best illustrated by the results obtained when running experiments on system B. However, several anomalies between scenarios and systems (hardware and operating systems) are identified. For system A and system C, both Linux machines, a degradation of performance is observed for "VoIP" scenarios for both sequential and multithreaded execution. This is not present for system B, where the "detailed radio model" scenario takes the longest time to compute. The "VoIP" scenario is different from the other scenarios as it has many users but physical computations are not very CPU intense, i.e. has fine-grained task granularity. Since there are more users there are considerably more objects in memory than for the other scenarios which make the "VoIP" scenario much more memory intense. Since both Linux systems have 64-bit versions of the OS the performance penalty when running the "VoIP" scenario compared to running it on system B is probably primarily an effect caused by the extensive memory overhead due to 64-bit memory addresses.

9.2.2 Multithreading performance and multithreading overhead

The results obtained from experiments on system B in Figure 24 are the results that best conform to Amdahl's law. In this graph the overhead caused by thread management and synchronization is clearly visible in comparison to the estimated ideal reduction in execution time where the "detailed radio model" scenario provides the most promising speedup result. The reason for this is that the "detailed radio model" scenario is very computationally expensive considering physical layer models. Hence, it has a higher task granularity than for example the "VoIP"

scenario which provides fine-grained task granularity. Java and the proposed solution appear to have considerable overhead since a very large task granularity is necessary in order to provide a speedup close to the ideal. This apparently becomes a problem as different task granularities would be required to achieve a uniform speedup for different scenarios which would in turn increase the complexity of a multithreaded implementation according to the current simulator design. The results obtained from experiments A and C, see Figure 23 and Figure 25, verify this and also indicate that overhead is so large that a degradation of performance is obtained with worst case speedups of S_N= 0.68 and S_N= 0.70 when executing the multithreaded simulator for a "VoIP" scenario on system A and C respectively.

The best speedup results for multithreaded simulation are generally obtained for "detailed radio model" scenarios with speedups of S_N≈1.5, S_N≈2.2, S_N≈ 2.5 considering physical layer computations on system A, B and C respectively when the number of threads equals the number of processors. As can be seen in Figure 25, the speedups obtained on system C are far from that suggested by the estimates derived from profiler data and Amdahl's law in contrast to the results of system A and B (see Figure 23 and Figure 24). This is likely to be a result of increased communication overhead for intraprocessor communication since system C is equipped with two cooperating processors with four cores each. In general it has been observed for multithreaded physical layer computations that:

- $S_N \in [0.68, 1.58]$ when executed on system A.
- $S_N \in [1.06, 2.24]$ when executed on system B.
- $S_N \in [0.70, 2.48]$ when executed on system C.

It is not surprising that none of the multithreaded experiments has the ability to reach ideal speedups since overhead can not be avoided, but the most important observation may be the dramatic difference in speedup between "detailed radio model" scenarios and "VoIP" scenarios. This indicates that multithreaded application performance is very tightly related to computational grain size which makes it difficult

to design a general multithreaded simulator that provides considerable speedups for all imaginable scenarios.

9.3 Comparison of job parallelization and multithreading

In order to increase performance and reduce the execution time of simulation work at Ericsson Research, parallelism is currently exploited by distributing independent simulation jobs over the processors available on a particular system. This leads to increased memory access and memory requirements that scale linearly with the number of concurrent jobs. Ideally the number of concurrent jobs is set equal to the number of processors on the host system. Multithreading techniques are superior in most cases to speedup of a single application by increasing CPU utilization and dividing the work among processors. However, to determine the most efficient way to complete as many simulation jobs as possible in the shortest time possible it is vital to compare the execution times involved in running multiple independent jobs concurrently or running multiple multithreaded simulators in sequence.

From Figure 26, Figure 27 and Figure 28 it is clear that running parallel independent jobs is more efficient on all the systems used for experimental evaluation considering the total simulation time. This is no surprise since the multithreaded prototype only exploits parallelism in the physical layers and is limited by the sequential parts of the simulator. What is more important is to analyze how the speedup scales with N when considering execution in the physical layer. However, running parallel jobs also provides better speedup for physical layer computations on all systems, even if the differences in speedup are as small as 0.1 for the "detailed radio model" scenario executing on system B.

The main difference when comparing the different approaches to running parallel jobs and running multithreaded simulations is that parallel jobs offers a somewhat uniform speedup while the speedup obtained by the multithreaded simulator is close to the speedup of running parallel jobs in some cases, especially for "detailed radio model" scenarios and results in degradation of performance in some cases "VoIP". In general it has been observed for parallel jobs that:

- $S_N \in [1.53, 1.77]$ considering physical layer computations and $S_N \in [1.52, 1.81]$ considering total simulation on System A.
- $S_N \in [2.25, 2.94]$ considering physical layer computations and $S_N \in [2.23, 3.19]$ considering total simulation on System B.
- $S_N \in [3.90, 4.88]$ considering physical layer computations and $S_N \in [4.10, 5.22]$ considering total simulation on System C.

Generally it can be stated that parallel jobs provide an approximate uniform speedup according to equation 9.6 on all systems for all scenario types, where N is defined as the number of processors.

$$S_N \approx \frac{2N}{3}, N \geq 2 \quad (9.6)$$

As can be seen by the approximation in the equation the speedup obtained by running parallel jobs is far from ideal, but currently provides a somewhat uniform and higher speedup than multithreading techniques. From this it is clear that overhead caused by increased memory access and higher system bandwidth requirements (approximately 30% of execution time) is currently less than the overhead caused by Java thread management and synchronization.

9.4 Recommendations for future work

The current means of exploiting parallelism by distributing simulation jobs as independent processes is currently more efficient than multithreading techniques when executing multiple simulation jobs, and hence it is recommended in order to minimize the time required by LTE simulation by means of this technique. However, multithreading techniques have indicated considerable speedups in scenarios with high computational grain size and might be used to speedup single simulations. Incorporating multithreading techniques within the same source code however increases code complexity and obstructs code maintenance.

To reduce the overhead when running parallel jobs (a.k.a. distributed experiments) it should be possible to redesign the simulator environment application to run multiple simulator instances or contexts within the same JVM instead of replicated JVM instances, hence

reducing memory requirements. A high degree of parallelism similar to spawning several independent processes can then be exploited by running each simulator on a separate thread without synchronization between them within the same JVM or application/process context. This might result in a speedup greater than the results in this thesis have indicated due to reduced memory bandwidth requirements as well as reduced system bus congestion.

It is recommended that Ericsson Research should re-evaluate the gains of multithreading techniques in the future for several reasons. For example future hardware and CPU architectures might not as easily accommodate the distribution of parallel jobs and might favor multithreaded execution. Distribution of parallel jobs also increases memory requirements linearly according to the number of processors. As Java distributions evolves, the overhead of thread management and synchronization is also likely to decrease, which have already been indicated in alpha releases of the JDK 1.7. More advanced algorithms as for example *work-stealing* also provide more efficient task management. Algorithms that improve load balance might also be of interest to exploit in order to improve multithreading performance.

When designing simulators in the future, it is recommended to design them according to design patterns that are easily mapped to multiple processors already in use at the beginning of the software design work. Since the development of simulators at Ericsson involves many developers it would be profitable to incorporate design patterns and enforce parallelism through the simulator execution model instead of relying on special case solutions. Decomposed models where each model can be executed independently and use thread-safe queues as interfaces between models or observer/listener patterns where models are notified of value changes through interprocess messages might be of interest if a new multithreaded simulator were to be designed from scratch.

References

[1] E. Dahlman, S. Parkvall, J. Sköld and P. Beming, *3G Evolution – HSPA and LTE for Mobile Broadband*. 2nd edition. Oxford: Academic Press, 2008. ISBN: 978-0-12-374538-5.

[2] Intel Coorporation, "Parallel Computing: Background". http://www.intel.com/pressroom/kits/upcrc/ParallelComputing_backgrounder.pdf
Retrieved: 2009-10-12.

[3] AMD, "Multi-Core Processing with AMD". http://www.amd.com/us/products/technologies/multi-core-processing/Pages/multi-core-processing.aspx
Retrieved: 2009-10-12.

[4] L. Zhang, "Performance Analysis and Optimization of a Radio Network Simulator", KTH/ICT/ECS, 2008.

[5] T. G. Mattson, B. A. Sanders, B. L. Massingill, *Patterns for Parallel Programming*. Westford: Addision-Wesley, 2008. ISBN: 0321228111.

[6] Ericsson Research: "Introduction to simulator platform". *Ericsson confidential internally supplied document, 2008.*

[7] Sun Microsystems. "Developer Resources for Java Technology" http://java.sun.com/.
Retrieved: 2009-10-19.

[8] J. Zyren, W. McCoy, "Overview of the 3GPP Long Term Evolution Physical Layer", *Freescale Semiconductor White Paper*. http://www.freescale.com/files/wireless_comm/doc/white_paper/3GPPEVOLUTIONWP.pdf
Retrieved: 2009-10-12.

[9] J.J van de Beek, P. Ödling, S.K. Wilson and P.O. Börjesson, "Orthogonal Frequency-Division Multiplexing", Invited chapter in *Review of Radio Science 1996-1999. Comm. C: Signals and Systems*.

International Union of Radio Science (URSI), Oxford University Press, 1999. ISBN: 0198565712.

[10] A. E. Sheikh, A. A. Ajeeli and E. M. O. Abu-Taieh. *Simulation Modeling – Current Technologies and Applications*. London: IGI Publishing, 2008. ISBN: 978-1-59904-198-8.

[11] OPNET Technologies Inc®, “OPNET”. http://www.opnet.com Retrieved: 2009-10-12.

[12] University of Southern Carlifornia, “The Network Simulator – Ns 2”. http://www.isi.edu/nsnam/ns/
Retrieved: 2009-10-12.

[13] AWE Communications, “WinProp Software Suite”.
http://www.awe-communications.com/
Retrieved: 2009-10-12.

[14] J. Song and N. Cackov, “WARNSIMulator”.
http://www.ensc.sfu.ca/~ljilja/cnl/projects/warnsim/index.htm
Retrieved: 2009-10-12.

[15] The MathWorks™ Inc, “The MathWorks”.
http://www.mathworks.com/
Retrieved: 2009-10-12.

[16] Vienna University of Technology, “INTHFT: LTE Simulators”.
http://www.nt.tuwien.ac.at/about-us/staff/josep-colom-ikuno/lte-simulators/
Retrieved: 2009-10-12.

[17] Ericsson Research: “Introduction to simulators”. *Ericsson confidential internally supplied document, 2009.*

[18] Ericsson Research: “Simulator overview”. *Ericsson confidential internally supplied document, 2009.*

[19] M. O. Tokhi, M. A. Hossain and M. H. Shaheed, *Parallel Computing for Real-time Signal Processing and Control*. London: Springer-Verlag London Limited, 2003. ISBN: 1-85233-599-8.

[20] J. L. Hennessy, D. A. Patterson and Andrea C. Arpaci-Dusseau. *Computer Architecture: A Quantitative Approach - Fourth Edition*. CITY: San Francisco: Morgan Kaufmann Publishers, 2006. ISBN: 0-12370-490-1.

[21] J. Held, J. Bautista and S. Koehl. "From a Few Cores to Many: A Tera-scale Computing Research Overview". White paper, Intel Corporation.
http://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf
Retrieved: 2009-09-30.

[22] G. Ahmdahl, "Validity of the single processor approach to achieving large scale computing capabilities". *AFIPS spring joint computer conference proceedings (30), 1967*: pp. 483-485.

[23] R. H. Carver and K-C. Tai, *Modern Multithreading – Implementing, Testing, and Debugging Multithreaded Java and C++/Pthreads/Win32 Programs.* New Jersey: John Wiley & Sons, Inc. 2006. ISBN: 0-471-72504-8.

[24] Sun Microsystems. "JSE 5.0 - Concurrency Utilities".
http://java.sun.com/j2se/1.5.0/docs/guide/concurrency/overview.html
Retrieved: 2009-10-12.

[25] B Goetz, "Let's Resync Concurrency Features in JDK™ 7". *Sun Microsystems*.
http://developers.sun.com/learning/javaoneonline/2008/pdf/TS-5515.pdf
Retrieved: 2009-10-12.

[26] A. Kaminsky. "Parallel Java Library".
Department of computer science, Rochester Institute of Technology.
http://www.cs.rit.edu/~ark/pj.shtml
Retrieved: 2009-10-12.

[27] Java Parallel Processing Framework. "JPPF Home".
http://www.jppf.org/index.php
Retrieved: 2009-10-12.

[28] EPCC, University of Edinburgh, "JOMP". http://www2.epcc.ed.ac.uk/computing/research_activities/jomp/index_1.html
Retrieved: 2009-10-21.

[29] R. Van Nieuwpoort, "Manta: Fast Parallel Java". http://www.cs.vu.nl/~rob/manta/index.html
Retrieved: 2009-10-12.

[30] Aart J.C. Bik and Dannis B. Gannon. "javab Manual (version 1.0BETA)". *Lindley Hall 215, Computer Science Department, Indiana University, Indiana, USA.*

[31] Indiana University, "High performance Java". http://www.extreme.indiana.edu/hpjava/
Retrieved: 209-10-22.

[32] B. Barney. "Introduction to parallel Computing". *Livermore Computing*. https://computing.llnl.gov/tutorials/parallel_comp/
Retrieved: 2009-10-22.

[33] K. Scott, "Computer Science: Rough Times Ahead". *Science, Vol. 299, No. 5607*, pp. 668-669.

[34] M. Kuulusa. "DSP Processor Core-Based Wireless System Design". *Ph. D thesis, Tempere University of Technology, Publications 296*, 152 pages.

[35] NVIDIA Corporation, "CUDA Zone – What is CUDA". http://www.nvidia.com/object/cuda_what_is.html
Retrieved: 2009-10-22.

[36] Khronos Group, "OpenCL", http://www.khronos.org/opencl/
Retrieved: 2009-10-22.

[37] Company for Advanced Supercomputing Solutions, "Hoopoe project - jCUDA". http://www.hoopoe-cloud.com/Solutions/jCUDA/Default.aspx
Retrieved: 2009-10-22.

[38] D. Madhava Rao, N. V. Thondugulam, R. Radhakrishnan and P. A. Wilsey, "Unsynchronized Parallel Discrete Event Simulation", *Proceedings of the 1998 Winter Simulation Conference,* 1998, pp. 1563-1570.

[39] A. Hind, "Parallel Simulation for Performance Modelling of Telecommunication Networks", *Teletraffic Symposium, 8th, IEEE Eigth UK, 1991,* pp. 9/1-9/6.

[40] P. Konas, "Where Object-Oriented Technology Meets Parallel Simulation", *Proceedings of the 29th Annual Hawaii International Conference on System Sciences*, 1996, pp. 360-363.

[41] N. Kalantery and S.C Winter. "High Performance Parallel Simulation of Telecommunication Networks", *High Performance Computing for Advanced Control*, IEEE Colloquium, 1994.

[42] A. Boukerche and S. K. Das, "A Dynamic Load Balancing Algorithm for Conservative Parallel Simulations", *Modeling, Analysis, and Simulation of Computer and telecommunication Systems, 1997, MASCOTS '97, Proceedings Fifth Internaltional Symposium,* pp. 32-37.

[43] P. Heidelberger and D. Nicol, "Building Parallel Simulations From Serial Simulators", *Modelling, Analysis, and Simulation of Computer and Telecommunication Systems, 1996, MASCOTS '96, Proceedings of the Fourth International Workshop, IEEE,* pp. 2-4.

[44] A. Boukerche and A. Fabbri, "Partitioning Parallel Simulation of Wireless Networks", *Simulation Conference Proceedings, 200, Winter, IEEE,* pp. 1449-14557, vol.2.

[45] L. Bononi, M. Bracuto, G. D'Angelo and L. Donatiello, "Exploring the Effects of Hyper-Threading on Parallel Simulation", *Simulation Conference Proceedings, 2000, Winter, IEEE,* pp. 1449-1457, vol. 2.

[46] L. Bajaj, R. Bagrodia, R. Meyer, "Case Study: Parallelizing a Sequential Simulation Model", *Parallel and Distributed Simulation, 1999. Proceedings Thirteenth Workshop, IEEE.*

[47] EJ-Technologies. “JProfiler – Java Profiling”. http://www.ej-technologies.com/products/jprofiler/overview.html Retrieved: 2009-10-25.

[48] W. Kahan and J. D. Darcy. “How Java’s Floating-Point Hurts Everyone Everywhere”. *1 March, ACM 1998 Workshop on Java for High-Performance Network Computing, Standford University.* Available online: http://www.eecs.berkeley.edu/~wkahan/JAVAhurt.pdf

[49] Sun Microsystems. “CRC32 (Java 2 Platform SE v1.4.2” http://java.sun.com/j2se/1.4.2/docs/api/java/util/zip/CRC32.html Retrieved: 2010-11-30.

[50] Internet Engineering Task Force (IETF). “RFC 1952”. http://www.ietf.org/rfc/rfc1952.txt Retrieved: 2010-02-01.

Appendix A: System specifications

System specifications for the systems used in experiments. Three different aliases have been used in order to referrer to these systems.

System Alias	CPU	CPU Cores	RAM Memory (MB)	OS
A	Intel® Core™2 6600 @ 2.40 GHz, 4096 KB cache.	2	2,048	Linux 2.6.16 64-bit (x86_64)
B	Intel® Core™2 Extreme X9650 @ 3.00 GHz, 6144 KB cache.	4	3,567	Windows Vista™ Enterprise 32-bit.
C	2 x Intel® Xeon™ E5440 @ 2.83 GHz, 6144 KB cache.	2 x 4	64,975	Linux 2.6.16 64-bit (x86_64)

Table 3: System specifications for the systems used for performance measurements.

Appendix B: Questionnaire for evaluation of readability and usability of parallel constructs

Please formulate your answers short and consistent.

1. **Do you have any prior experience of multithreading constructs and parallel programming from Java and/or any other object oriented language? Specify language and relevant experience.**

2. The abstract classes *UnorderedTask* and *OrderedTask* enable asynchronous execution of arbitrary code by implementing standard Java interfaces (*Runnable* and *Callable<Runnable>*) in specialized anonymous classes as illustrated by the code example in Figure 33.

```
new UnorderedTask(this) {
   public void run() {
      /* process on separate thread */
      asynchronousOperationA();
   }
};
new OrderedTask(this) {
   public Runnable call() {
      /* process on separate thread */
      final Result result = asynchronousOperationB();
      return new Runnable() {
         public void run() {
            process(result);  /* process on main thread */
         }
      };
   }
};
/* wait for tasks to complete */
getSimulator().getTaskManager().completeTasks();
```

Figure 33: Code example 1 illustrating anonymous task class specialization.

UnorderedTask is used to execute a code segment asynchronously where order of execution doesn't matter. *OrderedTask* is used to execute a code segment asynchronously and process the results of the asynchronous computation sequentially in the order that the tasks where created.

When a task is created it submits itself for execution on a separate thread. A task manager coordinates execution. By calling the *completeTasks()* method of the task manager, the method will not return until all tasks have finished their asynchronous execution and all *Runnable* objects returned by instances of OrderedTask have been executed in the sequence that they we're created.

a) Do you understand the purpose and difference between the two classes/constructs *UnorderedTask* and *OrderedTask*? If not explain shortly what confuses you.

b) Do you see any obvious advantages with the constructs in Figure 33 compared to parallel constructs you've seen examples of or used previously?

c) Do you see any obvious disadvantages with the constructs in figure Figure 33 compared to parallel constructs you've seen examples of or used previously?

3. Would you be comfortable using constructs like the ones described in Figure 33 in your daily implementation work in order to increase application performance? If not, why?

4. Can you tell what the console output from the program in Figure 34 would be? If not, specify why.

```
...
new UnorderedTask(this) {
   public void run() {
      System.out.print("async operation A");
   }
};
new UnorderedTask(this) {
   public void run() {
      System.out.print("async operation B");
   }
};
System.out.print("sequential operation C");
getSimulator().getTaskManager().completeTasks();
...
```

Figure 34: Fictive code example of *UnorderedTask* usage.

5. Can you tell what the console output from the program in Figure 35 would be? If not, specify why.

```
...
new OrderedTask(this) {
   public Runnable call() {
      final Result r = async_operation(some_data);
      return new Runnable() {
         public void run() {
            process(r);
            System.out.println("seq part A");
         }
      };
   }
};
new OrderedTask(this) {
   public Runnable call() {
      final Result r = async_operation(some_data);
      return new Runnable() {
         public void run() {
            process(r);
            System.out.println("seq part B");
         }
      };
   }
};
System.out.println("seq part C");
getSimulator().getTaskManager().completeTasks();
...
```

Figure 35: Fictive example of *OrderedTask* usage.

Appendix C: UML class diagram for task-oriented framework

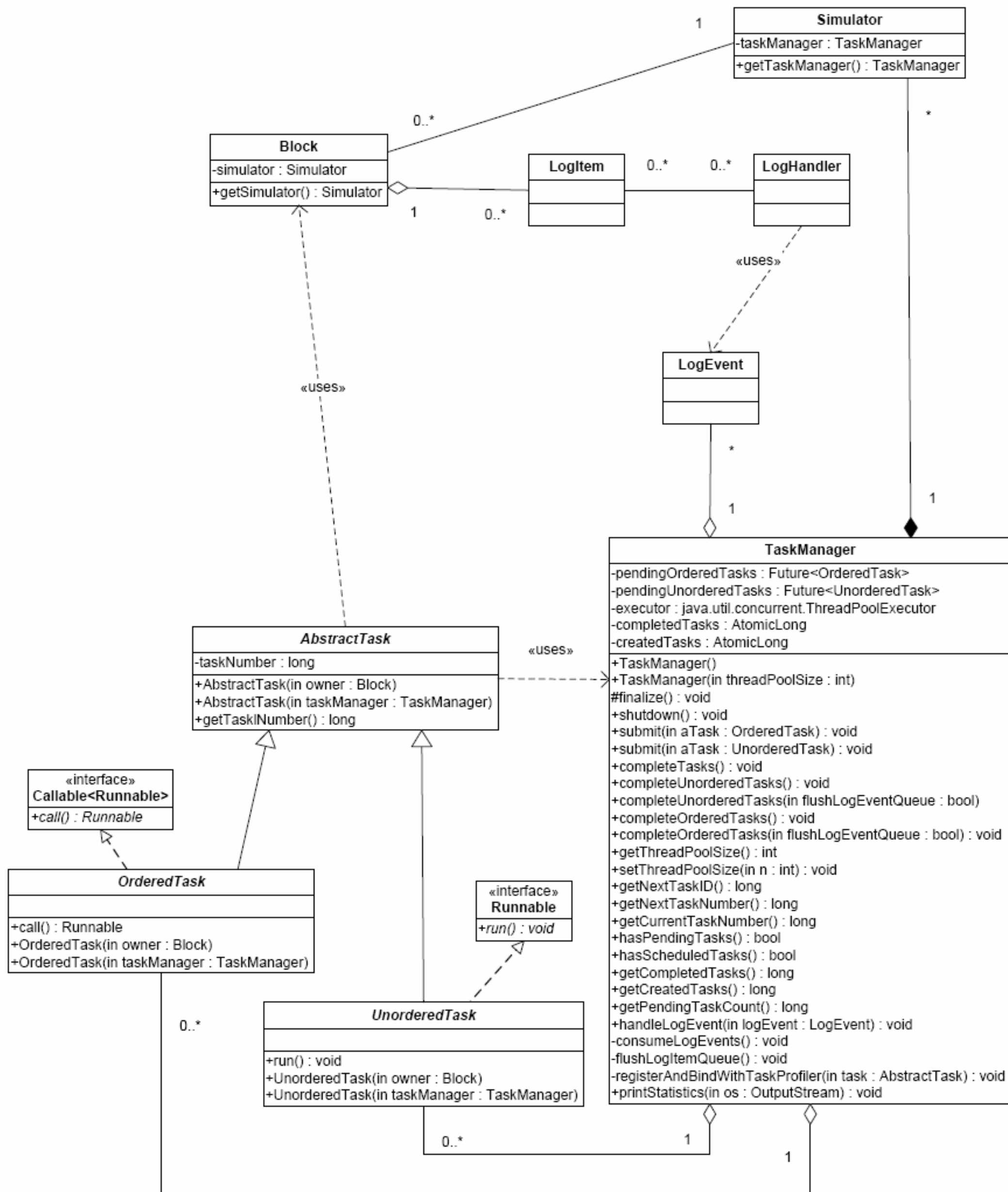


Figure 36: UML class diagram of task-oriented concurrency framework.

Appendix D: Summary of questionnaire results

A summary of the received answers to the questionnaire in Appendix B is presented here. A total of X people answer the questionnaire. Y percent of these are constituted by research engineers at Ericsson and Z percent are people that are working with computer engineering outside of Ericsson.

Response 1 of 6

1. No.

2. a) Yes

b) I have not seen any other constructs previously.

c) I have not seen any other constructs previously

3. Yes

4. No, I cannot tell the order. The “async operation A” and “async operation B” are output from asynchronously executed code.

5. Yes, the order will be “seq part A”, “seq part B” and “seq part C”.

Response 2 of 6

1. Yes I have experience from multithreading and parallel programming in both Java and C# .NET. I’ve gained this experience when working with development of systems for business administration and multithreaded generation of reports.

2. a) The difference and the purpose of the two abstract classes UnorderedTask and OrderedTask is very clear from their names.

b) To be honest, I’m not really fond of anonymous classes, but in this case I find the usage and creation of new jobs relatively simple and straight forward.

c) I cannot see any obvious drawbacks with these constructs.

3. I can see myself using this framework for multithreading in my daily work.

4. What is certain is that "sequential operation C" will be printed first since *completeTasks* will be called after that print statement. Which one of the other statements that will be printed next is impossible to say since they aren't executed in any sequential order.

5. In this case it's clear in which order the statements will execute. This is clear since there is two OrderedTask that are added and they will execute in the order they we're created. The program will print the following:

seq part C

seq part A

seq part B

Response 3 of 6

1. Sometimes I use threads when I develop Windows applications. One example is when data should be retrieved from a database at the same time as a GUI representing the data should be drawn to the screen. Using threads it's possible to show the GUI and display a progress bar to the user to display something like "Collecting data from database…" or similar to avoid letting the user think that the application has hung.

2. a) Yes

b) If the application has an operation Thread1:HeavyCalculation() and the user starts it by mistake and want to cancel. Then Thread2:StopAllOperations() might be called to stop Thread1 without waiting for the operation to complete. Am I thinking right?

c) No

3. Yes, it will prevent the program from crashing when a user uses my GUI in a way I didn't intend for. The user will also get a smoother experience and experience that the application is faster.

4. C will be executed first, then it's uncertain whether A or B will be executed next. You can guess that A will be executed before B if A is lighter.

5. (No answer)

Response 4 of 6

1. C, Java and pthreads.

2. a) Yes

b) Fits well with the simulator with asynchronous processes and synchronous processing of results.

c) No

3. Yes

4. No, the print order is undefined.

5. Yes: C, A, B.

Response 5 of 6

1. No

2. a) Unclear why OrderedTask contains an asynchronous part.

b) –

c) –

3. Sure

4. Since it's unordered I don't know if A, B or C is printed out first.

5. A is printed before B, but I'm not sure if C is printed first or last.

Response 6 of 6

1. Yes, on a basic level when I need multiple amounts of calls to be able to execute irrespective of the other calls in a Java environment.

2. a) I think its very simple. Honestly.

b) If the order of the executions is important I think the OrderedTask looks easy to use. UnOrderedTask looks like the kind of threading I'm used to.

c) Not as I can figure out now.

3. Yes.

4. I have experienced something like:

async operation C
async operation A
async operation B

Reason? The thread that is creating the UnOrderedTask is in turn....could vary though.

5.

seq part C
seq part A
seq part B

I'm unsure though. A will always be done before B, but I don't know if C will come first....