Linköping Studies in Science and Technology
Dissertations No. 1732

Distributed System Simulation Methods

For Model-Based Product Development

Robert Braun

LINKÖPING UNIVERSITY

Division of Fluid and Mechatronic Systems
Department of Management and Engineering
Linköping University, SE-581 83 Linköping, Sweden

Linköping 2015

Copyright © Robert Braun, 2015

Distributed System Simulation Methods
For Model-Based Product Development

ISBN 978-91-7685-875-2

ISSN 0345-7524

Distributed by:

Division of Fluid and Mechatronic Systems
Department of Management and Engineering
Linköping University
SE-581 83 Linköping, Sweden

Printed in Sweden by LiU-Tryck, Linköping 2015.

To Emma

“Computers are useless. They can only
give you answers.

Pablo Picasso

Abstract

With the current trend towards multi-core processors and computer grids, parallelism in system simulation is becoming inevitable. Performing multiple tasks concurrently can save both time and money. However, this also puts higher demands on simulation software. This thesis investigates how simulation-based product development can benefit from distributed simulation.

One parallelization method is to cut apart models for simulation with distributed solvers by using time delays. The transmission line element method (TLM) is used, which eliminates numerical errors by introducing physically motivated time delays. Results show that linear speed-ups can be obtained for large models with balanced workloads. Different task scheduling techniques are tested. It is found that a fork-join algorithm with implicit synchronization performs best for models with a large total workload. On the other hand, a task-pool implementation with lock-based synchronization works better with smaller workloads.

The distributed solver method virtually equals co-simulation between different simulation tools. Co-simulation also induces time delays at the interface variables, which makes TLM a useful method. An emerging standard for coupling of simulation tools is the Functional Mock-up Interface (FMI). Experiments show that FMI works well together with TLM, even when connecting different simulation domains such as system simulation and multi-body mechanics. Combining FMI with TLM enables a framework for tool-independent co-simulation with distributed solvers and time steps.

When introducing distributed solvers, it can be suitable to maintain the same structure that was used in the modelling process. A popular object-oriented equation-based modelling language is Modelica. Instead of flattening the model for simulation with a centralized solver, TLM makes it possible to use different solvers for different objects in the model. Two approaches are investigated. First, Modelica support is implemented in the Hopsan TLM tool with a code generator. Second, Modelica models are imported to Hopsan through the FMI interface. Both methods are feasible. While the first requires less overhead

work, the second makes it possible to mix models from different Modelica tools and to use all the features of each individual tool. Depending on the number of TLM elements and their locations, model structures with different granularities can be achieved.

Parallelism can be introduced at different levels. The distributed solver approach results in parallelism at the model-level. It is also possible to parallelize equation solvers, or to run several simulations in parallel. The latter is especially useful for simulation-based design optimization. An algorithm for profile-based scheduling of parallel optimization jobs is proposed that takes parallelism at both model-level and optimization level into account. Parallel optimizations can then be executed on either homogeneous or heterogeneous computer grids. This makes it possible to utilize the large amount of unused computer power that exists in many organizations.

Distributed optimization requires efficient parallel optimization algorithms. While population-based methods are inherently parallel, they also suffer from slow convergence rates. For this reason, possibilities for parallelizing the Complex-RF algorithm, which is originally of sequential nature, are also investigated.

The thesis concludes that distributed simulations constitute a powerful improvement to simulation-based product development. Combining parallelism with co-simulation facilitates cooperative development and increases reusability and flexibility of simulation models. A natural continuation is to develop a general integrated co-simulation framework with support for FMI, TLM, distributed solvers, equation-based models and job scheduling for parallel optimization.

Populärvetenskaplig sammanfattning

När man utvecklar en ny produkt vill man testa att den fungerar. Att bygga fysiska prototyper kan vara dyrt, kostsamt och i vissa fall farligt. Med hjälp av datorsimulering kan man istället testa produkten i en säker datormiljö. På så vis kan man spara mycket tid och pengar. Nyttan med simulering begränsas emellertid ofta av långa simuleringstider. Tidigare har man kunnat lita på att datorernas processorer, och därmed även datorsimuleringar, blir snabbare med åren. På senare tid har dock utvecklingen istället gått mot processorer som har flera kärnor och därigenom kan utföra flera beräkningar samtidigt. För att kunna utnyttja detta krävs att datorprogram utformas så att flera arbetsuppgifter kan utföras parallellt.

En simuleringsmodell består vanligtvis av ett ekvationssystem som beskriver produktens fysikaliska egenskaper. En lösningsalgoritm används för att beräkna utvariabler utifrån en viss uppsättning invariabler. För att utföra detta parallellt krävs att ekvationssystemet delas upp i mindre delsystem, där en individuell lösningsalgoritm används för vart och ett av dessa. Att dela upp ett ekvationssystem medför dock att vissa variabler får en tidsfördröjning som kan orsaka numeriska fel. Som tur är har så gott som allting i den verkliga världen också tidsfördröjningar. Genom att utnyttja dessa i modelleringen kan modeller delas upp och beräknas samtidigt på flera processorkärnor utan att några numeriska fel uppstår. Experiment har visat att detta kan spara mycket tid för stora modeller.

Fysikaliska tidsfördröjningar kan även utnyttjas för att koppla ihop olika simuleringsprogram med varandra. Detta är användbart eftersom olika program är lämpliga för olika saker. Vissa program kan också vara populära i olika organisationer, branscher eller forskningsområden. Att kunna koppla samman modeller oberoende av var de skapats underlättar samarbete och gör det möjligt att modellera varje del av systemet i det bäst lämpade programmet.

I avhandlingen undersöks även hur modeller kan delas upp när det ekvationsbaserade språket Modelica används. Sådana språk gör det möjligt att skriva ekvationer som man är van vid, istället för att skriva om dem för att anpassa dem till ett visst programmeringsspråk. Eftersom språket är objektorienterat så passar det väl ihop med distribuerade lösare.

Utöver att dela upp en modell i parallella delmodeller går det även att köra flera simuleringar samtidigt med samma modell. Detta är exempelvis vanligt vid simuleringsbaserad optimering, där man testat olika invariabler för att få ett så bra resultat som möjligt. Om både optimeringen och varje enskild simulering kan köras parallellt, måste man ta hänsyn till detta när man schemalägger körningen för en viss hårdvara. En algoritm för att schemalägga optimeringar mot nätverk av datorer med flerkärniga processorer har därför utvecklats. Möjligheten att göra om en sekventiell optimeringsalgoritm så att den kan köras parallellt har också undersökts.

Sammanfattningsvis kan sägas att det finns stora vinster med distribuerade simuleringar i utveckling av nya produkter. En naturlig fortsättning är att utveckla ett generellt ramverk för att koppla samman och köra delmodeller från olika program med distribuerade lösningsalgoritmer och effektiv schemaläggning.

Acknowledgements

There are many people to whom I would like to express my gratitude for supporting my doctoral studies. First of all I would like to thank my supervisor, professor Petter Krus, for giving me the opportunity to become a part of the division and for sharing his invaluable knowledge of the transmission line element method.

I would also like to thank my closest colleague and fellow Hopsan developer Peter Nordin. This project would not have been possible without your devoted support and excellent programming skills.

My sincere thanks also go to our industrial partners at Atlas Copco. Our cooperation has provided many fruitful ideas and well-needed feedback. You also made it possible for me to test my inventions in real-world applications.

Thanks also to all our Hopsan users in industry, academia and not least our undergraduate students, who struggle with Hopsan come rain come shine and discover bugs in the most inventive ways.

I would also like to express my appreciation to all my former and present colleagues at the divisions of Fluid and Mechatronic Systems and Machine Design. Thanks to our friendly and respectful working climate, it always feels great to go to work.

Last but not least I would like to thank my beloved Emma for her love and support, and my parents for always supporting my academic career. Finally, I also want to express my gratitude to our cat for taking a nap on my keyboard every time he decides I need a coffee break.

This work has been supported by the Swedish Foundation for Strategic Research (SSF) and the ProViking research school as part of the project High-Speed Simulation for Product Design and Operation (HiPO).

Robert Braun
Linköping, October 2015

Abbreviations

CPU	central processing unit
DE	differential evolution
DOP	degree of parallelism
EA	evolutionary algorithms
EOO	equation-based object-oriented language
FJS	fork-join scheduling
FMI	Functional Mock-up Interface
FMU	Functional Mock-up Unit
GA	genetic algorithms
OMC	OpenModelica Compiler
PSO	particle swarm optimization
PSS	pre-simulation scheduling
TBB	Intel® Threading Building Blocks
TLM	the transmission line element method
TPS	task pool scheduling
TSS	task-stealing scheduling

Notation

A_L	Four-pole element see eq. (2.15)	[-]
B_L	Four-pole element see eq. (2.15)	[-]
C	Lumped capacitance see eq. (2.4)	[m ³ /Pa]
C_d	Distributed capacitance see eq. (2.1)	[m ² /Pa]
C_L	Four-pole element see eq. (2.15)	[-]
D_L	Four-pole element see eq. (2.15)	[-]
EF	Parallelization efficiency see eq. (7.3)	[-]
G	Conductance see eq. (2.1)	[m ³ /sPa]
H_x	Entropy see eq. (10.1)	[bits]
L	Lumped capacitance <i>see eq.</i> (2.5)	[s ² Pa/m ³]
L_d	Distributed capacitance see eq. (2.1)	[s ² Pa/m ⁴]
N	Frequency-dependent friction factor see eq. (2.15)	[-]
N_m	Average number of iterations for convergence see eq. (10.2)	[-]
P_{opt}	Probability of finding global optima see eq. (10.2)	[-]
R	Resistance see eq. (2.1)	[sPa/m ³]
SL	Parallelization slow-down see eq. (7.4)	[-]
SU_m	Model-level speed-up see eq. (10.5)	[-]
SU_{abs}	Absolute speed-up see eq. (7.1)	[-]
SU_{alg}	Algorithm-level speed-up see eq. (10.4)	[-]
SU_{rel}	Relative speed-up see eq. (7.2)	[-]
T	Time delay see eq. (2.3)	[s]

V	Fluid volume see eq. (2.8)	$[\text{m}^3]$
Z_C	Characteristic impedance see eq. (2.1)	$[\text{Nsm}^5]$
Δx	Parameter uncertainty see eq. (10.1)	$[-]$
$\phi^{(2)}$	Entropy rate based performance index see eq. (10.2)	$[-]$
ε_x	Tolerance for convergence see eq. (10.2)	$[-]$
a	Speed of sound see eq. (2.9)	$[\text{m/s}]$
c	Wave variable see eq. (2.21)	$[\text{Pa}]$
$n_{eval,e}$	Number of equivalent evaluations with parallelism see eq. (10.3)	$[\text{s}]$
n_{eval}	Number of evaluations see eq. (10.4)	$[-]$
p	Fluid pressure see eq. (2.9)	$[\text{Pa}]$
p_a	Degree of parallelism for algorithm see eq. (10.4)	$[-]$
p_m	Degree of parallelism for each model see eq. (10.5)	$[-]$
q	Fluid flow rate see eq. (2.9)	$[\text{m}^3/\text{s}]$
t_{comm}	Communication time see eq. (10.3)	$[\text{s}]$
t_m	Optimization model simulation time see eq. (10.3)	$[\text{s}]$
t_{opt}	Optimization execution time see eq. (10.3)	$[\text{s}]$
x_R	Feasible design region see eq. (10.1)	$[-]$
c	Wave variable see eq. (2.22)	$[\text{Pa}]$
n	Problem size see eq. (7.1)	$[-]$
t_p	Execution time with parallel implementation see eq. (7.1)	$[\text{s}]$
t_s	Execution time with sequential implementation see eq. (7.1)	$[\text{s}]$
β_e	Fluid bulk modulus see eq. (2.8)	$[\text{Pa}]$

Papers

This thesis is based on the following eight appended papers, which will be referred to by their Roman numerals. The first author has conducted the research in papers [I], [II], [III], [V], [VI] and [VII]. In paper [IV] the first author has been responsible for the co-simulation interface. In paper [VIII], the first author is the main author while the second author is responsible for optimization algorithms and the load-balancing algorithm. All papers are printed in their original state with the exception of minor errata and changes in text and figure layout in order to maintain consistency throughout the thesis.

- [I] Robert Braun and Petter Krus. “Multi-threaded distributed system simulations using the transmission line element method”. In: *SIMULATION* (2015). Submitted.
- [II] Robert Braun, Peter Nordin, and Petter Krus. “Improved Scheduling Techniques for Parallel Distributed-Solver System Simulation”. In: *SIMULATION* (2015). Submitted.
- [III] Robert Braun and Petter Krus. “Multi-Threaded Real-Time Simulations of Fluid Power Systems Using Transmission Line Elements”. In: *Proceedings of the 8th International Fluid Power Conference (IFK)*. Dresden, Germany, Mar. 2012.
- [IV] Robert Braun, Liselott Ericsson, and Petter Krus. “Full Vehicle Simulation of Forwarder with Semi Active Suspension using Co-Simulation”. In: *ASME/BATH 2015 Symposium on Fluid Power and Motion Control*. Chicago, USA, Oct. 2015.
- [V] Robert Braun and Petter Krus. “An Explicit Method for Decoupled Distributed Solvers in an Equation-Based Modelling Language”. In: *Proceedings of the 6th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. Berlin, Germany, Oct. 2014, pp. 57–64.

- [VI] Robert Braun and Petter Krus. “Tool-Independent Distributed Simulations Using Transmission Line Elements And The Functional Mock-up Interface”. In: Bergen, Norway, Oct. 2013.
- [VII] Robert Braun and Petter Krus. “Parallel Implementations of the Complex-RF Algorithm”. In: *Engineering Optimization* (2015). Submitted.
- [VIII] Peter Nordin, Robert Braun, and Petter Krus. “Job-Scheduling of Distributed Simulation-Based Optimization with Support for Multi-Level Parallelism”. In: *The 56th Conference on Simulation and Modelling (SIMS 56)*. Linköping, Sweden, Oct. 2015.

The following four complementary papers are not part of the thesis, but constitute an important part of the background.

- [IX] Martin Sjölund, Robert Braun, Peter Fritzson, and Petter Krus. “Towards Efficient Distributed Simulation in Modelica using Transmission Line Modeling”. In: *3rd International Workshop on Equation-Based Object-Oriented Languages and Tools*. Oslo, Norway, Oct. 2010.
- [X] Mikael Axin, Robert Braun, Alessandro Dell’Amico, Björn Eriksson, Peter Nordin, Karl Pettersson, Ingo Staack, and Petter Krus. “Next Generation Simulation Software Using Transmission Line Elements”. In: *Fluid Power and Motion Control*. Bath, England, Oct. 2010.
- [XI] Robert Braun, Peter Nordin, Björn Eriksson, and Petter Krus. “High Performance System Simulation Using Multiple Processor Cores”. In: *The Twelfth Scandinavian International Conference On Fluid Power*. Tampere, Finland, May 2011.
- [XII] Robert Braun and Petter Krus. “Towards A Parallel Distributed Equation-Based Simulation Environment”. In: *53rd SIMS Conference on Simulation and Modelling*. Bergen, Norway, Oct. 2012.

Contents

I	Introduction	1
1	Introduction	3
1.1	Motivation and Needs	4
1.2	Research Questions	5
1.3	Delimitations	5
1.4	Contribution	6
II	Frame of Reference	7
2	Parallel Simulation	9
2.1	Parallelism across the Solver	10
2.2	Parallelism across the System	10
2.2.1	Background	10
2.2.2	Transmission Line Equations	11
2.2.3	Derivation	13
2.2.4	Parasitic Inductance	15
2.2.5	Parallel Simulations	19
2.2.6	Related Methods	19
2.3	Applications	20
2.3.1	Sensitivity Analysis	20
3	Hopsan Simulation Tool	21

3.1	Simulation Core	22
3.2	Graphical Interface	23
3.2.1	Scripting	23
3.3	HopsanGenerator Module	25
4	Simulation Tool Connectivity	27
4.1	Functional Mock-Up Interface	27
4.2	Related Work	30
5	Equation-Based Modelling	31
5.1	Modelica	31
5.2	Parallel Simulation in Modelica	33
6	Optimization Algorithms	35
6.1	Direct-Search Methods	35
6.1.1	Pattern Search	36
6.1.2	Nelder-Mead Method	37
6.1.3	Complex-RF Algorithm	38
6.2	Population-based Methods	39
6.2.1	Particle Swarm Optimization	39
6.2.2	Evolutionary Methods	41
III	Contributions	43
7	Multi-threaded Simulations	45
7.1	Implementation	45
7.2	Profiling	47
7.3	Task Scheduling	48
7.3.1	Pre-Simulation Scheduling	49
7.3.2	Task-Stealing Scheduling	50
7.3.3	Task Pool Scheduling	50
7.3.4	Fork-Join Scheduling	51
7.4	Performance Evaluation	52

8	Multi-Domain Co-Simulation	55
8.1	Real-time Simulation through Library Export	56
8.1.1	Implementation	56
8.1.2	Results and Analysis	58
8.2	Simulation of Active Suspension in Forwarder	59
8.2.1	Interface	60
8.2.2	Synchronization	60
8.2.3	Evaluation	61
9	Distributed Equation Solvers	63
9.1	A Code Generation Approach	65
9.1.1	Code Generation	65
9.1.2	Modelica Solver	66
9.1.3	SymHop Symbolic Algebra Library	67
9.2	A Co-Simulation Approach	68
9.3	Modelling Techniques	71
10	Parallel Distributed Optimization	73
10.1	Parallel Implementations of Complex-RF	74
10.1.1	Task-prediction	74
10.1.2	Multi-retraction	75
10.1.3	Multi-distance	76
10.1.4	Multi-direction	77
10.1.5	Parallel Performance Evaluation	78
10.2	Work Scheduling for Distributed Optimization	81
IV	Discussion and Conclusions	85
11	Discussion	87
12	Conclusions	91
13	Outlook	93

14 Review of Papers	95
A Pre-Simulation Scheduler Code	97
B Task Stealing Scheduler Code	99
C Task Pool Scheduler Code	103
D Fork-Join Scheduler Code	105
E Veristand Interface Source Code	107

Appended papers

I Multi-Threaded Distributed System Simulations using the Transmission Line Element Method	109
II Improved Scheduling Techniques for Parallel Distributed-Solver System Simulation	127
III Multi-Threaded Real-Time Simulations of Fluid Power Systems using Transmission Line Elements	145
IV Full Vehicle Simulation of Forwarder with Semi Active Suspension using Co-simulation	157
V An Explicit Method for Decoupled Distributed Solvers in an Equation-Based Modelling Language	175
VI Tool-Independent Distributed Simulations using Transmission Line Elements and the Functional Mock-up Interface	195
VII Parallel Implementations of the Complex-RF Algorithm	207
VIII Job-Scheduling of Distributed Simulation-Based Optimization with Support for Multi-Level Parallelism	229

Part I

Introduction

1

Introduction

With simulation-based product development, concepts and designs can be examined in computer models. This reduces the need for expensive and time-consuming physical prototyping. It also enables experiments that would be dangerous or infeasible in the physical world. By discovering design flaws at an early stage in the development process, costly design iterations can be avoided. This can improve quality and reduce time-to-market for new products.

In the computer industry there is currently a strong trend towards parallel computing. This makes it possible for a computer to perform several tasks simultaneously, which can greatly reduce time requirements for heavy computations. However, it also makes software development more difficult, since the code must be adapted for parallel execution. This is further complicated by the fact that computer hardware can be parallel on different levels. It is, for example, possible to use parallel computers, parallel processors, multi-core processors or graphics processors.

The modelling languages used to create simulation models are of great importance. Languages that feel intuitive for the user can speed up the development process, increase understanding of the model, facilitate knowledge sharing and support design decisions. Equation-based object-oriented languages, such as Modelica, are used on a large scale in both industry and academia. Supporting or being compatible with such languages is critical for the acceptance of new simulation paradigms.

Simulation-based optimization is an area especially suitable for parallel computing. By finding the optimal solution in a computer model, good designs and parametrizations can be generated without the need for physical prototypes. While this is a powerful development tool, it also consumes substantial computer resources. A related area is sensitivity analysis, where the relationship between certain input and output variables is studied. Testing multiple designs

simultaneously can increase the chances of finding an optimal solution at an early stage.

Adapting computer simulations to parallel computing is not trivial. Simulation models are usually tightly coupled and must be solved sequentially. Cutting them apart will introduce a time delay, which affects the results. One solution is to exploit physical time delays that exist in the real world. This is the concept of the transmission line element method, which has consequently remained a popular simulation technique for several decades.

1.1 Motivation and Needs

In computer simulation there is a strong connection between computational speed and practical usefulness. The most common purpose of simulation-based product development is to save time and money by eliminating the need for costly and time-consuming physical prototyping. Increasing performance is therefore of high priority. Historically it has been possible to rely on continuously increasing processor speeds. The same simulation code would run faster and faster for each new processor generation. Recently, however, development has turned towards processors with multiple processing units, or cores, while the speed increase of each individual core has levelled off. This puts high demands on simulation tool developers to adapt their code for parallel hardware. Previous attempts at parallel simulations have often required a great deal of manual work from the user. A user-friendly and highly automated method for parallelism is therefore desirable.

Another obstacle that limits the usefulness of simulation is inadequate tool interoperability. Most simulation tools are specialized in certain disciplines or domains. Different tools are also favored by different organizations or technical fields. As a consequence, efficient coupling between simulation tools is becoming increasingly important. This requires standardized interfaces and methods in order to ensure compatibility.

A trend in simulation is the use of equation-based languages, which has increased considerably in popularity in recent decades. For this reason, it is important that such languages can also make use of parallel hardware. Equation systems must be partitioned in a way that maximizes performance, modularity, usability and re-usability. Meanwhile, user overhead should be kept to a minimum by avoiding new language constructs.

A common use for computer simulation is design optimization. This requires the simulation model to be executed a large number of times, and is therefore especially time-consuming. For this reason, a lot of work has been done on parallel optimization algorithms. Nevertheless, there is still a trade-off between the fast convergence rates of direct-search methods and the parallelizability of population-based methods. In addition, efficiently combining parallel algorithms with parallel simulation models largely remains an open question.

1.2 Research Questions

The original research question is formulated as follows:

RQ1: How can simulation-based product development benefit from distributed-solver simulations, with respect to usability, re-usability, development cost and time-to-market?

During the work, additional and more specific questions have emerged:

RQ2: How can parallel simulations with distributed solvers using the transmission line element method be implemented with minimal overhead work for the user?

RQ3: In which ways can the transmission line element method be used for tool coupling, and what problems can arise?

RQ4: How can equation-based languages benefit the most from distributed equation solvers?

RQ5: How can optimization time be reduced by parallel direct-search algorithms?

RQ6: How can optimal speed-up be achieved for simulation-based optimization if both the optimization algorithm and the model itself support parallel execution?

1.3 Delimitations

Only continuous-time simulation has been considered. The reason for this is that the majority of the work is based on TLM, which induces time delays that naturally limit the propagation of events.

Only parallelism at model-level and above has been investigated. Lower level parallelism such as parallel solvers or explicit language constructs are covered by other publications [1][2].

All parallel implementations of system simulation use shared memory parallelism. Message passing interfaces were disregarded at an early stage, since the rate of data exchange was expected to be high.

Also, only multi-core processors and network computing are targeted. Graphical processing units have not been included, since the applications at hand require task parallelism rather than data parallelism. All experiments have been performed on personal computers, since this is what the intended users will have.

1.4 Contribution

The main contributions of this thesis are as follows:

- An automatic algorithm for sorting, scheduling and synchronization of distributed solvers has been implemented in the Hopsan simulation environment. Speed-up results have been measured and analyzed.
- The parallel efficiency for different scheduling techniques in relation to model size has been investigated.
- Translation of Functional Mock-up Units to TLM-compatible code has been implemented in Hopsan. This makes it possible to use Hopsan as a co-simulation host for FMI with distributed solvers.
- A code generator for a subset of the Modelica language that generates TLM-compatible code has been implemented in Hopsan.
- A survey of modelling techniques for decoupling simulation models with TLM has been performed. Example models demonstrate the approaches.
- Several parallelization strategies for the Complex-RF optimization algorithm have been implemented and evaluated in Hopsan.
- An algorithm for scheduling of multi-level parallel optimization jobs in computer networks has been developed. A decent trade-off between parallelism at model-level and algorithm-level is computed based on the target hardware.

Part II

Frame of Reference

2

Parallel Simulation

Reducing the execution time of computer simulation can save both time and money. By introducing simulation at an early stage of the design cycle, the number of costly design iterations can be reduced. High-speed simulation is especially important for demanding applications such as real-time simulation, design optimization and batch simulations.

According to Moore's law, the number of transistors that can inexpensively be placed in integrated circuits will double approximately every second year [3]. Previously, this law could also be used to predict the increase in processor speeds. While the law still appears to hold, however, the relationship between speed and number of transistors has declined in recent years. This is a consequence of the so-called *power wall* [4]. The frequency of a processor is proportional to the voltage, and thereby to the square of the heat dissipation. At a certain point, the cost of cooling the processor will exceed the benefits of the higher speed. For this reason, processor manufacturers have turned to producing processors with several processing units, so called multi-core processors. While each individual core may actually be slower than those in single-core processors, the overall speed of all cores working together can still be faster.

This development puts large demands on software developers to write parallel code. A continuous increase in performance in simulation tools is not possible without exploiting multi-core technology to its fullest. A typical simulation model consists of a system of equations, where state variables are computed by a solver. This can be parallelized either across the system, across the solver or across time [5].

2.1 Parallelism across the Solver

A solver can be parallelized by distributing the computational effort of multiple integration steps over several processing units. A common method is to use

block-parallel corrector-predictor methods as explained in [6]. In [5], a wave front method for a predictor-corrector scheme and a diagonal implicit Runge-Kutta method are presented. If the model topology is known, it can be utilized for parallel matrix inverting [7]. The PVODE solver, a parallel version of the CVODE solver, has support for solving independent groups of an equation system in parallel [8].

2.2 Parallelism across the System

Solving different parts of a simulation model on parallel processing units requires the underlying equation system to be decoupled into independent groups. Partial differential equations (PDE) often exhibit data independence and can be integrated in parallel. Systems of ordinary differential equations (ODEs) and differential algebraic equations (DAEs), however, are typically coupled and require sequential solvers. Decoupling such systems introduces a time delay of one time step at the interface section. If not handled properly, this delay will affect numerical stability and disrupt simulation results. There are two ways of dealing with this. The first approach is to monitor the magnitude of the numerical error and, if necessary, apply methods to minimize this error. A common method is to reduce the step size or change the integration method until the error becomes acceptably small. The numerical error can, however, be avoided completely if the time step is modelled with a physical motivation. This is the fundamental concept of the transmission line element method (TLM). Essentially, this means that separations in space can be modelled as separations in time. All physical elements have a limited wave propagation speed, which induces a time delay. The magnitude of such delays results from the characteristic impedance of the element. By introducing such impedances into the model, correct wave propagation can be maintained. In this way, no numerical errors will be introduced. Thanks to the time delays, the equation system can be separated into smaller decoupled sub-systems. These systems can then use their own *distributed solvers*. This kind of physically motivated decoupling thus enables parallel simulations.

Related techniques include automatic block diagonalization [9] and dynamic decoupling [10]. The TLM method is unique in that it is physically motivated and eliminates all numerical errors.

2.2.1 Background

TLM originates from bilateral delay line models as presented by Auslander [11] and from transmission line modelling, which was used by Johns and O'Brien [12] for simulation of electrical networks and subsequently by Boucher and Kitsios [13][14] for simulating fluid systems. It is also related to the method of characteristics, which was later used in the Hytran software [15]. Transmission line modelling was also used by Partridge et al. [16] to simulate mechanical systems. Non-linear TLM models for power electronic circuits were developed

by Hui and Christopoulos [17][18]. In 1990, TLM was adopted by Krus et al. for distributed-solver simulations of fluid power systems in the Hopsan simulation tool [19]. It was shown that TLM models of hydraulic pipes could be simulated much faster than conventional pipe models with a maintained level of detail [20]. TLM has also been successfully used for multi-body simulations by Krus [21] and Fritzson et al. [22]. A method for implementing distributed solvers in the equation-based modelling language Modelica has also been presented by Johansson and Krus [23].

Simulating TLM models with automatic step size control has been shown to be feasible. Pulko et al. [24] have presented two methods for thermal diffusion models based on potential difference and potential drops. Another attempt using minimization of parasitic inductances was made by Jansson et al. [25]. The feasibility of using variable time steps in electrical TLM stubs has been demonstrated by Hui et al. [26].

2.2.2 Transmission Line Equations

Most physical elements can be described by four main properties: resistance (R), conductance (G), capacitance (C) and inductance (L). Models of such objects are referred to as *transmission line elements*. Even though these terms are normally used for electric systems, their mathematical meaning can be applied to any kind of power-transmitting system. This includes for example mechanical systems, fluid networks, acoustics and magnetics. An equivalent circuit for a transmission line element is shown in Fig. 2.1. If resistance and conductance are both zero, the element is referred to as *lossless*. It will thus only contain the frequency domain properties L and C , as shown in Fig. 2.2.

Figure 2.1 The four properties of a transmission line, resistance (R), conductivity (G), capacitance (C) and inductance (L), shown in an electric circuit.

Combining these four properties yields the impedance of the transmission line. Input impedance is defined as the ratio between input pressure and flow. Characteristic impedance (Z_C) is defined as the input impedance for an infinitely long transmission line. In general, this can be seen as the complex equivalent of resistance. It considers not only stationary processes, but also phase shifts. The expression for the characteristic impedance in Fourier space (Eq. (2.1)) can

Figure 2.2 In a lossless transmission line, both resistance and conductance are zero.

be derived from the telegrapher's equations. The same equation for a lossless line is shown in Eq. (2.2).

$$Z_C = \sqrt{\frac{R + j\omega L_d}{G + j\omega C_d}} \quad (2.1)$$

$$Z_C = \sqrt{\frac{L_d}{C_d}} \quad (2.2)$$

The time delay in a transmission line can be expressed as the square root of the inductance and the capacitance per length, multiplied by the length, see Eq. (2.3)

$$T = \sqrt{L_d C_d} \Delta l \quad (2.3)$$

Capacitance or inductance per unit length can be lumped into a total capacitance or inductance for the whole line by simply dividing by the length, see equations Eq. (2.4) and Eq. (2.5).

$$C = C_d \Delta l \quad (2.4)$$

$$L = L_d \Delta l \quad (2.5)$$

Combining Eqs. (2.2) to (2.5) then yields Eq. (2.6):

$$T = \frac{L}{Z_C} = C Z_C = \sqrt{LC} \quad (2.6)$$

The final step is to rewrite the capacitance as a function of the physical properties of the element. This can, for example, be the size of a hydraulic volume and the bulk modulus of the fluid, see Eq. (2.7). This yields the expression for the hydraulic characteristic impedance, as can be seen in Eq. (2.8)

$$C = \frac{V}{\beta_e} \quad (2.7)$$

$$Z_c = \frac{\beta_e T}{V} \quad (2.8)$$

2.2.3 Derivation

For the derivation, the fluid power domain will be used. Hence, intensity and flow variables will be represented by pressure (p) and flow (q). Consider a transmission line modelled as a pipe with pressure and input flow on each side as boundary values, see Fig. 2.3. A wave propagates through the pipe at time T ,

Figure 2.3 A transmission line can be modelled as a pipe with fluid pressure and input flow on each side as boundary values.

which equals the length of the pipe divided by the speed of sound in the fluid. The relationship between pressure and flow is determined by the continuity equation and the Navier-Stokes equation. If the flow is assumed laminar, if the pipe is assumed non-elastic, and if all tangential flow is ignored, these can be expressed according to Eqs. (2.9) and (2.10) [27].

$$\frac{\partial p(t, x)}{\partial t} = -a Z_C \cdot \frac{\partial q(t, x)}{\partial x} \quad (2.9)$$

$$\frac{\partial p(t, x)}{\partial x} = -\frac{Z_C}{a} \cdot N(t) \cdot \frac{\partial q(t, x)}{\partial t} \quad (2.10)$$

These are known as the telegrapher's equations [28], and describe wave propagation through the pipe. Similar equations can be derived for other physical domains, such as mechanics and electrics. By eliminating p , a second order differential equation can be obtained, see Eq. (2.11). This is known as the wave equation.

$$\frac{\partial^2 q(t, x)}{\partial x^2} = \frac{N(t)}{a^2} \frac{\partial^2 q(t, x)}{\partial t^2} \quad (2.11)$$

The general solution to Eq. (2.11) in Laplace space is represented by Eq. (2.12):

$$Q(s, x) = C_1 e^{sx\beta} + C_2 e^{-sx\beta} \quad (2.12)$$

where

$$\beta = \frac{\sqrt{N(s)}}{a}$$

Inserting Eq. (2.12) into Eq. (2.9) yields Eq. (2.13):

$$P(s, x) = -Z_c \sqrt{N(s)} \left[C_1 e^{sx\beta} + C_2 e^{-sx\beta} \right] \quad (2.13)$$

Boundary values for the first end of the pipe are then used to calculate constants C_1 and C_2 . Letting $P(s, x = 0) = P_1(s)$ and $Q(s, x = 0) = Q_1(s)$ yields

$$C_1 = \frac{1}{2} \left(Q_1(s) - \frac{P_1(s)}{Z_c \sqrt{N(s)}} \right) \quad C_2 = \frac{1}{2} \left(Q_1(s) + \frac{P_1(s)}{Z_c \sqrt{N(s)}} \right) \quad (2.14)$$

Finally, the boundary values for the second end yields the relationship between pressure and flow on both ends of the pipe. Letting $P(s, x = L) = P_2(s)$ and $Q(s, x = L) = Q_2(s)$ where $L = aT$ results in Eq. (2.15), which is known as the *four-pole equation*.

$$\begin{bmatrix} A_L & B_L \\ C_L & D_L \end{bmatrix} \times \begin{bmatrix} Q_1 \\ P_1 \end{bmatrix} = \begin{bmatrix} -Q_2 \\ P_2 \end{bmatrix} \quad (2.15)$$

where

$$\begin{aligned} A_L &= D_L = \cosh Ts\sqrt{N} \\ B_L &= -\frac{1}{Z_c \sqrt{N}} \sinh Ts\sqrt{N} \\ C_L &= -Z_c \sqrt{N} \sinh Ts\sqrt{N} \end{aligned}$$

The most important observation is that the variable x has been eliminated. Each end thus depends only on the other end of the pipe. $N(s)$ is a frequency-dependent friction factor. Using the definition of hyperbolic functions and omitting friction ($N = 1$), Eqs. (2.16) and (2.17) can be obtained.

$$\frac{(e^{Ts} + e^{-Ts})}{2} Q_1(s) - \frac{(e^{Ts} - e^{-Ts})}{2Z_c} P_1(s) = -Q_2(s) \quad (2.16)$$

$$-Z_c \frac{(e^{Ts} - e^{-Ts})}{2} Q_1(s) + \frac{(e^{Ts} + e^{-Ts})}{2} P_1(s) = P_2(s) \quad (2.17)$$

Subtracting Eq. (2.16) from Eq. (2.17) yields Eq. (2.18).

$$P_1(s)e^{Ts} = P_2(s) + Z_c(Q_2(s) + Q_1(s)e^{Ts}) \quad (2.18)$$

Transforming Eq. (2.18) back to the time domain gives the transmission line equation:

$$p_1(t + T) - Z_c q_1(t + T) = p_2(t) + Z_c q_2(t) \quad (2.19)$$

And due to symmetry:

$$p_2(t + T) - Z_c q_2(t + T) = p_1(t) + Z_c q_1(t) \quad (2.20)$$

Hence, it is clear that the pressure on one side at a given moment in time is a function of the flow on the same side at the same time, and the pressure and flow on the other end T seconds ago. The latter represents the wave propagating through the pipe. The two sides of the line are thus decoupled and can be solved independently. It should be noted that these equations equal the trapezoid rule of integration but with twice the time step. Consequently, TLM is the only unconditionally stable explicit single-step integration method [24]. To simplify the equations further, the wave information can be expressed as *wave variables*:

$$c_1(t + T) = p_2(t) + Z_c q_2(t) \quad (2.21)$$

$$c_2(t + T) = p_1(t) + Z_c q_1(t) \quad (2.22)$$

By combining Eqs. (2.21) and (2.22) with Eqs. (2.19) and (2.20), the transmission line equations can be obtained in their standard form:

$$p_1(t + T) = c_1(t + T) + Z_c q_1(t + T) \quad (2.23)$$

$$p_2(t + T) = c_2(t + T) + Z_c q_2(t + T) \quad (2.24)$$

This means that pressure and flow only need to be calculated at the ends of a line. Calculation time for a line model will therefore be independent of the length of the line. This makes it possible to create much more economical models than those using distributed pressures and flows [20].

2.2.4 Parasitic Inductance

According to Eq. (2.6), a time delay requires both L and C to be different from zero. In reality, the size of T will be decided by the sizes of C and L . In simulation, however, it is often desirable to use the same time delay for every TLM element in the model. This can be achieved by inserting the elements in parts of the model with a large capacitance and a small inductance. The correct value of C and the desired value of T can then be used. While this introduces an error in L , the size of the error is likely to be insignificant. This phenomenon is known as *parasitic inductance* (L_P).

One method of combating this is to compensate by reducing inductance in neighbouring components in the model. This, however, assumes that the total neighbouring inductance is larger than L_P . If not, the only remaining solution is to reduce the time delay of the TLM element. Compensating for parasitic inductance can be automated by the simulation tool. To illustrate the process, we use a model of a hydraulic piston, shown in Fig. 2.4. The piston is connected to a mechanical spring and two hydraulic volumes, all modelled as TLM elements. Each such element gives rise to its own parasitic inductance. The mass

Figure 2.4 TLM elements may cause parasitic inductance, which need to be compensated for.

Figure 2.5 Step response with TLM elements differs from the ideal response due to parasitic inductance.

of the piston is 10 kg. After 1 second the input pressure is instantly increased from 1 bar to 101 bar, which gives rise to a step response, see Fig. 2.5. The uncompensated TLM response is compared with an ideal response, calculated by letting $T \rightarrow 0$. As can be seen, the responses differ significantly from each other. Before we can do any compensating, it is necessary to know the magnitude of the three inductances. The equation for inductance can be found by rearranging Eq. (2.6):

$$L = Z_C T \quad (2.25)$$

Figure 2.6 With compensation for parasitic inductance, step response equals the ideal response.

A time step of 0.1 ms is used. Characteristic impedance of the spring is defined by Eq. (2.26), where K_S is the spring stiffness:

$$Z_C = K_S T \quad (2.26)$$

With a stiffness of $K_s = 100$ kN/m, the parasitic inductance hence becomes 0.001 kg. For the hydraulic volumes, the impedance depends on the size of the volume (V) and the effective bulk modulus of the fluid (β_e) according to Eq. (2.27):

$$Z_C = \frac{\beta_e T}{V} \quad (2.27)$$

With $V = 0.1$ l and $\beta_e = 1$ GPa, the hydraulic inductance becomes 100000 kg/m⁴. This can be converted to mechanical friction by multiplying with the cylinder area (A_p) squared:

$$L_{mec} = L_{hyd} A_p^2 \quad (2.28)$$

In this case, the cylinder area is 0.001 m². This yields a mechanical parasitic inductance of 0.1 kg from each volume. The total parasitic inductance that needs to be compensated for is then 0.201 kg. We therefore reduce the piston inertia from 10 kg to 9.799 kg. Simulation results after the compensation are shown in Fig. 2.6. A comparison of the power spectral densities of the response with and without compensation is shown in Fig. 2.7.

Figure 2.7 Power spectral densities of a step response with and without compensation for parasitic inductance.

2.2.5 Parallel Simulations

Its decoupled nature makes the TLM method suitable for parallel simulations. Several experiments on this have been made. Early attempts using transputer technology to decouple fluid networks were made by Burton et al. [29]. This evolved into an early simulation environment for parallel TLM simulations [30]. Similar attempts were later made on electronic circuits by Fung and Hui [31], where a speed-up of 1.6 could be achieved with 2 transputers. An experiment using parallel desktop computers with network communication was made by Krus et al. [19]. This was subsequently used for real-time simulation by Jansson and Krus [32]. TLM was later used by Nyström and Fritzson for explicit decoupling of equation-based models in the Modelica language [33]. A related attempt using automatic decoupling by utilizing model structure was recently made by Sjölund et al [34]

TLM can also be used for simulating electromagnetic wave propagation, known as the transmission line matrix method [35]. Such models include a very large number of independent nodes and thus offer a high degree of parallelism. Several experiments on supercomputers have been performed. First, it was shown that speed-up could be achieved on up to 25 cores [36]. This was later improved with speed-up on up to 5000 cores [37]. Subsequently, scalable simulation with speed-up on up to 127,500 cores was achieved [38].

A related use of TLM is co-simulation between simulation tools. One such experiment involves connecting the Beast roller bearing simulation tool by SKF with the multi-body dynamics simulation tool MSC Adams. A meta-modelling

language for parallel co-simulations using TLM has been presented by Siemers et al. [39], where it was shown that the number of arms in a simulated mechanical pendulum had only a very small effect on the total simulation time.

2.2.6 Related Methods

Several other methods for decoupling of simulation models exist. One example is waveform relaxation [40], where systems are decomposed into independent equations by treating unknown variables as inputs. All sub-systems are then solved iteratively from an initial guess of the solution using a relaxation method such as *Gauss-Seidel* or *Gauss-Jacobi*. Related methods are dynamic decoupling [41][10] and to decouple models by relaxing data dependencies [42]. Another option is to use modular time-integration [43]. In contrast to TLM, all those methods require tools for monitoring and minimizing numerical errors. On the other hand, TLM can induce modelling errors due to parasitic inductances. One option, of course, is to use different methods for different parts of the model. The above-mentioned methods are all based on parallelization across the model. An alternative approach is to use parallel solver methods, as described in Section 2.1. While these methods also offer parallel simulation, they can be used in conjunction with TLM. As long as the number of available processing units is large enough, each decoupled part of the model can in turn be decomposed into independent sub-parts.

2.3 Applications

Distributed simulations can reduce simulation time. This both enables more powerful simulation techniques and shortens the design cycle for new products. An example of a simulation technique with high demands on performance is real-time simulation. Examples of methods important for the design cycle include design optimization and sensitivity analysis. These methods can be very time consuming and thereby costly. Real-time is explained further in Section 8.1. Optimization is discussed in Chapters 6 and 10

2.3.1 Sensitivity Analysis

Sensitivity and uncertainty analysis are methods for investigating how uncertainties of output variables depend on uncertainties in certain input variables. The purpose can be to better understand the model, discover modelling errors or identify the importance of specific input variables. For example, models can be simplified by finding and removing non-significant variables. If a mathematical analysis of the model is not feasible, it is necessary to simulate the model with different parameters and compare the results. Luckily, all simulations are independent of each other, which makes it possible to simulate multiple models in parallel. Running parallel models will give higher speed-up than parallelizing the model itself in most cases. Using model-level parallelism is therefore unsuitable as long as the number of parameter points is larger than the number of available processing units.

3

Hopsan Simulation Tool

Hopsan is an open-source simulation software for fluid, mechanical and electrical systems [44]. The first version of Hopsan was written in Fortran with a graphical user interface written in Visual Basic [45]. Development first began in 1977 at the Division of Hydraulics and Pneumatics at Linköping University [46]. Since then it has been widely used for industrial applications, research and education. All benchmark components used in this thesis (see Chapter 7) are included in the official downloadable release. This makes it possible for other researchers to repeat the multi-threading experiments. Like most other simulation tools, the first version of Hopsan used a centralized solver algorithm. In 1985, a *predictor-corrector* solver was implemented, making it possible to solve stiff differential equations [47]. In the early 1990s, distributed solvers using the transmission line element method were adopted, which offered better scalability and improved numerical robustness [46].

Over the years, increased complexity in combination with outdated programming solutions made further maintenance and development ineffective. Development of a new generation based on modern tools was therefore launched in 2009. This version is object-oriented, cross-platform, and written in C++ [X]. Most features in the first version of Hopsan have been re-implemented in the new version, such as scripting, numerical optimization, sensitivity analysis, and frequency analysis. Hopsan mainly consists of three modules: the simulation core, a graphical user interface and a command line interface. There are also modules for code generation, remote communication and symbolic operations. Each module is compiled separately, and the simulation core can be used independently of the user interface.

3.1 Simulation Core

The Hopsan simulation core is an object-oriented dynamic library written in C++. It has an API that allows creating and modifying models, running simulations, and collecting data variables. A model consists of a system object, which contains several component objects. It may also contain other system objects, which are denoted sub-systems. The component objects contain equations, solvers and parameters. Each component contains one or more port objects. Connections between components are implemented as node objects, which are used to connect two or more port objects with each other. The node objects contain the simulation variables and logging functions. See Fig. 3.1 for an overview of the classes in the simulation core. All component objects are pre-compiled as dynamic libraries, which are linked from HopsanCore at run-time. No compilation is thus required prior to simulation. In this way, it is possible to quickly reconfigure a model by rearranging or replacing components. This can for example make it possible for an optimization algorithm to modify the model structure without losing performance.

Figure 3.1 The Hopsan simulation core consists of classes for systems, components, ports and nodes.

All simulations in Hopsan use fixed time steps and distributed solvers. Even though variable step size can indeed be used with distributed solvers, experiments have shown that the benefits are modest [25]. Components in Hopsan are divided into three types: capacitive components (C-type), resistive components (Q-type), and signal components (S-type). The C-type components represent the TLM elements and are used to decouple the Q-type components. Consequently, all C-type components are independent of other C-type components and all Q-type components of other Q-type components. A simulation is performed by means of a loop. Each step consists of first calculating the

C-type components, then the S-type, then the Q-type, and finally performing the data logging. Fig. 3.2 shows how the components in a simple hydraulic circuit are simulated type by type.

Figure 3.2 Each time step in Hopsan consists of first solving the C-type components then the S-type and finally the Q-type.

3.2 Graphical Interface

Hopsan has a graphical user interface created with Qt, an open-source cross-platform application framework [48]. Communication with the simulation core API is handled by a wrapper object. The interface consists of a workspace area, a library widget, and a message widget, see Fig. 3.3. Simulation results are displayed in plot windows, which contain all data analysis tools such as frequency analysis and data import or export. Most functionality can be controlled by a scripting terminal.

Other tools in the program include numerical optimization, sensitivity analysis, and loss calculation. Simulations can also be animated, either in real-time or as a replay of a finished simulation.

3.2.1 Scripting

Hopsan simulations can be scripted through the interpreted scripting language HCOM [45]. Commands have intentionally been made short to facilitate the everyday work for the user. This language is the basis for the numerical optimization tool. An example of an optimization script in HCOM is shown in Listing 3.1.

Listing 3.1 *The optimization tool in Hopsan is controlled by the HCOM scripting language.*

```
#Set parameter function
define setpars
    chpa GainP.k.y optpar(optvar(evalid),0)
    chpa GainI.k.y optpar(optvar(evalid),1)
enddefine

#Cost function
define obj
    #Absolute Average Difference
    error = Position_Sensor.out.y-Step.out.y
    abs error
    obj1 = aver(error)

    #Overshoot
    maxVal = max(Position_Sensor.out.y)
    if(maxVal > 0.7)
        obj2 = maxVal-0.7
    else
        obj2 = 0
    endif

    totalObj = +5*1*exp(2)*obj1+1*1*exp(2)*obj2
    opt set obj optvar(evalid) totalObj
enddefine

opt set algorithm complexrf
opt set maxevals 100
opt set npoints 4
opt set nparams 2
opt set functol 1e-05
opt set partol 0.0001
opt set alpha 1.3
opt set rfak 0.1
opt set gamma 0.3
opt set limits 0 0 0.03
opt set limits 1 0 0.003

opt run
```

Figure 3.3 The graphical user interface in Hopsan.

3.3 HopsanGenerator Module

An external module called HopsanGenerator is used for import, export, code generation and compilation. It must be linked against HopsanCorebut can be used independently of the graphical user interface. There are two basic functions, shown in Fig. 3.4. First, it can import external source code and compile it into a Hopsan component. Second, in the opposite direction, it can generate and compile code from a Hopsan model that can be used by external tools. When compiling component libraries, the same version of the MinGW compiler must be used.

Figure 3.4 The HopsanGenerator module is used for importing and exporting simulation code.

4

Simulation Tool Connectivity

Simulation tools are usually tailor-made for a specific kind of problems. It is therefore suitable to model each sub-part of a product in the best-suited tool. Different organizations, departments or professional communities may also favour different tools. Simulating each sub-part alone, however, makes it difficult to understand the product as a whole. It is thus necessary to simulate all sub-parts together. Connecting models from different simulation programs is called co-simulation. This is especially important for design optimization, sensitivity analysis and uncertainty analysis, where changes in one sub-part can affect the behaviour of another one. Cutting apart an equation system into sub-systems will introduce time delays in variables at the intersection points. For this reason, TLM is suitable, since it can replace flexible elements in the model with physically motivated time delays. In this way, numerical errors can be avoided. Instead, an explicit and predictable modelling error will be introduced.

4.1 Functional Mock-Up Interface

The Functional Mock-up Interface (FMI) is an open standardized interface for co-simulation and model exchange between simulation tools. It was developed by the MODELISAR consortium in 2009 [49]. At the time of writing FMI is supported by 77 tools [50]. The purpose of the standard is to allow models to be interchanged between different vendors, departments or modelling disciplines. Each sub-part can then be modelled in the most appropriate tool. When used for model exchange, models are provided as equations only, and state variables must be integrated by the host application. With co-simulation on the other hand, models are provided with included solvers, and can be used

Figure 4.1 A Functional Mock-up Unit is a zip file that contains pre-compiled binaries or source code with an XML description file.

as stand-alone modules in co-simulation environments. Models are compiled into binary files with a pre-defined C interface. This ensures that FMUs are compiler independent. They must, however, still be compiled and used on the same platform. An FMU contains, among other things, functions for initializing, simulating and accessing data variables. Each model is shipped as a Functional Mock-up Unit (FMU), which is a ZIP package with the FMU file extension. This can contain pre-compiled libraries for various targets, source code, and an XML description file. The last contains information about variables, parameters, as well as general properties and capabilities of the FMU. Optional contents are documentation and graphical icons, see Fig. 4.1. The host program loads an FMU by parsing the XML file and then linking against the library.

The FMI interface is based on platform dependent type definitions and methods. With the FMI 2.0 standard, model exchange and co-simulation shares a common interface. There is thus no need to use separate standards. Some of the fundamental methods of the API are shown in Listing 4.1. The `fmi2Status` type is an enumeration returned by all functions to indicate success or failure. The information about the FMU is stored in the `fmi2Component` pointer, which is passed around between functions. Variables are referenced by the `fmi2ValueReference` handle and represented by custom data types, for example `fmi2Real` and `fmi2Integer`.

Listing 4.1 *Some methods available in the FMI 2.0 interface for co-simulation and model exchange.*

```
fmi2Component fmi2Instantiate(fmi2String instanceName,
                             fmi2Type fmuType,
                             fmi2String fmuGUID,
                             fmi2String fmuResourceLocation,
                             const fmi2CallbackFunctions* f,
                             fmi2Boolean visible,
                             fmi2Boolean loggingOn);

void fmi2FreeInstance(fmi2Component c)

fmi2Status fmi2SetupExperiment(fmi2Component c,
                              fmi2Boolean toleranceDefined,
                              fmi2Real tolerance,
                              fmi2Real startTime,
                              fmi2Boolean stopTimeDefined,
                              fmi2Real stopTime)

fmi2Status fmi2EnterInitializationMode(fmi2Component c)

fmi2Status fmi2ExitInitializationMode(fmi2Component c)

fmi2Status fmi2Terminate(fmi2Component c)

fmi2Status fmi2Reset(fmi2Component c)

fmi2Status fmi2GetReal(fmi2Component,
                      const fmi2ValueReference vr[],
                      size_t nvr,
                      fmi2Real value[])

fmi2Status fmi2SetReal(fmi2Component,
                      const fmi2ValueReference vr[],
                      size_t nvr,
                      fmi2Real value[])

fmi2Status fmi2GetRealOutputDerivatives(fmi2Component c,
                                         const fmi2ValueReference vr[],
                                         size_t nvr,
                                         const fmi2Integer order[],
                                         fmi2Real value[])

fmi2Status fmi2SetRealInputDerivatives(fmi2Component c,
                                       const fmi2ValueReference vr[],
                                       size_t nvr,
                                       const fmi2Integer order[],
                                       const fmi2Real value[]);

fmi2Status fmi2DoStep(fmi2Component c,
                    fmi2Real currentCommunicationPoint,
                    fmi2Real communicationStepSize)

fmi2Status fmi2CancelStep(fmi2Component)

fmi2Status fmi2GetStatus(fmi2Component c,
                       const fmi2StatusKind s,
                       fmi2Status* value)
```

4.2 Related Work

An overview of methods and stability criteria for co-simulation was presented in [51]. A framework for tool-independent equation-based models by using open-ended model compilers was also proposed. In [52], a framework for asynchronous co-simulation of simulation components using TLM was presented. Interpolation was used for asynchronous exchange of data variables. An important finding was the relationship between TLM step size and maximum time step in each program. If both programs can produce output variables at a specified synchronization interval, it is sufficient that both programs have a time step smaller than or equal to the TLM step. If at least one of the programs cannot deliver data at the specified interval, however, the maximum time step must be limited to half the TLM step. This guarantees that variables with the desired time delay can always be interpolated between two communication points. As a continuation of this work a general meta-modelling language for connecting sub-models through TLM elements was developed [39]. The language only supports TLM connections because stability in the connections is important. In addition, TLM only causes explicit and predictable modelling errors instead of numerical errors. It was shown that when simulating a pendulum, the execution time does not increase significantly when the number of arms increases, due to the parallel execution.

5

Equation-Based Modelling

Programming languages can be either imperative or declarative. Imperative languages, for example C and C++, are based on statements that modify the current state of the program. Together, these statements define *how* to solve a specific problem. Declarative languages, on the other hand, aim at describing the problem itself. How it is solved is then defined by the underlying language implementation. Equation-based object-oriented languages (EOOs) are one example of declarative languages. A simulation model is defined by a set of equations and parameters. Each tool supporting the language can then use an arbitrary numerical method to solve the equation system. Since there is no need to use statements, it is not necessary to rearrange the equations into assignments. In addition, equation-based languages do not impose a fixed causality in the model. This makes the modelling process easier and more intuitive. Examples of equation-based languages include Modelica [53] and Simulink/SimScape[54].

5.1 Modelica

Modelica is an equation-based object-oriented hybrid declarative language for component-oriented modelling [53]. It supports both continuous-time and discrete-event simulation. It is managed by the non-profit Modelica Association. Examples of Modelica tools include OpenModelica [55], JModelica [56], System Modeler [57], Dymola [58], SimulationX [59] and MapleSim [60].

A system in Modelica consists of **classes**, also called **models**, which contain a set of equations and parameters. These can be instantiated into objects in the system. A class can also contain instances of other classes. Since Modelica is object-oriented, it also supports class inheritance. Another important ele-

Listing 5.1 *A Modelica model of a linear inertia with two flanges*

```
model LinearInertia "Inertia"
  Flange P1, P2;
  parameter Real M(unit="kg")    = 100 "Mass";
  parameter Real B(unit="Ns/m")  = 10  "Damping";
  parameter Real k(unit="N/m")   = 1   "Spring coefficient";
equation
  M*der(P2.v)+B*P2.v+k*P2.x = P1.f-P2.f;
  der(P2.x) = P2.v;
algorithm
  P2.x = -P1.x;
  P2.v = -P1.v;
end LinearInertia;
```

Listing 5.2 *A mechanical connector in the Modelica language*

```
connector Flange "Mechanical Flange"
  Position(unit="m")    x "Position";
  Velocity(unit="m/s")  v "Velocity";
  flow Force(unit="N")  f "Force";
end Flange;
```

ment is the **connector**, which can be used to connect classes with each other. An example of a Modelica model is shown in Listing 5.1. It consists of a linear inertia with two flanges, represented as connectors, see Listing 5.2. The **der()** operator is used to specify derivatives of state variables. These must be integrated by the solver using a suitable numerical integration method.

Equations in Modelica are divided into **assignments** and **equalities**. Assignments define one variable as a function of the right-hand expression:

```
x := y + 3;
```

Equalities on the other hand define an acausal relationship between the left-hand and the right-hand expressions:

```
x + y = z + 3;
```

Modelica also supports time events and state events for discrete-time modelling. It is possible to mix continuous-time and discrete-time modelling, known as **hybrid modelling**. This can for example be useful for modelling of non-continuities.

5.2 Parallel Simulation in Modelica

Several experiments on using parallel hardware to improve simulation speed in equation-based languages have been conducted. Methods include task-merging algorithms [61], software pipelining [62] and parallel schedules for flattened equations [63]. A parallel extension to OpenModelica using explicit language constructs has also been developed [64]. Several authors have pointed out that ODE solvers for continuous-time and time-invariant systems without algebraic loops can be run in parallel on GPU cards [65][66][67]. Another use of GPU technology is array computations [68]. A different approach is to use distributed solvers in Modelica by using TLM [23]. This was used for coarse-grained parallel simulation on computer clusters by [33]. In [34], automatic parallelization with TLM elements by utilizing dependency analysis was demonstrated.

6

Optimization Algorithms

Design optimization is the use of simulation models to find an optimal solution to a given problem. A solution can be either a design concept or a set of parameters, represented by a point in the parameter space. The quality of each solution is computed by a cost function. Choosing the most suitable algorithm for optimization can have great impact on performance and results. It is desirable to obtain a high convergence rate while also having a high probability of finding the global optimum. Combining these goals can be difficult, which is reflected by the large number of available algorithms. Algorithms can be divided into gradient-based and non-gradient-based methods. For general models, which can be non-linear or discontinuous, the gradient cannot always be obtained. Only non-gradient-based algorithms have therefore been considered in this thesis.

6.1 Direct-Search Methods

Direct-search methods search the parameter space with a few trial points. At each iteration, one or more of these points are moved depending on the locations and objective values of the remaining points. This requires no information about the gradient of the objective function. Direct-search methods can therefore be used on non-differentiable or non-continuous problems. Early examples of direct-search methods include pattern search [69]. Later on, methods using reflection, such as the Nelder-Mead simplex and the Complex methods were introduced. Direct-search methods typically have a high convergence rate, require few evaluations and are easy to parameterize. On the other hand, they are more difficult to parallelize than population-based algorithms.

6.1.1 Pattern Search

Pattern search is a group of non-gradient-based optimization methods, where the parameter space is searched by using a pre-defined pattern [69]. Fig. 6.1 shows the simplest implementation, as explained in [70].

Figure 6.1 Pattern search optimization uses a pattern emanating from the best known point to search the parameter space.

At each iteration step, it is tested whether or not an increase or decrease of each parameter gives an improvement. If so, the next iteration starts from the new best point. Should no improvement be found, the step size is halved and the procedure is repeated. With n parameters, every iteration step will evaluate $2n$ or $2n - 1$ points. Since the evaluations are independent, the method is parallelizable up to this value. Theoretical speed-up as a function of number of processing units is shown in Fig. 6.2. It is possible to increase the parallelizability further by using larger search patterns [71].

Figure 6.2 The pattern search algorithm can be easily parallelized until the number of processors equals twice the number of parameters.

6.1.2 Nelder-Mead Method

The Nelder-Mead method is a heuristic method first proposed in 1965 [72], based on the simplex method devised by Spendley et al. [73]. For a problem with n parameters, it uses $n + 1$ search points, denoted the simplex. The initial step is to randomly distribute all points in the parameter space and evaluate them. At each iteration step, the worst point is then reflected through the centroid of the remaining points, as shown in Fig. 6.3a. If the reflected point is better than the previously best point, the simplex is expanded by moving the point further in the reflection direction, see Fig. 6.3b. Should the reflected point be worse than the previously best point, the simplex is contracted by moving the point towards the centroid, see Fig. 6.3c. If the contracted point is the worst point in the simplex, the simplex is reduced by moving all points towards the best point, as shown in Fig. 6.3d. The new worst point is then reflected until the optimization converges.

While the Nelder-Mead method is naturally sequential, some possibilities for parallelization exist. Attempts to run Nelder-Mead optimizations in parallel have been made by using multi-directional search [74], and by reflecting multiple points at each iteration step [75]. A trivial approach is to pre-evaluate the expansion, contraction and reduction points in parallel with the reflection at each iteration step [76].

Figure 6.3 The nelder-mead method uses reflection (a), expansion (b), contraction (c) or reduction (d) to search the parameter space.

6.1.3 Complex-RF Algorithm

A drawback with simplex methods is that they tend to easily collapse into sub-dimensions when they approach constraints. This was addressed by Box in 1965 [77], which resulted in the related Complex (constrained simplex) algorithm, later improved by Guin in 1968 [78]. Compared to the Nelder-Mead method, the Complex method typically uses a greater number of search points. This increases the probability of the Complex turning around when a constraint is reached. Similarly to the Nelder-Mead method, each iteration step begins with reflecting the worst point through the centroid of the remaining points, see Fig. 6.4a,b. An important difference, however, is the use of over-reflection ($\alpha > 1$). If the reflected point is still worst, it is gradually moved towards the centroid, according to Fig. 6.4c. To avoid locking to the centroid in case that the centroid is worse than the reflected point, the point can also be moved gradually towards the best point. The Complex will alternately grow due to the over-reflection and shrink due to points being moved towards the centroid.

The Complex algorithm can be improved by adding a random factor and a forgetting factor, as explained by Krus in 2003 [79]. This method is called Complex-RF. The random factor reduces the risk of converging at local optima, while the forgetting factor ensures that the Complex consists predominantly of recently evaluated points. This will introduce a filtering effect, which improves the abilities of crossing discrete steps in the cost functions. The Complex-RF algorithm has also been extended to the Complex-RD algorithm for mixed discrete optimization methods [80].

Figure 6.4 The Complex method uses reflections (a,b) and retractions (c,d) with longer reflections and more search points than the Nelder-Mead method.

6.2 Population-based Methods

Population-based algorithms are a family of heuristic methods that use large populations of independent search points. All points are evaluated independently at each iteration step, which makes such algorithms inherently parallel. On the other hand, the large number of points usually results in a slow convergence rate. Population-based methods can also be sensitive to changes in their parametrization [81]. Examples of population-based methods include particle swarm, firefly and evolutionary algorithms.

6.2.1 Particle Swarm Optimization

Particle swarm optimization (PSO) is a family of optimization algorithms based on social behaviour and swarm intelligence [82]. An optimization consists of a large number of independent particles that independently search the parameter space. Fig. 6.5 illustrates the algorithm. Each particle has a velocity vector, which determines its position at the next iteration. At each iteration step, the velocity vector is changed towards both the particle's own best known point and the globally best known point. Each particle also has an inertia that limits

rapid changes in velocity. This prevents the swarm from collapsing too early and ensures that a large part of the parameter space is searched before the swarm converges to an optimum. Choosing the inertia parameter is a trade-off between exploration and exploitation. A large inertia will make particles less attracted by the best points, hence searching a larger part of the design space. A small inertia makes the swarm converge faster, but with a smaller chance of finding the optimal solution. Strategies for dynamically changing inertia weight during optimization have been discussed in several publications [83][84][85].

Figure 6.5 Particle swarm optimization algorithms are based on social behaviour and swarm intelligence.

Since all particles are evaluated independently, PSO offers good opportunities for parallelism [86]. The degree of parallelism is only limited by the number of particles in the swarm, see Fig. 6.6. It is indeed possible to increase this limit to use an unlimited number of central processing units (CPUs) by successively increasing the number of particles. However, this may also change the behaviour of the algorithm. It should be pointed out that the particles are independent not only in parameter space, but also in time. When evaluation of a particle is completed, it does not need to wait for other particles before evaluating its next step. It is thus possible to use asynchronous parallelism [87].

Figure 6.6 For population-based algorithms, parallelism is only limited by the number of points.

6.2.2 Evolutionary Methods

Evolutionary algorithms (EA) are methods that mimic concepts of evolution, such as mutation, reproduction and natural selection. The most well-known example is genetic algorithms (GA) [88]. The optimization is initialized with a set of randomly generated individuals. Each individual represents a set of parameters, denoted its genome. After each evaluation, individuals are stochastically chosen for reproduction. Individuals with good objective values have a higher chance of being selected. New individuals are generated for the next iteration by randomly merging the genomes of the parents. A very powerful GA is differential evolution (DE) [89], which is based on mutation and recombination of trial vectors. Parallelization properties of EA are very similar to those of PSO. Therefore, only PSO has been included in the parallel optimization experiments in this thesis.

Part III

Contributions

7

Multi-threaded Simulations

With multi-core technology, computers can perform multiple tasks simultaneously. While this can potentially provide significant speed-up, it also makes the programming work far more difficult. Taking advantage of parallel hardware is becoming increasingly important for simulation software developers. System simulation can be parallelized either across the solver or across the system. The first approach has been discussed in several publications, as described in Section 2.1. With the second approach, equation systems are divided into several smaller, independent, sub-systems. This induces a time delay in the interface variables. With TLM, time delays can be modelled with a physical motivation. Hence, no numerical errors are introduced. While modelling errors may occur, these have a physical meaning and can be treated as model parameters. In contrast to numerical errors, the modelling errors are predictable and controllable. In addition, the numerical stability of the model is not affected by introducing the delays. TLM is thus a very suitable method for parallel simulation.

7.1 Implementation

The Hopsan simulation tool is used for the implementations and experiments, see Chapter 3. The simulation loop in Hopsan is shown in Fig. 7.1. Each iteration consists of two parallel bag-of-task problems, for C-type and Q-type sub-models, respectively. This is quite similar to the odd-even communication group pattern as described in [90]. There are also two sequential tasks, namely the signal sub-models (S-type) and logging of output variables. Such tasks can also be parallelized to a certain degree, but this has not been investigated further due to their relatively small workload. The critical path length of

each simulation step with unlimited number of cores is thus the sum of the execution time for the sequential tasks, the slowest C-type sub-model and the slowest Q-type sub-model. Parallelization overhead is expected to arise from load imbalance, thread spawning and synchronization.

A key objective with the implementation was that TLM elements should be placed explicitly by the user. Unlike fully automatic strategies this ensures that the model is never modified without user interaction. The actual parallelism is still automated, so that users require no knowledge of parallel computing. The implementation targets shared memory systems, mainly because a high rate of data exchange is expected. This can become a bottleneck on distributed memory systems, where variables need to be passed around between processors. A drawback with shared memory, however, is that it suffers from poor scalability [91]. Distributed memory could thus be advantageous when simulating a large number of relatively large sub-models. Moreover, only task parallelism has been considered. Data parallelism is not suited for the problem at hand, since simulation models are almost exclusively heterogeneous with different equations in each sub-model. Multi-threading was implemented by using Intel® Threading Building Blocks (TBB), a C++ template library for writing multi-threaded software [90]. Advantages include cross-platform support and a large set of pre-defined template patterns. It is also written in C++ and is thus compatible with Hopsan. Tab. 7.1 shows the TBB classes that have been used:

<code>tbb::atomic</code>	Atomic variable
<code>tbb::mutex</code>	Mutex lock
<code>tbb::task_group</code>	Group of tasks executed in parallel
<code>tbb::parallel_for</code>	Parallel for-loop

Table 7.1 *Template classes from TBB used for the multi-threaded implementation*

Figure 7.1 *Sub-models in Hopsan are simulated in a loop, one component type at a time.*

7.2 Profiling

Perfect load balancing is not possible in most simulations, because the required execution time varies between different sub-models and also over time. If the model consists of many sub-models or if the difference in simulation time between them is small, however, it is possible to achieve a decently balanced partitioning. The sub-division of a model into smaller parts consists of the following steps:

1. Measure execution time for each sub-model
2. Sort sub-models by measured execution time
3. Distribute sub-models evenly among simulation threads

The performance of each sub-model is measured by simulating it single-threaded for a few time steps and measuring its average execution time. This is done before the actual simulation begins. Simulation threads keep track of the sub-models by using vectors with pointers to the sub-model objects, one for each sub-model type. Sub-models in each vector are then sorted from slowest to fastest based on their measured performance. Dividing a set of numbers into subsets with the smallest possible divergence is known as the partition problem:

Find a partition $S = \{S_1, S_2, \dots, S_n\}$ such that $\max(\text{sum}(S_1), \text{sum}(S_2), \dots, \text{sum}(S_n))$ is minimized.

Even though this problem is NP-complete, it is possible to find approximate solutions using heuristic methods. A straightforward approach is to use the *greedy algorithm* [92], where the unassigned sub-model with the longest execution time is iteratively assigned to the thread with the least total execution time. This algorithm results in a $4/3$ approximation of the optimal solution, and has a running time of $O(n^2)$.

A slightly better method is the quite clever *Karmarkar-Karp algorithm* [93]. This method consists of two steps, see Fig. 7.2. In the first step, the two largest elements in the set are replaced by their absolute difference, shown in grey. This is repeated until only one element remains. It is important to remember which two elements each difference element represents. In the second step, the subsets are generated by repeating this process backwards. The final difference element is added to one of the sets. It is then replaced by the two elements it represents. This is repeated until all the original elements have appeared in both subsets. Although this method can only be used for two subsets, it can in turn be used to split each subset into two new subsets. This method also results in a $4/3$ -approximation of the optimal solution and has the same running time as the greedy algorithm. However, it does have a higher chance of finding the optimal solution in many cases.

Figure 7.2 The Karmarkar-Karp algorithm can be used to solve the partition problem for two subsets.

7.3 Task Scheduling

Scheduling is the process of mapping a group of parallel tasks to the available parallel hardware. In this work, it is defined as assigning parallel tasks to parallel threads. The actual mapping between threads and CPUs is handled by the operating system.

A good scheduler should minimize the parallelization overhead. Three main sources of overhead are identified: *thread spawning*, *load imbalance* and *data contention*. Thread spawning overhead comes from the extra time required to create (and destroy) threads. Load imbalance overhead occurs if the load is unbalanced among the threads, so that some threads will have to wait for others. Finally, when several threads attempt to access a shared variable they will have to wait for each other. This is known as data contention. Overhead is categorized as either static or dynamic (independent of or dependent on number of threads) and constant or proportional (independent of or dependent on model size).

Distributed solver simulation can be either synchronous or asynchronous. In this thesis, only synchronous simulation is considered. Regardless of the method, it is an absolute necessity that threads never begin calculating a new time step before all required variables have been provided by its neighbouring sub-models. If this should happen, sub-models would work on outdated variables, calculating erroneous results. Four scheduling algorithms have been examined, with different methods for synchronization, load balancing and thread spawning, see Tab. 7.2.

Algorithm	Threads	Spawning	Dyn. Load Bal.
PSS	1/CPU	once	no
TS	1/CPU	once	yes
TP	unlimited	once	yes
FJ	unlimited	every time step	yes

Table 7.2 Four task scheduling algorithms, with different methods for load balancing and thread spawning, have been examined.

7.3.1 Pre-Simulation Scheduling

With pre-simulation scheduling (PSS), mapping between tasks and threads is computed prior to execution and thereafter remains constant. This method is analysed more in detail in paper [I]. The idea is to spawn one thread per CPU, and distribute the sub-models equally among the threads. Each thread is thus responsible for simulating one group of C-type sub-models and one group of Q-type. Barrier locks are used to synchronize the threads. This makes it possible to force asynchronous threads to wait for each other, and thereby act as if they were synchronous [94]. When a thread has finished execution of a certain type of sub-model, it arrives at the barrier. A busy-waiting lock ensures that no thread is allowed to continue past the barrier until all threads have arrived. There are several ways to implement such barriers. In this case, the first thread is used as *master thread* and the others as *slave threads*. When a slave thread arrives at a barrier, it increases an atomic counter variable. It then waits for the master thread to unlock the barrier. The master thread on the other hand waits for all slave threads to increase the counter before unlocking the barrier. This procedure is described in Fig. 7.3. There is a similar barrier for each sub-model type.

An advantage of this method is that threads only need to be created once and can be kept alive during the entire simulation. This minimizes overhead from thread spawning. A drawback is that it cannot correct load imbalance from sub-models with variable workload. Another problem with this kind of barrier implementation is the risk of dead-locks. These can occur if the number of threads exceeds the number of available CPUs. If one thread per CPU is already waiting at the barrier, the remaining threads cannot be executed, because there are no free CPUs. The simulation would then wait forever for all threads to arrive at the barrier. Normally, however, there will never be more threads than cores. This problem has therefore not been investigated further.

The code for the master thread, slave threads and barrier lock is shown in Appendix A. Operators are executed in parallel by using `tbb::task_group` templates. Barrier locks are implemented using `tbb::atomic` variables. As can be seen, the master thread must always lock the next barrier before unlocking the current one. This is to make sure that another thread does not accidentally arrive at the next barrier before the master thread has locked it.

Figure 7.3 A busy-waiting barrier lock is used to synchronize simulation threads.

7.3.2 Task-Stealing Scheduling

The pre-simulation scheduler can be improved to handle sub-models with varying workload. One such method is task-stealing scheduling (TSS). Threads that finish their tasks early can steal tasks from slower threads. In this way, load imbalance can be corrected dynamically during execution. The stealing in itself, however, may also induce additional overhead.

The algorithm is explained in detail in paper [II]. Each thread has two vectors, which contain finished and unfinished sub-models. When a sub-model has been executed, it is moved from the unfinished to the finished vector. After each iteration step, the vectors are swapped before the next iteration begins. Threads can steal and simulate sub-models from other threads using pointers to their vectors. When there is nothing left to steal, the iteration is complete and threads rendezvous at the barrier locks.

7.3.3 Task Pool Scheduling

Task pool scheduling (TPS) is commonly used for consumer-producer problems. In this case, there are no producers but a group of *worker threads* represent the consumers. Fig. 7.4 illustrates the algorithm. Sub-models are stored in pools, one for C-type and one for Q-type components. Threads pick and simulate one task at a time from the pools. Each pool also has a lock, which is used for synchronization. As soon as one pool is empty, the next pool is unlocked.

Sub-models in the pools are sorted in the same profile-based order as with the PSS algorithm. Hence, the threads measured to be slowest will be simulated

Figure 7.4 With the task pool algorithm, worker threads collect tasks from pools and execute them. The next Q-pool is unlocked once all C-tasks are finished and vice versa.

first. The order can vary, however, depending on the actual execution time of each model. This makes this method more robust to disturbances and variation in workload distribution. Another advantage is that the number of threads can be increased or decreased at any time during simulation.

7.3.4 Fork-Join Scheduling

Overhead due to load imbalance can be reduced by using fine-grained task parallelism. With fork-join scheduling (FJS), sub-models are executed in two parallel for-loops at each time step, see Fig. 7.5. Load balancing is handled by the internal work-stealing scheduler provided by TBB. Once all sub-models of one type have been executed, the simulation can continue with the next sub-model type. This method is very intuitive and explicit synchronization classes are not required. It is naturally robust against disturbances and variations in load distribution. The drawback is overhead due to excessive thread spawning since all threads must be re-spawned at each iteration step.

Figure 7.5 The fork-join scheduling algorithm uses the `parallel_for` template provided by TBB.

7.4 Performance Evaluation

The most natural way to quantify the benefits of parallel implementations is to compare the execution time of a parallel implementation with that of a sequential implementation. The ratio between sequential and parallel execution time is defined as the *parallelization speed-up*. Sequential time can be either the time of a purely sequential implementation or the single-threaded execution time of the parallel implementation. The ratio between execution time of a sequential and a parallel implementation is defined as *absolute speed-up*, see Eq. (7.1). The ratio between single-threaded and multi-threaded execution of the same parallel implementation is defined as the *relative speed-up*, see Eq. (7.2). Here, n denotes the problem size, p the number of processor cores, t_s the time with sequential implementation, and t_p the time with parallel implementation.

$$SU_{abs}(p, n) = \frac{t_s(n)}{t_p(p, n)} \quad (7.1)$$

$$SU_{rel}(p, n) = \frac{t_p(1, n)}{t_p(p, n)} \quad (7.2)$$

Another measurement is the ratio between speed-up and number of CPUs. This is called the *parallelization efficiency*, see Eq. (7.3).

$$EF(p, n) = \frac{SU_{rel}(p, n)}{p} \quad (7.3)$$

Due to overhead costs, a parallel implementation running on one processing unit is usually slower than its sequential counterpart. A measurement of this is the *parallelization slowdown*, defined as the ratio between single-threaded execution time of a parallel implementation and the execution time of a sequential implementation, see Eq. (7.4).

$$SL(n) = \frac{t_p(1, n)}{t_s(n)} \quad (7.4)$$

The PSS algorithm was analysed and evaluated in paper [I]. Benchmark models were constructed by using various numbers of synthetic sub-models with tunable workload. Three different computers were used, one quad-core, one six-core and one dual quad-core. Execution time for the parallel implementation with one, two, four, six or eight threads was compared with sequential execution time. For each number of threads, approximately 100 simulations were run with variable workload ranging from 0 to 60 kFLOP/iteration. Perfect load

balancing was assumed since unbalanced load is difficult to define. The fact that speed-up is reduced by load imbalance would also be an obvious and not so interesting result. Tab. 7.3 shows average parallel speed-up and efficiency for the three test computers.

	$SU_{abs}(p, n)$	$SU_{rel}(p, n)$	$EF(p, n)$
Linux, 4 cores, 2 threads	1.95	1.92	96.0%
Linux, 4 cores, 4 threads	3.51	3.46	86.4%
Windows, 6 cores, 2 threads	1.95	1.93	96.5%
Windows, 6 cores, 4 threads	3.61	3.56	89.0%
Windows, 6 cores, 6 threads	4.78	4.73	78.8%
Windows, 2x4 cores, 2 threads	1.95	1.94	97.0%
Windows, 2x4 cores, 4 threads	3.36	3.35	83.6%
Windows, 2x4 cores, 8 threads	5.42	5.39	67.4%

Table 7.3 Average speed-up and efficiency with pre-simulation scheduling on a synthetic model.

The different scheduling techniques were compared in paper [II]. Results show that FJS is better than the other methods for models with a large total workload. Fig. 7.6 shows simulation time against total model workload for FJS and TPS. Again, a synthetic model with perfect load balancing is assumed. FJS re-spawns all threads at each iteration step. Consequently, overhead costs for thread spawning, which is independent of model size, appear to dominate in the case of small models. On the other hand, load imbalance overhead becomes dominant in the case of large models, where FJS thus outperforms TPS.

Perhaps the most important aspect in applied research is the applicability in industrial use. Three example models, consisting of a hydro-mechanical rock drill, a digital hydraulic system and a mechanical pendulum, were used to compare the scheduling techniques. The first model is provided by Atlas Copco Rock Drills, and is actively used for product development. The second model is used in an ongoing research project [95]. A synthetic model with varying workload distribution was also examined. All models are simulated with 4 and 6 threads on a six-core processor. Results are shown in Tab. 7.4. From the results, it is clear that a larger number of CPUs is not always beneficial. In addition, some models benefit more from parallelism than others. TSS is the most successful method for evening out the load distribution in the varying workload model. On the other hand, PSS is the most efficient method for all the three real simulation models. This indicates that PSS has the lowest constant overhead, which makes sense considering that both scheduling and thread spawning are handled beforehand.

Figure 7.6 Overhead from thread-spawning appear to be dominant in the case of small models, while load imbalance overhead is dominant in the case of large models.

	Drill		Digital		Pendulum		Varying	
	<i>SU</i> (4)	<i>SU</i> (6)	<i>SU</i> (4)	<i>SU</i> (6)	<i>SU</i> (4)	<i>SU</i> (6)	<i>SU</i> (4)	<i>SU</i> (6)
PSS	4.10	3.18	2.11	3.69	2.17	2.00	1.96	2.82
TSS	3.84	2.34	2.11	3.69	1.17	1.39	3.28	4.85
TPS	2.70	1.96	1.35	2.45	1.94	1.84	2.25	3.11
FJS	2.19	1.31	3.07	2.19	1.06	0.85	2.42	3.33

Table 7.4 Speed-up for the different algorithms on four test models. It is clear that a larger number of CPUs is not always beneficial.

8

Multi-Domain Co-Simulation

Chapter 7 describes distributed-solver simulation, where different parts of a model are decoupled and solved independently. If the solvers are placed in different simulation tools, this is essentially equivalent to a co-simulation. As described before, decoupling induces a time delay in the interface variables. A common solution to this is to monitor the numerical error due to the delay and reduce the time steps until the error becomes acceptably small. An alternative solution is to use TLM elements, which replaces numerical errors with predictable and controllable modelling errors.

Exporting pre-compiled executable libraries is an efficient method for coupling of simulation tools. The slave tool generates source code, which is then compiled to a binary file containing subroutines required for running simulations. The binary can either communicate with the slave tool through network sockets or inter-process communication, or include the actual simulation code, known as *hosted simulation*. The host program can dynamically link against the library and call the subroutines. This method can be used not only for co-simulation, but also to run models in environments that are not supported by their original tool.

Another way of coupling two simulation tools together is to use a third program in between. This requires the third tool to be able to communicate with both of the tools that are to be coupled. A simulation environment with good compatibility with other software is Matlab/Simulink. The S-function interface [96] enables both C++ and Matlab code to be imported. By exporting models from two different programs to the same Simulink model, the models can be coupled even though their original programs are not compatible. For this reason, code generation for Simulink has been implemented in Hopsan.

The HopsanGenerator module exports C++ code, which includes the model, the HopsanCore source code and a wrapper for the S-function API. This code can subsequently be compiled into an S-function from Matlab and imported to Simulink. Examples of applications include connection between Hopsan and Adams [97] and connecting Hopsan with software for control system development [98].

8.1 Real-time Simulation through Library Export

Real-time applications put high demands on simulation performance. The simulation must always run with at least the same speed as in reality. This means that the execution time for performing a time step must never be longer than the time the step represents. Synchronization is normally handled by a real-time clock in the operating system. Requirements on real-time simulation can be hard, firm or soft. With hard real-time requirements, it is never acceptable to miss a deadline. Firm real-time allows occasional misses. With soft real-time, misses are allowed, but results will degrade after their deadline. It might thus be possible to discard a time step in the simulation if it should exceed its time limit. It may, however, reduce numerical robustness and is usually not a proper solution. Multi-core technology can be used to increase performance, allowing larger models to be simulated in real-time. On the other hand, parallelism can be harder to achieve in real-time systems. TLM is especially suitable for real-time applications [99]. Apart from the parallelization possibilities, the physically motivated time delays can also be useful for handling communication delays.

Figure 8.1 Models are downloaded to the target computer as dynamic-link libraries (.dll) and simulated in real-time.

8.1.1 Implementation

The applicability of multi-threading on real-time systems was tested on a hardware-in-the-loop rig, consisting of a host computer, a target computer, and a communication rack connecting the target computer to sensors and actuators on a hydraulic transmission test rig, see Fig. 8.1. Only the computers were used in the experiments, since simulation performance was the only measurement of interest.

The target computer has a quad-core processor and runs the Windows XP system with a LabVIEW [100] interface to perform the simulation work. Models must be pre-compiled into dynamic libraries (DLL) by using Microsoft Visual C++. A graphical interface called Viking [101] is used to download the libraries from the host system to connect them, to set parameters, and to control the actual simulation.

LabVIEW uses the VeriStand interface [102] to import dynamic libraries. The source code used to compile the DLL files must therefore provide subroutines according to the VeriStand API. One requirement is that internal and external data variables must be separated from each other, so that the simulation code never accesses the external variables directly. A name for the model and a base rate for simulation step size must also be provided.

There are three functions that can be modified: `USER_Initialize()`, `USER_TakeOneStep()` and `USER_Finalize()`. These are executed before the simulation begins, once every time step, and after the simulation has finished, respectively. `USER_TakeOneStep()` takes the external input/output data arrays and the current time value as arguments, see Appendix E.

An export routine in the HopsanGenerator module was created that generates this code directly from a Hopsan model. Connectivity components are used to define input and output variables, see Fig. 8.2. These can be either of signal input, signal output or power port type. Power ports will generate multiple input and output variables corresponding to the TLM equations.

Figure 8.2 Connectivity components, shown as rectangles with black arrows, must be used at the interface points of the sub-models.

All mapping and scheduling is done manually by assigning different Hopsan DLLs to different cores from the Viking graphical interface. The model is separated into sub-models and each sub-model is compiled into its own DLL file. Granularity is determined by the size of the sub-models. A coarse-grained separation will reduce the number of connections but may on the other hand cause

load imbalance. To ensure correct execution of the TLM equations, it is important that C- and Q-type components are executed separately. Fig. 8.3 shows different mapping and synchronization methods. The first two sub-figures show single-threaded simulations. It is possible to use either a smaller model and simulate the whole model on every time step (T_{RT}) or a larger model and simulate C-type and Q-type sub-models on every second step, respectively. With the first method, the execution of Q-type sub-models is delayed by half a time step. With the second method, each library only simulates itself on certain time steps, depending on its type. This was implemented by adding an if-statement in each `simulate()` function, which compares the current step number by using a modulus function. With this method, it is possible to map several components of different type to the same core, allowing good possibilities to tune the load balancing. The same principles can be used for multi-threading. Total model size, however, can be larger without increasing the step size.

Figure 8.3 Mapping and synchronization methods for single-threaded (a,b) and multi-threaded (c,d) real-time simulation. It is possible to either simulate the whole model on every time step or simulate each sub-model type on every second step.

8.1.2 Results and Analysis

In off-line simulations, parallel speed-up can be measured by adjusting model workload and measuring the resulting execution time. In real-time simulations, however, execution time is defined by the real-time clock, and will not depend on the model size. Speed-up can then instead be measured as *the maximum model size that can be simulated without losing real-time performance*. Each core on the real-time system reports if it misses an iteration step, that is, if the execution time was longer than the designated time frame. Experiments were performed by using components with tunable workload, similar to the ones described in Chapter 7. The workload was increased until at least one of the cores started reporting missed iterations.

Results showed that the system was quite unpredictable. In some measurements, the performance was drastically lower for no apparent reason. Some iteration steps were also always missed occasionally, regardless of model size. In most cases, however, the maximum model size was almost exactly four times larger with four threads than with one. Overhead costs from the communication wrapper functions were completely insignificant. The reason for this might be that only a few sub-models were used. Overhead costs are likely to have a larger impact for more fine-grained models. Coarse-grained models are, however, normally desirable to facilitate the setup work for the simulation. Numerical correctness was verified against a sequential simulation in paper [III].

8.2 Simulation of Active Suspension in Forwarder

System simulation is useful for simulating power-transmitting systems, for example hydraulic circuits, drive trains or electric circuits. While the interaction between energy storage, control units and actuators can be modelled with good accuracy, counteracting forces on actuators often pose a great challenge. Examples include actuators subjected to higher-order load dynamics or forces determined by complex trigonometry. One solution to this is to couple the system simulation tool with a multi-body simulation tool. The method is illustrated by an authentic test case. A hydraulic system for power transmission and active suspension in a forestry forwarder, shown in Fig. 8.4, is to be developed. The vehicle mechanics are modelled in Adams and the hydraulic system in Hopsan. Model validation, evaluation of system concepts and tuning of controller parameters in one program at a time is not practicable. Co-simulation is thus inevitable.

Figure 8.4 Co-simulation is used to develop a hydraulic system for power transmission and active suspension in a forestry forwarder.

8.2.1 Interface

FMI for co-simulation is used for tool coupling. Adams is used as the host application and Hopsan as the slave. The induced time delays of the interface variables are modelled as TLM elements. Hopsan sends characteristic impedances and wave variables to Adams, which in turn uses function elements to calculate forces. The TLM boundary equation (see Eq. (2.23)) will thus be inserted in the equation system in Adams. The remaining TLM computations are made in Hopsan. Fig. 8.5 shows the calculations for one TLM element. It is natural that the TLM element is placed in the program with the best support for TLM, in this case Hopsan. Calculations of F_2 from c_2 and v_2 must, however, be made in Adams in order to avoid incorrect time delays. The code exported from Hopsan is compiled using the MinGW compiler. The resulting executable library is linked with Adams from a wrapper compiled with Visual C++. Mixing different compilers works because FMI uses a plain C interface, which is compiler-independent [103].

Figure 8.5 *Calculations for a TLM element connecting Hopsan with Adams.*

8.2.2 Synchronization

The TLM element induces a time delay ΔT between Hopsan and Adams. In order to calculate the force at time T , Adams must have access to Hopsan variables from time $T - \Delta T$. This is ensured by forcing both programs to exchange data variables at a fixed communication interval. Requirements on the size of this interval are investigated in [52]. If both programs are able to produce data variables exactly at the communication points, no interpolation is required, as shown in Fig. 8.6a. In this case, the size of the communication interval is only limited by the time delay of the TLM element. Should one or both of the programs not be able to produce values at the specified interval,

however, interpolation is needed to obtain variables with the correct time delay. This is shown in Fig. 8.6b. To avoid extrapolation, the communication interval is then limited to half of the TLM time delay. In Hopsan, which uses fixed time steps, this is not a problem. Adams, however, uses solvers with variable step size. In these experiments no interpolation was used. Hence, time events were required in Adams in order to enforce calculations at the correct times. In practice, the step size in Adams was thus limited to the communication time delay used in the TLM element.

Figure 8.6 With synchronous communication, no interpolation of data variables is required.

The FMI interface could be improved to facilitate interpolation. If both programs keep their own, independent interpolation tables the interface can be used as it is. Variables must then be interpolated both before every simulation step, and before sending variables to the other program. In many cases, however, it is desirable to keep all interpolation in the host application. This makes it possible to use interpolation also in slave FMUs that do not support it. In order for this to work, the host application must be able to simulate the slave tool step by step with a priori knowledge about the expected step size. This requires methods for asking the slave tool about current step size and for simulating a single step without providing a stop time. In this way, the host tool will have full control of the simulation, which makes it possible to populate the interpolation table and interpolate variables for correct time instances.

8.2.3 Evaluation

Experiments confirm that the coupling works as intended. It was possible to evaluate the system concept with both tools working together. The TLM communication step of 0.1 ms significantly reduced computational performance in Adams during certain parts of the simulation. While Hopsan in general is faster than Adams, Adams compensates for this by using adaptive step size.

When possible, the step size is increased to improve performance. The co-simulation, however, enforces the small step size from Hopsan on Adams as well. A possible solution could be to let the TLM element use a larger time delay than the step size in Hopsan. However, increasing the delay induces parasitic inductance, which is difficult to compensate for in Adams.

It was possible to improve the design process by first computing an approximate solution using a simplified model in Hopsan. The resulting parameters can be used as input for the co-simulation, which is then used for fine-tuning, as shown in Fig. 8.7. This method requires significantly less time than using co-simulation during the whole design cycle.

Figure 8.7 Development time could be reduced by computing an approximate solution in Hopsan before fine tuning it using co-simulation with Adams.

9

Distributed Equation Solvers

Integrating TLM in an equation-based language enables distributed simulation with a structure that reflects the underlying equation system. Hence, equations and solvers can share the same object-oriented structure. Decoupling of equation systems into smaller parts provides two kinds of speed-up. First, the independent solvers enable parallel execution, as described in Chapter 7. Second, the computational cost for solving several smaller equation systems will be smaller compared to solving all equations together. The reason is that equation solving typically involves matrix inversion. The computational cost of inverting a matrix increases quadratically with its dimension, in this case the number of equations. Separating an equation system into smaller parts can thus provide speed-up even without parallel execution.

To use distributed solvers in a simulation tool, support for TLM elements with time delays must be implemented. Possibilities to use TLM elements for distributed simulation in existing Modelica tools were outlined in [IX]. Methods using `sample()` and `delay()` operators were tested according to Listings 9.1 and 9.2. It was also shown that derivatives could be approximated by using inlined solvers. This enables distributed solvers, which can use individual time steps.

```
when sample(-T,T) then
  left.c = pre(right.c) + 2 * Zc * pre(right.q);
  right.c = pre(left.c) + 2 * Zc * pre(left.q);
end when;
```

Listing 9.1 A TLM element in Modelica implemented using the `sample()` operator.

```
left.c = delay(right.c + 2 * Zc * right.q, T);
right.c = delay(left.c + 2 * Zc * left.q, T);
```

Listing 9.2 A TLM element in Modelica implemented using the `delay()` operator.

Distributed equation solvers can be implemented using three basic approaches:

1. Introducing TLM support in an existing Modelica tool

This method was explained further in [34]. The main advantage is that users can continue to use tools they are familiar with.

2. Introducing Modelica support in a tool with existing TLM support

Creating code-generators for TLM tools has been demonstrated by [23] and [XII]. A similar approach using Mathematica also exists [104]. With this method, the existing platform for distributed solvers can be used.

3. Exporting Modelica models using FMI

With the FMI standard, models from different tools can easily be connected and co-simulated. This makes it possible to use tool-independent distributed parallel simulation.

9.1 A Code Generation Approach

Code generation for distributed solvers from Modelica models was implemented in the Hopsan simulation tool. Distributed solvers make it possible to interconnect models from various sources such as Modelica, Mathematica, FMI or C++. The intention is to achieve seamless integration of sub-models generated from Modelica and components from other sources.

9.1.1 Code Generation

All models are translated to C++ code, which Hopsan can execute. Should several Modelica components be connected to each other, they will be grouped by merging their equations, see Fig. 9.1. In this way, the whole group can be translated to one single C++ component.

Figure 9.1 Independent groups of Modelica components are identified and compiled into C++ sub-models.

The TLM method requires certain variables to be interchanged. For this reason, the TLM boundary equations must be appended to the Modelica models. This can of course be done manually by the user. It is, however, possible to add the equations automatically during code generation. The procedure is demonstrated in Fig. 9.2. This eliminates the need to explicitly adapt models to TLM. The drawback is that the model is modified without user interaction.

Connecting models with TLM elements also requires hard-coded connector types. A TLM connection consists of intensity, flow, characteristic impedance and a wave variable. The two last can be added automatically by the code generator, similarly to the TLM boundary equations. An example of a TLM connector for the hydraulic domain is shown in Fig. 9.3.

```

model LaminarOrifice
  parameter Real Kc = 1e-11;
  NodeHydraulic P1, P2;
equation
  P2.q = Kc*(P1.p-P2.p);
  0 = P1.q + P2.q;
  P1.p = P1.c + P1.q*P1.Zc;
  P2.p = P2.c + P2.q*P2.Zc;
end LaminarOrifice;

```

Figure 9.2 TLM equations are automatically inserted into the Modelica code by the code generator.

```

connector NodeHydraulic "Hydraulic Node"
  Real      p "Pressure";
  flow Real  q "Volume flow";
  Real      c "Wave variable";
  Real      Zc "Char. impedance";
end NodeHydraulic;

```

Figure 9.3 Domain-specific connectors are used to interface Modelica models with TLM elements.

9.1.2 Modelica Solver

Equations are solved by a solver utility class, as shown in figure 9.4. First, the equations and variables are rearranged using a block lower triangular transformation (BLT). This minimizes the number of coupled equations while converting the rest to pure assignment operations. In this way, only part of the equations must be solved numerically. Available integration methods include explicit and implicit methods with step size ranging from 1 to 4. The algebraic equation system is solved by using LU decomposition. Hence, the solution x to the equation $Ax = b$ can be obtained without calculating the inverse of matrix A by replacing A with the product of a lower matrix L and an upper matrix U :

1. Perform LU-decomposition of A
 $Ax = b \Rightarrow LUx = b$
2. Solve $Ly = b$ for y
3. Solve $Ux = y$ for x

An alternative method based on inlining the derivatives using bilinear transform has also been implemented. This is quite similar to the trapezoidal rule

Figure 9.4 A solver utility class, which communicates with the sub-model, is used to solve the equations.

of integration. Equations are transformed into discrete form by replacing all Laplace operators with the z operator, according to equation 9.1.

$$F_d(z) = F_a(s) \Big|_{s = \frac{2}{T} \frac{z-1}{z+1}} = F_a \left(\frac{2}{T} \frac{z-1}{z+1} \right) \quad (9.1)$$

Finally, it is used that the z operator represents a pure time delay, and that the time derivative of a delayed variable is, by definition, zero. With this method, the integration of each derivative is inserted directly into the equations.

9.1.3 SymHop Symbolic Algebra Library

Numerical manipulation of equations requires a symbolic library. An important requirement is a fast execution speed in order to minimize overhead. A model where expressions are stored in tree structures was chosen. All expressions are represented by the **Expression** class. **Expression** objects can be of different types. Depending on the type, they will contain different sub-expressions. For instance, an addition expression will contain a list of terms. A power expression, on the other hand, will contain sub-expressions for base and exponent. In contrast to a binary tree, each expression can hold multiple sub-expressions. This facilitates symbolic manipulation such as simplification and differentiation. A SymHop tree for the expression $3X/(Y + Z + 5)$ is shown in Fig. 9.5.

Figure 9.5 SymHop tree describing the expression $3X/(Y + Z + 5)$.

9.2 A Co-Simulation Approach

An alternative to using code generation with TLM support or integrating TLM in an equation-based language is to use co-simulation. In this way, each sub-model can first be created in an arbitrary equation-based tool, and then connected to other sub-models in a TLM tool. It is thus possible to use different tools depending on requirements, preferences or knowledge. A suitable interface for connecting simulation tools is FMI, as described in Section 4.1.

Two simulation tools, Hopsan and OpenModelica, were used to verify the method. Sub-models were created in OpenModelica, where the TLM equations were explicitly inserted into the code. An example of a gearbox model is shown in Listing 9.3. The two last equations are the TLM boundary equations.

The OpenModelica Compiler (OMC) was used to compile the Modelica model into an FMU. Due to the distributed solver structure in Hopsan, FMI for co-simulation would have been the best choice, but was not yet supported by OMC. FMI for model exchange was therefore used. When importing the FMU to Hopsan, it was compiled into executable C++ code. This code includes a solver using the forward Euler integration method. In this way, the FMU will virtually work like an FMU for co-simulation when used in Hopsan. Imported FMUs become Q-type components which can be added to a Hopsan model and connected with TLM elements of C-type. This ensures the independence between components. Fig. 9.6 illustrates the process of importing several sub-models from FMUs.

Each TLM connection consists of multiple variables with different causalities. For simplicity, it is desirable to include information about mapping between variables and the TLM connections in the FMU. For this purpose, an additional tag was added to the XML description file, see Listing 9.4. Introducing this feature in the FMI standard would be a useful improvement.

The method is demonstrated by creating a model of a hydraulic linear position servo with a PI-controller, see Fig. 9.7. Sub-models of a pump, a relief valve, a control valve, a piston and a linear mass were created in OpenModelica. These

Listing 9.3 A mechanical gearbox modelled in Modelica. The last two equations are the TLM equations.

```

model OpenModelicaGearBox
  output Real T1(start = 0.001) "torque";
  output Real a1(start = 0) "angle";
  output Real w1(start = 0) "angular velocity";
  input Real c1(start = 0) "wave variable";
  input Real Zc1(start = 0) "characteristic impedance";
  output Real T2(start = 0.001) "torque";
  output Real a2(start = 0) "angle";
  output Real w2(start = 0) "angular velocity";
  input Real c2(start = 0) "wave variable";
  input Real Zc2(start = 0) "characteristic impedance";
  input Real r(start = 2) "ratio";
  parameter Real J(unit = "rad/s") = 0.1 "Inertia";
  parameter Real B(unit = "Nms/rad") = 10 "Damping";
equation
  J*der(w2) + B*w2 = T1*r - T2;
  der(a2) = w2;
  w1 = -w2*r;
  a1 = -a2*r;
  T1 = c1 + Zc1*w1;
  T2 = c2 + Zc2*w2;
end OpenModelicaGearBox;

```

Figure 9.6 The OpenModelica compiler is used to compile Modelica models into FMUs, which can then be imported by Hopsan.

were subsequently imported to Hopsan where they were connected by four TLM elements representing three volumes and a mechanical spring. Simulation time was reduced by 2.6 on a quad-core processor. As explained at the beginning of the chapter, speed-up may arise from both parallel execution and reduced matrix dimensions.

A benefit with the co-simulation method is that the distributed solvers become a natural part of the modelling process. Each tool can provide sub-models with different solvers and time steps. The complexity of the solver can then be adapted to the needs of each part of the system. Integration methods and

```
<?xml version="1.0" encoding="UTF-8"?>
<fmutilm>
  <tlmport type="hydraulicq">
    <q>q1</q>
    <p>p1</p>
    <c>c1</c>
    <Zc>Zc1</Zc>
  </tlmport>
  <tlmport type="hydraulicq">
    <q>q2</q>
    <p>p2</p>
    <c>c2</c>
    <Zc>Zc2</Zc>
  </tlmport>
  <tlmport type="mechanicq">
    <v>v3</v>
    <f>F3</f>
    <x>x3</x>
    <me>me3</me>
    <c>c3</c>
    <Zc>Zc3</Zc>
  </tlmport>
</fmutilm>
```

Listing 9.4 An additional XML tag with information about mapping between variables and TLM connections was added to the FMU.

time steps are thus no longer limited by the model's most sensitive parts. An interesting feature of FMI is the support for higher order interpolation of interface variables. This was not used in these experiments since the use of model exchange requires all solver algorithms to be placed in the same simulation tool. All communication was thus synchronized.

Figure 9.7 A model of a hydraulic linear position servo is used to demonstrate the method.

9.3 Modelling Techniques

An important question with TLM is where in the model to introduce the TLM elements. To avoid parasitic inductance, it is suitable to choose couplings with high flexibility (capacitance) and small inertia (inductance). One possibility is to replace all capacitances in the model with TLM elements. This, however, will limit the maximum size of the time step in each sub-model to the requirements from its stiffest neighbouring element. This could have an unacceptably large negative impact on simulation performance for the stiff elements. An element is defined as stiff if it requires an undesirably small time step to avoid parasitic inductance. Similarly, elements that can handle large time delays are defined as weak. Four different methods for introducing TLM in models have been identified:

- Use TLM only for simulation tool decoupling
- Use TLM only for the weak connections
- Use TLM everywhere except for the stiff connections
- Use TLM for all connections with distributed time step size

Using TLM for decoupling of simulation tools, i.e. inserting the TLM element at the interface between the tools, can enable distributed solvers for tools that do not support this internally. Another option is to decouple the model only at the weak connections. In this way, the model will be separated into a few, larger, sub-models, resulting in coarse-grained parallelism. This can, for example, be used to run each sub-model on its own processing unit. It is also possible to attack the problem from the opposite direction, by replacing all capacitances with TLM elements except the stiffest ones. This results in a fine-grained parallelism, with good possibilities for parallelization. By leaving the stiffest connections untouched, synchronous communication can be used while unacceptably low time steps can still be avoided. It is obviously possible to mix fine-grained and coarse-grained parallelism. If different physical domains are simulated together, one domain may be overall stiffer than the others. For example, mechanical systems, which are typically stiff, can be modelled coarse-grained with only a few sub-models. Hydraulic circuits on the other hand have larger capacitances and are more suitable for fine-grained modelling.

Finally, a more complicated option is to use asynchronous communication with distributed step sizes. Each sub-model can theoretically use its own, independent, step size. The only requirement is that it is equal to or smaller than half the communication time step used by the surrounding TLM elements. In this way, stiff TLM connections can be simulated with a smaller step size without affecting the rest of the model. On the other hand, interpolation errors might occur in the interface between sub-models with different step sizes. It is also possible to use variable step size in each sub-model. Asynchronous TLM communication with interpolation methods for co-simulation was discussed further in [52]. Experiment models that demonstrate the different approaches can be found in paper [V].

10

Parallel Distributed Optimization

Simulation-based optimization can be parallelized on different levels, hereinafter referred to as *multi-level parallelism*. It is possible to use a parallel algorithm but also decouple the simulation model into independent sub-models. Moreover, equation solvers in each sub-model might also support parallelism. Another option is to run several optimization jobs in parallel, which can increase both performance and the possibility of finding the global optimum. Multi-level parallelism in an optimization algorithm is illustrated by Fig. 10.1.

Figure 10.1 In simulation-based optimization, parallelism can be introduced at algorithm-level, model-level, sub-model-level and solver level.

Likewise, computer hardware can also support parallelism on different levels. This includes, for instance, parallel computers, parallel processors, multi-core processors and graphics processor units (GPU). In order to maximize the speed-up, a decent mapping between parallel code and parallel hardware is required. This is an NP-complete combinatorial problem, meaning that all possible combinations must be tested in order to guarantee finding the optimal solution. Furthermore, speed-up from parallelism at algorithm and model-level will not

only depend on the algorithm and model that are used, but also on the combination of the two.

Much work has been done on parallelizing population-based methods. Despite their faster convergence rate, direct-search methods have attracted much less attention. For this reason, a number of parallel versions of the Complex-RF algorithm have been developed. Their speed-up for different levels of parallelism has been measured. A PSO algorithm is also included for comparison. This data has in turn been used to implement and evaluate an online scheduler.

10.1 Parallel Implementations of Complex-RF

The Complex-RF algorithm evaluates one point at each iteration step and is thus of sequential nature. However, this does not necessarily prevent the use of parallelism. One parallelization approach is speculative multi-threading using task-prediction. By guessing tasks that may need to be performed in the future, these can be executed in advance on additional processing units. Then it is checked whether the guess was correct or not. Pre-calculated results can be used until an incorrect guess is encountered. Another method is to modify the algorithm to allow evaluation of more than one point at each iteration step. This, however, may affect the convergence rate and optimization results. Finally, an obvious method is to run several optimization jobs to solve the same problem in parallel. As soon as one of the jobs has converged, the others can be cancelled. This is useful when the convergence rate differs largely depending on the initial parameter distribution, which is often the case with direct-search methods.

10.1.1 Task-prediction

By predicting future reflections based on the current one, it is possible to evaluate multiple reflections in parallel. The concept is shown in Fig. 10.2. Points are numbered from worst (4) to best (1). The iteration begins with reflecting the worst point (x_4) in the normal way, see Fig. 10.2a. If we assume that the reflected point is no longer the worst, we know that x_3 will be the worst point at the next iteration. We can therefore calculate this reflection in parallel with the first reflection, as shown in Fig. 10.2b. In the same way, we can also predict reflections of x_2 and x_1 , see Figs. 10.2c and 10.2d. If an assumption should prove to be incorrect, the remaining predicted points are discarded. Technically, it is possible to predict an unlimited number of future evaluations. The chance of success for each additional prediction, however, will decrease due to the increased number of possible outcomes. An advantage with this method is that the original sequential algorithm in itself is not modified. While its speed is increased, all other properties remain unchanged.

Figure 10.2 By predicting future reflections, multiple points can be evaluated in parallel.

10.1.2 Multi-retraction

If a reflected point is still worse than all remaining points in the Complex, it must be retracted iteratively towards the centroid, with a pull towards the best point. This operation can easily be parallelized by evaluating several retraction steps simultaneously. Fig. 10.3 illustrates the method. After each parallel evaluation, results are examined, and the first point that is an improvement to the Complex is kept. In this way, the original algorithm is not modified and the outcome will be identical with a sequential execution. This method can be combined with any of the other methods mentioned in this chapter.

Figure 10.3 When retracting towards the centroid, multiple steps can be evaluated in parallel.

10.1.3 Multi-distance

One way of parallelizing an algorithm is to exploit its parametrization. The most critical parameter for Complex-RF is probably the reflection factor α . Changing the distance with which points are reflected can have a substantial impact on the convergence rate. Multi-distance parallelism is based on reflecting the worst point with multiple reflection factors at each iteration step and then keeping only the best result. Fig. 10.4 illustrates the method. Each processing unit will reflect the worst point (x_4) with a different distance. Reflection factors are evenly distributed between a minimum and a maximum distance. Two parametrizations were tested: only over-reflection ($1.0 < \alpha < 2.0$) and a combination of over- and under-reflection ($0.0 < \alpha < 2.0$). It is technically possible to also allow reversed reflection ($\alpha < 0.0$), but this contradicts the hill-climbing nature of the algorithm. Multi-distance parallelism will cause a modification of the original algorithm. Results are expected to be better for convex problems where the initial points are far from the optimum since the modification allows longer reflections.

Figure 10.4 The Complex-RF algorithm can be parallelized by reflecting with multiple distances.

10.1.4 Multi-direction

With non-linear problems, one can never be certain in which direction the optimum lies. Hill-climbing algorithms, such as the simplex and Complex methods, assume it to be in the direction between the worst point and the centroid of the others. Sometimes, however, searching in other directions can be more efficient. This can be used for parallelization. Instead of just reflecting the worst point, it is possible to reflect the p worst points, where p is the number of processing units. This method is shown in Fig. 10.5. Obviously, p must be smaller than or equal to the number of points. The most successful of the reflected points is kept for the next iteration. Like the multi-distance method, this also induces a modification of the original algorithm. It is expected to improve the convergence rate for non-linear problems in particular, since more directions are searched in each iteration step.

Figure 10.5 Parallelism can be introduced by reflecting multiple points simultaneously and keeping the best result.

10.1.5 Parallel Performance Evaluation

The parallelization methods were evaluated on three test models: the sphere function, the Rosenbrock function and a model representing a hydraulic position servo. All three models, as well as the parameters for the optimization algorithms, are presented in detail in paper [VII].

Each model was optimized with each algorithm at least 100 times using 2, 4, 8, 16 and 32 threads. The multi-distance, multi-retraction and (theoretically) task-prediction methods can make use of an unlimited number of threads. For the multi-direction method, however, the number of threads is limited to the number of search points. The maximum number of threads was thus limited to 8 in this case. The main reason for limiting experiments to 32 threads was that the intended platforms were desktop computers or small computer clusters. The number of search points is also limited by the number of parameters, which reduces the parallelizability. Furthermore, the usefulness of additional threads appears to decline as the number of threads increases.

In addition to the methods mentioned above, two other methods were tested. The first is to run multiple optimizations of the same problem in parallel and stop when the first optimization converges. The second is a combination of task-prediction and multi-retraction. A few threads are used to evaluate predicted points while the rest pre-evaluate possible retraction points.

Tab. 10.1 shows the measured speed-up for each method. Speed-up is defined as the ratio between average number of evaluations with sequential execution and number of equivalent parallel evaluations with multi-threading.

Threads		2	4	8	16	32
Algorithm						
Sphere	Task-Prediction	1.19	1.28	1.30	1.31	1.30
	Multi-retraction	1.56	1.71	1.72	1.69	1.71
	Combination	1.74	2.33	2.58	2.61	2.70
	Multi-direction	1.22	1.49	1.45	-	-
	Multi-distance _{1→2}	0.98	1.11	1.23	1.34	1.39
	Multi-distance _{0→2}	2.25	3.46	4.16	4.70	5.25
	Parallel jobs	1.04	1.09	1.14	1.19	1.24
Rosenbrock	Task-Prediction	1.16	1.34	1.33	1.39	1.39
	Multi-retraction	1.43	1.71	1.77	1.76	1.76
	Combination	1.50	2.14	2.46	2.59	2.66
	Multi-direction	1.54	2.01	1.78	-	-
	Multi-distance _{1→2}	1.07	1.16	1.27	1.33	1.40
	Multi-distance _{0→2}	2.33	2.93	3.28	3.70	4.04
	Parallel jobs	1.82	2.32	2.89	3.15	3.15
Servo	Task-Prediction	1.07	1.21	1.24	1.25	1.24
	Multi-retraction	1.43	1.74	1.75	1.78	1.82
	Combination	0.99	1.31	1.74	1.74	2.25
	Multi-direction	1.25	1.40	1.24	-	-
	Multi-distance _{1→2}	0.96	1.02	0.91	1.06	1.24
	Multi-distance _{0→2}	1.70	2.26	2.62	2.70	2.61
	Parallel jobs	1.17	1.36	1.47	1.61	1.69

Table 10.1 Speed-up of the parallel Complex-RF algorithms for each model with different numbers of threads.

Multi-distance with both over- and under-reflection gives the highest speed-up for all models. Multi-distance with only over-reflection, however, is hardly more efficient than the sequential algorithm. Running parallel jobs is a competitive alternative for the Rosenbrock model, but not for the sphere function. The method can be expected to give more speed-up for non-linear models, where the number of iterations varies greatly depending on the initial distributions

of the point. For simple models like the sphere, the initial distribution is likely to have a smaller impact. With the multi-direction method, speed-up appears to decline when using more than four threads. This indicates that searching in the wrong direction can reduce the overall efficiency, even if the direction provides better results at a certain reflection. The multi-retraction method gives substantial speed-up for all models, but the benefits seem to level off after four threads. This indicates that more than four retractions are seldom necessary. The combination of task-prediction and multi-retraction performs well for all models. However, efficiency still declines as the number of threads increases.

The objective of optimization is to gather information about the optimal solution to a given problem. The efficiency of an algorithm can thus be expressed using information theory. As the search points converge the entropy for each parameter will increase, according to (10.1). Δx is the uncertainty of the parameter and x_R its feasible region in the design space. Entropy rate based performance index (ERI) is defined as the average number of iterations required to reach a specified entropy and the probability of finding a local optima (P_{opt}) according to (10.2) [105].

$$H_x = -\log_2 \frac{\Delta x}{x_R} \quad (10.1)$$

$$\phi^{(2)} = \frac{H_x}{N_m} = \frac{1}{N_m} \left((1 - P_{opt}) \log_2 \left(\frac{1 - P_{opt}}{1 - \varepsilon_x^n} \right) + P_{opt} \log_2 \left(\frac{P_{opt}}{\varepsilon_x^n} \right) \right) \quad (10.2)$$

The parallel implementations of Complex-RF were compared with the original sequential algorithm. For comparison, a PSO algorithm with two parametrizations was also included. The latter is interesting because population-based methods are naturally parallel up to the population size. At least 100 optimizations were run with each algorithm on the sphere and Rosenbrock models. A more thorough description of the experiment is included in paper [VII]. Results are shown in Tab. 10.2. The multi-distance method with $0.0 < \alpha < 2.0$ is the best method in all cases. The advantage over PSO is however much larger for the sphere than for the Rosenbrock model. Another interesting observation is that the multi-distance method is more than twice as efficient with two threads compared to the sequential algorithm. This means that it can be beneficial to use the parallel method even if it is executed sequentially in only one thread.

		Threads	1	2	4	8	16	32
Sphere	Algorithm							
	Complex-RF		0.100	-	-	-	-	-
	Task-Prediction		-	0.121	0.129	0.130	0.133	0.131
	Multi-retraction		-	0.156	0.168	0.170	0.169	0.171
	Combination		-	0.155	0.167	0.201	0.262	0.267
	Multi-direction		-	0.183	0.247	0.265	-	-
	Multi-distance _{1→2}		-	0.109	0.122	0.136	0.148	0.158
	Multi-distance _{0→2}		-	0.294	0.409	0.500	0.535	0.622
	Parallel jobs		-	0.104	0.110	0.115	0.120	0.125
	PSO ₁₀₀₀		-	0.010	0.019	0.038	0.076	0.153
	PSO ₂₀₀₀		-	0.007	0.014	0.028	0.056	0.113
Rosenbrock	Complex-RF		0.028	-	-	-	-	-
	Task-Prediction		-	0.035	0.040	0.044	0.040	0.041
	Multi-retraction		-	0.044	0.053	0.051	0.051	0.052
	Combination		-	0.043	0.065	0.074	0.089	0.103
	Multi-direction		-	0.030	0.036	0.036	-	-
	Multi-distance _{1→2}		-	0.029	0.038	0.038	0.040	0.045
	Multi-distance _{0→2}		-	0.053	0.073	0.086	0.094	0.108
	Parallel jobs		-	0.074	0.102	0.129	0.141	0.141
	PSO ₁₀₀₀		-	0.006	0.012	0.024	0.047	0.094
	PSO ₂₀₀₀		-	0.004	0.008	0.017	0.033	0.066

Table 10.2 Performance indexes for the parallel Complex-RF algorithms and the PSO method with different numbers of threads.

10.2 Work Scheduling for Distributed Optimization

With simulation-based optimization, both the algorithm and the model can be parallelized. For this reason, overall speed-up will depend on both of these. The total time for an optimization (t_{opt}) can be computed as the number of equivalent evaluations with parallelism ($n_{eval,e}$) multiplied by the longest simulation time for any of the parallel evaluations, see Eq. (10.3) [VIII]. This is because all processors must stop and synchronize at each iteration step. In asynchronous optimization, the average simulation time could be used instead.

$$t_{opt} = n_{eval,e} \max_{0 \leq i < n} \mathbf{t}_{m,i} + t_{comm} \quad (10.3)$$

Speed-up of the algorithm (SU_{alg}) is defined as the ratio between the number of evaluations with sequential execution and the equivalent number of parallel

evaluations, see Eq. (10.4). Speed-up will depend on the degree of parallelism (DOP) of the algorithm.

$$n_{eval,e} = n_{eval}SU_a(p_a) \quad (10.4)$$

The execution time for each simulation (t_m) is in turn a function of the sequential simulation time ($t_{m,0}$) multiplied by the model speed-up (SU_m), as shown in Eq. (10.5).

$$t_m = t_{m,0}SU_m(p_m) \quad (10.5)$$

Network optimization makes it possible to utilize unused computers within an organization. It can also be used with dedicated simulation clusters. If the available computers have different performance and numbers of cores, the work must be distributed economically. The solution is to maximize the product of SU_a and SU_m depending on the DOPs for the algorithm and the model. This can be done either manually or automatically. An experimental framework was developed where the simulation tool connects to a master server which manages a number of simulation servers, see Fig. 10.6.

Figure 10.6 A framework for network optimization with efficient scheduling of multi-level parallelism was developed.

The master server monitors performance, reliability and number of available processing units in each simulation server. This information is then used to

compute decent work scheduling based on the expected speed-up of the algorithm and the model for different DOPs. In most cases, there is no use in parallelizing a model across several computers due to communication overhead. Simulation times for each evaluation are measured during the optimization. This is used to identify disturbances and re-schedule to maintain decent load balancing.

Part IV

Discussion and Conclusions

11

Discussion

The importance of adapting simulation software for parallel computing can be expected to increase in the future. Due to the power wall, development is being forced towards larger numbers of processing units. An important challenge will be to combine data parallelism, task parallelism and job parallelism with available resources such as SIMD hardware like GPU cards, MIMD hardware like multi-core processors, and homogeneous or heterogeneous computer networks.

When decoupling simulation models, inserting time delays is inevitable. Luckily, almost every physical element has a natural time delay. The relationship between intensity and flow can be described by a capacitance and an inductance, regardless of the physical domain. Examples include flexibility and inertia for mechanics and compressibility and fluid inertia for fluids. These in turn give rise to a time delay, corresponding to the wave propagation speed. When applying an intensity (force, voltage, pressure, etc.) at one side of the element, it takes some time before a change can be detected at the other side. By including such phenomena in the model, elements become naturally decoupled. Several methods for parallel simulation have previously been discussed [6][7][8][2][1]. These methods either require explicit language constructs, manual modifications to equations, or rely on data parallelism. This thesis presents an automatic method based on explicit placement of TLM elements. The most important advantage of the proposed method is probably that it minimizes additional work on the part of the user. By treating TLM elements as regular model components, the decoupling becomes a natural part of the modelling process, while the parallelization is still transparent to the user. A related method using automatic placement of TLM elements was demonstrated by Sjölund et al [34]. This approach can be used for existing Modelica models but is not transparent and does not enforce a connection to the physical world.

Results show that different scheduling algorithms are suitable for different test models. The reason is probably that the properties of the model affect the

impact from different kinds of overhead. Overhead is expected to arise mainly from thread spawning, busy-waiting synchronization and load imbalance. For models with a large total workload, thread spawning overhead seems to become insignificant. Load imbalance overhead, however, can be expected to increase proportionally with the total workload. Schedulers that re-use the same threads for the whole simulation are thus more effective below a certain total workload. In contrast, schedulers using implicit synchronization by re-spawning threads require no busy-waiting locks and perform well for larger models. It could be possible to choose the most efficient algorithm automatically by analysing the model. This would involve benchmarking the model and its sub-models to measure total workload and load distribution. However, it becomes complicated if sub-models have varying workload. A simpler approach could be to benchmark each scheduler and see which one performs best for the model at hand.

Sub-models with distributed solvers are very similar to co-simulation, where different simulation tools solve different parts of the whole model. Splitting a model over several simulation tools introduces a time delay at the interface variables. To avoid numerical errors, this delay must be controlled. The state-of-the-art solution is to limit the error by reducing the step size. With TLM, it is possible to maintain a larger step size. Connecting simulation tools with TLM, however, requires both programs to have compatible TLM interfaces.

An increasingly popular modelling paradigm is object-oriented equation-based languages. A natural approach for these is to use a solver structure that mirrors the mathematical structure of the model. The state-of-the-art solution, however, is to flatten the object-oriented model into one large equation system and use only one centralized solver. This is necessary due to tight coupling between equations, resulting from the lack of decoupling in the time domain. By introducing TLM elements, distributed solvers reflecting the structure of the model can be used. This provides speed-up from both parallelism and smaller equation system sizes as well as increased robustness and scalability.

A key issue for parallel simulation is whether the parallelism should be transparent to the user, or if it should be automated with no user interaction. Transparent parallelism requires more understanding on the part of the user. On the other hand, automatic parallelism leaves the user with no control or knowledge. An important factor is parasitic inductance, which can affect simulation results. Should this become too large, it is necessary to compensate for it. This can be done either by reducing neighbouring inertia or by reducing simulation time step. Transparent parallelism requires the user to do this. With automatic parallelism, on the contrary, it might be difficult to automatically identify the variables to be used for compensation.

Another important question is where to insert TLM elements in the model. One option is to replace only weak couplings with TLM elements. This gives a coarse-grained distribution and is a logical approach for co-simulation. It can also be suitable for a logical partitioning of models consisting of larger groups

of stiff equations. The opposite method is to introduce TLM element at all connections except the stiffest ones, which gives a fine-grained distribution. This is generally more suitable for models with a large number of distributed capacitances and for models where TLM has been used from the beginning of the modelling process. Different methods can thus be suitable for different physical domains. For example, fluid circuits in general have much larger capacitances than mechanical systems. If stiff connections are replaced with TLM elements, it might be necessary to reduce the step size to avoid parasitic inductance. There is thus a trade-off between parallelism and step size.

Simulation-based product development requires not only high-performance simulation but also efficient development tools. One such example is design optimization. Unfortunately, its usefulness is often limited by long execution times. Meanwhile, the utilization rate of computer resources is very low in many organizations. Exploiting both parallel computers and multi-core processors can greatly improve optimization performance. If both the model and the optimization algorithm can be parallelized, speed-up at multiple levels can be achieved. It is shown that naturally sequential optimization algorithms also can be re-written for parallel execution. Since population-based methods are naturally parallel, they should be more suited for taking advantage of parallel hardware. As the number of processing units increase, the importance of scalability will also increase. While this seems to promote population-based methods it must, however, be pointed out that direct-search algorithms normally require fewer iterations to converge. Hence, direct-search methods may outperform their population-based counterparts even at a high degree of parallelism.

Another useful property of optimization algorithms is predictability. The number of iterations can vary greatly depending on, for instance, the simulation model, initial parameter distribution and algorithm parameterization. For example, the multi-direction version of Complex-RF performs very well for two test models but is worse for the third. This cannot be known in advance. Hence, a slower but more predictable method may be preferable over a generally faster one. Experiments have shown that reflecting multiple distances with both under- and over-reflection provides the highest speed-up of the tested parallel methods for Complex-RF. On one test model, it even achieved super-linear speed-up on two processors. It can be expected that different methods give optimal results at different stages of an optimization. A possible improvement could therefore be to change parallel method during the optimization. At the early stage, finding a good seed (exploration) should be favoured for generating a good seed and avoiding premature convergence. Later on, improving a local optimum (exploitation) can be favoured to speed up the convergence rate. Long reflections are also probably more common in the beginning, while contractions are more common towards the end. This could be exploited by using for example multi-distance in the early phase and changing to multi-iteration after some time, when the entropy has increased.

12

Conclusions

Distributed solvers enable parallel execution of simulation models, which can greatly reduce simulation time. This can shorten the development cycle and reduce the need for costly physical prototyping. The distributed approach can also increase modularity and facilitate collaborative product development.

An automatic algorithm for sorting, scheduling and synchronization of distributed-solver models has been developed. This enables user-controlled parallel simulation with minimal input. Experiments show that the optimal scheduling method is determined by model size and load distribution. For large models, speed-up is almost proportional to the number of processing units. For small models, slow-down due to overhead costs occurs. Model size where speed-up exceeds overhead costs depends largely on the processor and number of threads but has an order of magnitude of 1 double-precision kFLOP/iteration.

TLM is a suitable method for co-simulation, since it eliminates numerical errors from the communication delay of the interface variables. Experiments confirm that the FMI standard can effectively be used with a TLM interface. The support for higher-order interpolation makes it possible to simulate sub-models asynchronously with distributed time steps. Further improvements of the interface are, however, required to make this useful in practice. It was also concluded that connecting a fast program with small step size with a slower program with larger step size can have a major negative impact on simulation performance.

TLM elements can be used to decouple equation-based Modelica models. This can be implemented either by introducing TLM in an existing Modelica compiler, or by introducing code generation from Modelica to an existing TLM compatible software. The first method requires less adaptation by existing users. The second method, on the other hand, makes it possible to mix models

from different Modelica tools both with each other and with models from other simulation domains. For this the FMI standard can be used, which increases compatibility, modularity and re-usability of the models.

Six parallel variants of the Complex-RF algorithm have been implemented. Most of them provide acceptable speed-up for a few threads, but the speed-up levels out as the number of processing units increases. A combination of methods has also been demonstrated. Optimization algorithms were compared by using an entropy rate based performance index. It could be shown that the best parallel version of Complex-RF always outperforms a population-based algorithm when using up to 32 processing units. If both the model and the optimization algorithm can be parallelized, an optimal distribution over the available hardware must be chosen. It is shown that total speed-up for the optimization job can be estimated from the model-level and algorithm-level speed-up for different numbers of processing units. An algorithm for multi-level scheduling on homogeneous computer grids has been developed. Experiments show that it is able to compute pseudo-optimal distributions for network optimization. Theoretically, it can provide acceptable results even for heterogeneous grids. For naturally parallel optimization algorithms, there is no advantage in parallelizing the model as long as the number of processing units does not exceed the number of search points.

13

Outlook

This thesis demonstrates several benefits of distributed simulations. Experiments are, however, limited to specific software and certain simulation domains. To expand the results to a larger user base, the methods need to be generalized. For this reason, a general framework for distributed simulation should be developed. Re-usability, compatibility and modularity can be ensured by importing sub-models using FMI. Numerical errors can be eliminated by connecting sub-models with TLM elements. Finally, correct values for both capacitance and inductance in each TLM element can be achieved by using asynchronous communication. Both numerical errors and modelling errors can thus be avoided. Each TLM element can have its own time delay and each sub-model can be solved with an individual step size and integration method. This requires interpolation of interface variables. FMI supports higher-order interpolation methods by providing access to derivatives of variables. With minor additions, this could allow each model to use its own, independent, step size. A framework that allows asynchronous simulation with support for FMI and TLM would enable completely tool-independent simulations. Possibilities to improve support for TLM in the FMI standard should be investigated. Another interesting feature would be to support multi-level scheduling of parallel simulations and simulation jobs, which can be used for optimization or sensitivity analysis.

In recent years, population-based optimization algorithms such as PSO and GA have increased in popularity. While direct-search methods may be more difficult to parallelize, they have a fast convergence rate and a sufficiently good chance of finding the global optimum for many problems. Combining different types of algorithms could provide a better trade-off between exploration and exploitation. A more thorough investigation and comparison of parallel optimization methods, for example by using entropy rate based performance index, would be interesting.

14

Review of Papers

Paper I

This article gives an in-depth analysis of the implementation of multi-threaded simulation in Hopsan. The process of decoupling equations using the transmission line element method is mathematically derived. Methods for partitioning models with decent load balancing are discussed. Simulation threads are synchronized by using a busy-waiting barrier lock. Speed-up is measured by running a large number of benchmarking simulations for different numbers of processors, with different model sizes and on different operating systems. Results show that speed-up is close to linear but is reduced as the number of processor cores increases.

Paper II

Different task scheduling techniques for multi-threaded distributed solver simulations are described and evaluated. In particular, the influence on performance from task granularity and dynamic load balancing is studied. Overhead from load imbalance, thread spawning and synchronization is discussed. It is concluded that fine-grained scheduling provides good speed-up for models with a large total workload while coarse-grained scheduling performs better on small models.

Paper III

An approach for running multi-core simulations in real-time applications is presented. Models are manually divided into sub-models, which in turn are compiled into shared libraries (DLLs). A graphical interface based on LabVIEW is used to load and connect these libraries in the real time system. Different approaches for how to synchronize C-type and Q-type components are dis-

cussed. Numerical correctness is verified by comparing the parallel model with an equivalent sequential model.

Paper IV

This paper is part of a research collaboration with the forest industry. A new system for active damping of forest forwarders is to be developed. Hydraulic systems for transmission and damping are modelled in Hopsan. A full vehicle model has previously been created in Adams. Co-simulation makes it possible to investigate the feasibility of the method and tune key parameters.

Paper V

Support for the Modelica language has been implemented in Hopsan. Modelica models can be interconnected with both regular Hopsan components and FMUs with TLM elements. The Modelica code is translated to Hopsan-compatible C++ code, which in turn is compiled into component libraries. TLM equations are automatically inserted in the Modelica code. The method is verified with example models and different modelling techniques are discussed.

Paper VI

FMI for co-simulation is implemented in Hopsan. FMUs from other tools can be imported and connected with TLM elements. This results in a co-simulation platform where models from different tools can be simulated together. Simulation performance can be increased by using the multi-threading support in Hopsan. The method is verified on an example model with FMUs from OpenModelica and the parallel performance is analysed.

Paper VII

Five parallelization methods for the originally sequential Complex-RF optimization algorithm are described and implemented. Possibilities for parallelism is identified and analysed. Speed-up for each method is measured on two test function and one real simulation model. The methods are also compared to a particle-swarm algorithm by using entropy rate based performance index.

Paper VIII

An algorithm for scheduling of network optimization with support for multi-level parallelism has been developed. The algorithm supports homogeneous and heterogeneous computer clusters, and takes both model-level and algorithm-level parallelism into account. Dynamic load balancing during execution is also supported. Online profiling is used to monitor the performance of the computers. Some parallel implementations of the Complex-RF algorithm have been developed in order to evaluate the method. These are compared with a PSO algorithm, which is inherently parallel.

A

Pre-Simulation Scheduler Code

```
class BarrierLock {
public:
    BarrierLock(size_t nThreads) {
        mnThreads = nThreads;
        mCounter = 0;
        mLock = true;
    }
    inline void lock() { mCounter=0; mLock=true; }
    inline void unlock() { mLock=false; }
    inline bool isLocked() { return mLock; }
    inline void increment() { ++mCounter; }
    inline bool allArrived() { return (mCounter == (mnThreads-1)); }
private:
    int mnThreads;
    tbb::atomic<int> mCounter;
    tbb::atomic<bool> mLock;
};
```

Listing A.1 *The barrier lock is implemented as a class with an atomic counter, an atomic lock variable, and some access functions.*

```

void operator() () {
    while(mTime < mStopTime) {
        while(!mpBarrier_S->allArrived()) {} //S-components
        mpBarrier_C->lock();
        mpBarrier_S->unlock();
        for(size_t i=0; i<mVectorS.size(); ++i) {
            mVectorS[i]->simulate(mTime, mTime+mTimeStep);
        }
        while(!mpBarrier_C->allArrived()) {} //C-components
        mpBarrier_Q->lock();
        mpBarrier_C->unlock();
        for(size_t i=0; i<mVectorC.size(); ++i) {
            mVectorC[i]->simulate(mTime, mTime+mTimeStep);
        }
        while(!mpBarrier_Q->allArrived()) {} //Q-components
        mpBarrier_N->lock();
        mpBarrier_Q->unlock();
        for(size_t i=0; i<mVectorQ.size(); ++i) {
            mVectorQ[i]->simulate(mTime, mTime+mTimeStep);
        }
        while(!mpBarrier_N->allArrived()) {} //Log data variables
        mpBarrier_S->lock();
        mpBarrier_N->unlock();
        logDataVariables();
        mTime += mTimeStep;
    }
}

```

Listing A.2 Simulation code for the master simulation thread.

```

void operator() () {
    while(mTime < mStopTime) {
        mpBarrier_S->increment();
        while(mpBarrier_S->isLocked()){} //S-components
        for(size_t i=0; i<mVectorS.size(); ++i) {
            mVectorS[i]->simulate(mTime, mTime+mTimeStep);
        }
        mpBarrier_C->increment();
        while(mpBarrier_C->isLocked()){} //C-components
        for(size_t i=0; i<mVectorC.size(); ++i) {
            mVectorC[i]->simulate(mTime, mTime+mTimeStep);
        }
        mpBarrier_Q->increment();
        while(mpBarrier_Q->isLocked()){} //Q-components
        for(size_t i=0; i<mVectorQ.size(); ++i) {
            mVectorQ[i]->simulate(mTime, mTime+mTimeStep);
        }
        mpBarrier_N->increment();
        while(mpBarrier_N->isLocked()){} //Log data variables
        for(size_t i=0; i<mVectorN.size(); ++i) {
            mVectorN[i]->logData(mTime);
        }
        mTime += mTimeStep;
    }
}

```

Listing A.3 Simulation code for the slave simulation threads.

B

Task Stealing Scheduler Code

```

void operator() () {
    ThreadSafeVector *pTemp;
    Component *pComp;
    while(mTime < mStopTime) {
        while(!mpBarrier_S->allArrived()) {} //S-components
        mpBarrier_C->lock();
        mpBarrier_S->unlock();
        for(size_t i=0; i<mVectorS.size(); ++i) {
            mVectorS[i]->simulate(mTime);
        }
        while(!mpBarrier_C->allArrived()) {} //C-components
        mpBarrier_Q->lock();
        mpBarrier_C->unlock();
        //Switch Q-vectors
        pTemp = mpUnFinishedQ->at(mThreadID);
        mpUnFinishedQ->at(mThreadID) = mpFinishedVectorQ;
        mpFinishedVectorQ = pTemp;
        //Simulate own C-components
        pComp = mpUnFinishedC->at(mThreadID)->tryAndTakeFirst();
        while(pComp) {
            pComp->simulate(mTime);
            mpFinishedVectorC->insertFirst(pComp);
            pComp = mpUnFinishedC->at(mThreadID)->tryAndTakeFirst();
        }
        //Steal components
        for(size_t i=0; i<mnThreads-1; ++i) {
            int j = (mThreadID+1+i)%mnThreads;
            pComp = mpUnFinishedC->at(j)->tryAndTakeLast();
            if(pComp) {
                pComp->simulate(mTime);
                mpFinishedVectorC->insertLast(pComp);
                break;
            }
        }
        while(!mpBarrier_Q->allArrived()) {} //Q-components
        mpBarrier_N->lock();
        mpBarrier_Q->unlock();
        //Switch C-vectors, simulate and steal Q-components
        [...]
        while(!mpBarrier_N->allArrived()) {} //Log data
        mpBarrier_S->lock();
        mpBarrier_N->unlock();
        logDataVariables();
        mTime += mTimeStep;
    }
}

```

Listing B.1 *Simulation code for the master simulation thread.*


```

void operator() () {
    ThreadSafeVector *pTemp;
    Component *pComp;
    while(mTime < mStopTime) {
        mpBarrier_S->increment();
        while(mpBarrier_S->isLocked()){} //S-components
        mpBarrier_C->increment();
        while(mpBarrier_C->isLocked()){} //C-components
        //Switch Q vectors
        pTemp = mpUnFinishedQ->at(mThreadID);
        mpUnFinishedQ->at(mThreadID) = mpFinishedVectorQ;
        mpFinishedVectorQ = pTemp;
        //Simulate own components
        pComp = mpUnFinishedC->at(mThreadID)->tryAndTakeFirst();
        while(pComp) {
            pComp->simulate(mTime);
            mpFinishedVectorC->insertFirst(pComp);
            pComp = mpUnFinishedC->at(mThreadID)->tryAndTakeFirst();
        }
        //Steal components
        for(size_t i=0; i<mnThreads-1; ++i) {
            int j = (mThreadID+1+i)%mnThreads;
            pComp = mpUnFinishedC->at(j)->tryAndTakeLast();
            if(pComp) {
                pComp->simulate(mTime);
                mpFinishedVectorC->insertLast(pComp);
                break;
            }
        }
        mpBarrier_Q->increment();
        while(mpBarrier_Q->isLocked()){} //Q-components
        //Switch C-vectors, simulate and steal Q-components
        [...]
        mpBarrier_N->increment();
        while(mpBarrier_N->isLocked()){} //Log data variables
        mTime += mTimeStep;
    }
}

```

Listing B.2 *Simulation code for the slave simulation threads.*

C

Task Pool Scheduler Code

```
void operator() () {
    Component *pComp;
    while(mTime < mStopTime) {
        if (mpTaskPoolC->isOpen()) { //C-components
            pComp = mpTaskPoolC->getComponent();
            while(pComp) {
                pComp->simulate(*mpTime);
                mpTaskPoolC->reportDone();
                pComp = mpTaskPoolC->getComponent();
            }
            while(mpTaskPoolC->isOpen()) {}
        }
        if (mpTaskPoolQ->isOpen()) { //Q-components
            pComp = mpTaskPoolQ->getComponent();
            while(pComp) {
                pComp->simulate(*mpTime);
                mpTaskPoolQ->reportDone();
                pComp = mpTaskPoolQ->getComponent();
            }
            while(mpTaskPoolQ->isOpen()) {}
        }
        mTime += mTimeStep;
    }
}
```

Listing C.1 *Simulation code for the slave simulation threads.*

```
for (size_t i=0; i<nSteps; ++i) {
    pTaskPoolS->open(); //S-components
    pComp = pTaskPoolS->getComponent();
    while (pComp) {
        pComp->simulate (mTime+mTimeStep);
        pTaskPoolS->reportDone();
        pComp = pTaskPoolS->getComponent();
    }
    while (!pTaskPoolS->isReady()) {}
    pTaskPoolS->close();
    pTaskPoolC->open(); //C-components
    pComp = pTaskPoolC->getComponent();
    while (pComp) {
        pComp->simulate (mTime+mTimeStep);
        pTaskPoolC->reportDone();
        pComp = pTaskPoolC->getComponent();
    }
    while (!pTaskPoolC->isReady()) {}
    pTaskPoolC->close();
    pTaskPoolQ->open(); //Q-components
    pComp = pTaskPoolQ->getComponent();
    while (pComp) {
        pComp->simulate (mTime+mTimeStep);
        pTaskPoolQ->reportDone();
        pComp = pTaskPoolQ->getComponent();
    }
    while (!pTaskPoolQ->isReady()) {}
    pTaskPoolQ->close();
    logDataVariables(); //Log data variables
    mTime = mTime+mTimeStep;
}
```

Listing C.2 *Simulation code for the master simulation thread.*

D

Fork-Join Scheduler Code

```
while(mTime < mStopTime) {
    for (size_t s=0; s < mComponentSignalptrs.size(); ++s) {
        mComponentSignalptrs[s]->simulate(mTime);    //S components
    }
    //C components
    tbb::parallel_for(tbb::blocked_range<int>(0,
        mComponentCptrs.size()),
        BodySimulateComponentVector( &mComponentCptrs, mTime));
    //Q components
    tbb::parallel_for(tbb::blocked_range<int>(0,
        mComponentQptrs.size()),
        BodySimulateComponentVector( &mComponentQptrs, mTime));
    logDataVariables();
    mTime += mTimestep;
}
```

Listing D.1 *Code for the fork-join simulation loop.*

```
void operator() ( const tbb::blocked_range<int>& r ) const {
    for (int i=r.begin(); i!=r.end(); ++i) {
        mpComponentPtrs->at(i)->simulate(mStopTime);
    }
}
```

Listing D.2 *Simulation code for the simulation worker threads.*

E

Veristand Interface Source Code

```
long USER_Initialize() {
    createSystem(1e-3);
    addComponent("out1", "SignalOutputInterface");
    addComponent("Gain", "SignalGain");
    addComponent("in1", "SignalInputInterface");
    connect("in1", "out", "Gain", "in");
    connect("Gain", "out", "out1", "in");
    setParameter("Gain", "k", 1);
    initSystem();
    rtSignal.Time = 0;
    return NI_OK;
}
void USER_TakeOneStep(double *inData, double *outData,
                      double timestamp) {
    rtSignal.Time += 0.001;
    if (inData) {
        rtInport.in1 = inData[1];
    }
    writeNodeData("in1", "out", 0, rtInport.in1);
    simulateOneTimestep(rtSignal.Time);
    rtOutput.out1 = readNodeData("out1", "in", 0);
    if (outData) {
        outData[0] = rtOutput.out1;
    }
}
long USER_Finalize() {
    return NI_OK;
}
```

Listing E.1 *The Veristand interface supports functions for initializing a simulation, taking one step and finalizing.*

Bibliography

- [1] Mahder Gebremedhin. “Automatic and Explicit Parallelization Approaches for Mathematical Simulation Models”. PhD thesis. Linköping University, Software and Systems, 2015.
- [2] Kristian Stavåker. “Contributions to Simulation of Modelica Models on Data-Parallel Multi-Core Architectures”. PhD thesis. Linköping University, Software and Systems, 2015.
- [3] G. E. Moore. “Cramming More Components onto Integrated Circuits”. In: *Electronics* 38.8 (Apr. 19, 1965), pp. 114–117. ISSN: 0018-9219. DOI: 10.1109/JPROC.1998.658762.
- [4] H. Sutter. “A Fundamental Turn Toward Concurrency in Software”. In: *Dr. Dobbs’s Journal* (2005).
- [5] Patrik Nordling and Peter Fritzson. “Solving ordinary differential equations on parallel computers — applied to dynamic rolling bearings simulation”. In: *Parallel Scientific Computing*. Vol. 879. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1994, pp. 397–415. ISBN: 978-3-540-58712-5. DOI: 10.1007/BFb0030169.
- [6] L.G. Birta and O. Abou-Rabia. “Parallel block predictor-corrector methods for ODEs”. In: *IEEE Transactions of Computers* C-36 (1987), pp. 299–311. DOI: 10.1109/TC.1987.1676902.
- [7] P. Nordling and A. Sjö. “Parallel solution of modular ODEs with application to rolling bearing dynamics”. In: *Mathematics and Computers in Simulation* 44 (1997), pp. 495–504. DOI: 10.1016/S0378-4754(97)00078-5.
- [8] G.D. Byrne and A.C. Hindmarsch. “PVMODE, an ODE Solver for Parallel Computers”. In: *International Journal of High Performance Computing Applications* 13 (1999), p. 354. DOI: 10.1177/109434209901300405.

- [9] Abir Ben Khaled, Mongi Ben Gaid, and Daniel Simon. “Parallelization Approaches for the Time-Efficient Simulation of Hybrid Dynamical Systems: Application to Combustion Modeling”. In: *Proceedings of the 5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. Nottingham, UK: Linköping University Electronic Press, Apr. 2013, pp. 27–36.
- [10] Alessandro Vittorio Papadopoulos and Alberto Leva. “Automating efficiency-targeted approximations in modelling and simulation tools: dynamic decoupling and mixed-mode integration”. In: *SIMULATION* 90.10 (2014), pp. 1158–1176. DOI: 10.1177/0037549714547296.
- [11] D.M. Auslander. “Distributed System Simulation with Bilateral Delay-Line Models”. In: *Journal of Basic Engineering* (June 1968), pp. 195–200. DOI: 10.1115/1.3605079.
- [12] P.B. Johns and M.A. O’Brian. “Use Of The Transmission Line Modelling (T.L.M) Method To Solve Nonlinear Lumped Networks”. In: *The Radio And Electronic Engineer* 50(1/2) (1980), pp. 59–70. DOI: 10.1049/ree.1980.0006.
- [13] R F Boucher and E E Kitsios. “Simulation of Fluid Network Dynamics by Transmission Line Modelling”. In: *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science* 200.1 (1986), pp. 21–29. DOI: 10.1243/PIME_PROC_1986_200_090_02.
- [14] E E Kitsios and R F Boucher. “Transmission Line Modelling of a Hydraulic Position Control System”. In: *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture* 200.4 (1986), pp. 229–236. DOI: 10.1243/PIME_PROC_1986_200_077_02.
- [15] Air Force Aero Propulsion Laboratory. *Aircraft Hydraulic System Dynamic Analysis*. Technical report. Air Force Aero Propulsion Laboratory, 1977.
- [16] G J Partridge, C Christopoulos, and P B Johns. “Transmission Line Modelling of Shaft System Dynamics”. In: *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science* 201.4 (1987), pp. 271–278. DOI: 10.1243/PIME_PROC_1987_201_120_02.
- [17] S.Y.R. Hui and C. Christopoulos. “Numerical simulation of power circuits using transmission-line modelling”. In: *IEE Proceedings A (Physical Science, Measurement and Instrumentation, Management and Education)* 137 (6 Nov. 1990), 379–384(5). ISSN: 0143-702X. DOI: 10.1049/ip-a-2.1990.0060.

- [18] SYR Hui and Christos Christopoulos. “Modeling non-linear power electronic circuits with the transmission-line modeling technique”. In: *Power Electronics, IEEE Transactions on* 10.1 (1995), pp. 48–54. DOI: 10.1109/63.368461.
- [19] P. Krus, A. Jansson, J-O. Palmberg, and K. Weddfelt. “Distributed Simulation Of Hydromechanical Systems”. In: *The Third Bath International Fluid Power Workshop*. Bath, England, 1990.
- [20] P. Krus, K. Weddfelt, and J-O. Palmberg. “Fast Pipeline Models for Simulation of Hydraulic Systems”. In: *Journal Of Dynamic Systems Measurement And Control* 116 (1994), pp. 132–136. DOI: 10.1115/1.2900667.
- [21] P. Krus. “Modelling of Mechanical Systems Using Rigid Bodies and Transmission Line Joints”. In: (Dec. 1999). DOI: 10.1115/1.2802523.
- [22] Dag Fritzson, Jonas Ståhl, and Iakov Nakhimovski. “Transmission Line Co-Simulation of Rolling Bearing Applications”. In: *The 48th Scandinavian Conference on Simulation and Modeling*. Göteborg, Sweden, Oct. 2007.
- [23] B. Johansson and P. Krus. “Modelica in a Distributed Environment Using Transmission Line Modelling”. In: *Modelica 2000 Workshop*. Lund, Sweden, Oct. 2000.
- [24] S. H. Pulko, A. Mallik, R. Allen, and P. B. Johns. “Automatic timestepping in TLM routines for the modelling of thermal diffusion processes”. In: *International Journal of Numerical Modelling: Electronic Networks, Devices and Fields* 3.2 (1990), pp. 127–136. ISSN: 1099-1204. DOI: 10.1002/jnm.1660030208.
- [25] A. Jansson, P. Krus, and J-O Palmberg. “Variable time step size applied to simulation of fluid power systems using transmission line elements”. In: *Fifth Bath International Fluid Power Workshop*. Bath, England, 1992.
- [26] S.Y.R. Hui, K.K. Fung, M.Q. Zhang, and C. Christopoulos. “Variable time step technique for transmission line modelling”. In: *IEE Proceedings A (Science, Measurement and Technology)* 140 (4 July 1993), 299–302(3). ISSN: 0960-7641. DOI: 10.1049/ip-a-3.1993.0046.
- [27] Taco J. Viersma. *Analysis, synthesis, and design of hydraulic servosystems and pipelines*. Elsevier Scientific Pub. Co., 1980. ISBN: 0444418695.
- [28] Oliver Heaviside. “Electromagnetic induction and its propagation”. In: *The Electrician* (1887).
- [29] J.D. Burton, K.A. Edge, and C.R. Burrows. “Partitioned Simulation Of Hydraulic Systems Using Transmission-Line Modelling”. In: *ASME WAM* (1993).

- [30] J. D. Burton, K. A. Edge, and C. R. Burrows. “Modeling Requirements for the Parallel Simulation of Hydraulic Systems”. In: *Journal of Dynamic Systems, Measurement, and Control* 116 (1994), pp. 137–145. DOI: 10.1115/1.2900668.
- [31] K.K. Fung and S.Y.R. Hui. “Transputer simulation of decoupled electrical circuits”. In: *Mathematics and Computers in Simulation* 42 (1997), pp. 1–13. DOI: 10.1016/0378-4754(95)00051-8.
- [32] A. Jansson and P. Krus. “Real-Time Simulation Using Parallel Processing”. In: *The Second Tampere International Conference On Fluid Power*. Tampere, Finland, 1994.
- [33] Kaj Nyström and Peter Fritzson. “Parallel Simulation with Transmission Lines in Modelica”. In: *5th International Modelica Conference*. Vienna, Austria, Sept. 2006.
- [34] Martin Sjölund, Mahder Gebremedhin, and Peter Fritzson. “Parallelizing Equation-Based Models for Simulation on Multi-Core Platforms by Utilizing Model Structure”. In: *Proceedings of the 17th Workshop on Compilers for Parallel Computing*. Lyon, France, July 2013.
- [35] Peter B Johns and Raymond L Beurle. “Numerical solution of 2-dimensional scattering problems using a transmission-line matrix”. In: *Proceedings of the Institution of Electrical Engineers*. Vol. 118. 9. IET. 1971, pp. 1203–1208.
- [36] David W Bauer Jr and Ernest H Page. “Optimistic parallel discrete event simulation of the event-based transmission line matrix method”. In: *Simulation Conference, 2007 Winter*. IEEE. 2007, pp. 676–684. DOI: 10.1109/WSC.2007.4419661.
- [37] David W Bauer Jr, Christopher D Carothers, and Akintayo Holder. “Scalable time warp on blue gene supercomputers”. In: *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*. IEEE Computer Society. 2009, pp. 35–44. DOI: 10.1109/PADS.2009.21.
- [38] Sudip K Seal and Kalyan S Perumalla. “Reversible parallel discrete event formulation of a tlm-based radio signal propagation model”. In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 22.1 (2011), p. 4. DOI: 10.1145/2043635.2043639.
- [39] Alexander Siemers, Dag Fritzson, and Iakov Nakhimovski. “General meta-model based co-simulations applied to mechanical systems”. In: *Simulation Modelling Practice And Theory* 17.4 (2009), pp. 612–624. DOI: doi:10.1016/j.simpat.2008.10.006.

- [40] E. Lelarsmee, Albert E. Ruehli, and A.L. Sangiovanni-Vincentelli. “The Waveform Relaxation Method for Time-Domain Analysis of Large Scale Integrated Circuits”. In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 1.3 (July 1982), pp. 131–145. ISSN: 0278-0070. DOI: 10.1109/TCAD.1982.1270004.
- [41] Alessandro Vittorio Papadopoulos and Alberto Leva. “Automating dynamic decoupling in object-oriented modelling and simulation tools”. In: *5th Workshop on Equation-Based Object-Oriented Modeling Languages and Tools EOOLT, Linköping Univ. Electronic Press*. Nottingham, England, 2013, pp. 37–44.
- [42] Abir Ben Khaled, Mohamed El Mongi Ben Gaïd, Daniel Simon, and Gregory Font. “Multicore simulation of powertrains using weakly synchronized model partitioning”. In: *E-COSM’12 IFAC Workshop on Engine and Powertrain Control Simulation and Modeling*. Reuil-Malmaison, France, Oct. 2012.
- [43] Abir Ben Khaled-El Feki. “Distributed real-time simulation of numerical models : application to power-train”. Theses. Université de Grenoble, May 2014.
- [44] Linköping University, Division of Fluid and Mechatronic Systems. *Hopsan Project*. 2015. URL: <http://www.iei.liu.se/flumes/system-simulation/hopsan> (visited on 06/02/2015).
- [45] Jonas Larsson. *User’s guide to Hopsan*. Linköping University, Department of Mechanical Engineering. Linköping, Sweden, Aug. 2002.
- [46] B. Eriksson, P. Nordin, and P. Krus. “Hopsan NG, A C++ Implementation Using The TLM Simulation Technique”. In: *The 51st Conference On Simulation And Modelling*. Oulu, Finland, 2010.
- [47] P. Krus and J-O. Palmberg. “Integration of Simulation and Measurement in Time and Frequency Domains”. In: *International Conference on Fluid Power Transmission and Control*. Hangzhou, China, 1989.
- [48] The Qt Company. *Qt / Cross-Platform Application & UI Development Framework*. 2015. URL: <http://qt.io/> (visited on 06/02/2015).
- [49] T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauß, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J.-V. Peetz, and S. Wolf. “The Functional Mockup Interface for Tool independent Exchange of Simulation Models”. In: *8th International Modelica Conference 2011*. Como, Italy, Sept. 2009.
- [50] Modelica Association. *Functional Mock-up Interface*. Nov. 6, 2015. URL: <https://www.fmi-standard.org/> (visited on 11/06/2015).
- [51] Jonas Larsson. “A Framework for Implementation-Independent Simulation Models”. In: *SIMULATION* 82.9 (2006), pp. 563–579. DOI: 10.1177/0037549706071902.

- [52] Iakov Nakhimovski. “Contributions to the Modeling and Simulation of Mechanical Systems with Detailed Contact Analyses”. PhD thesis. Linköping University, PELAB - Programming Environment Laboratory, 2006.
- [53] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Computer Society Pr, 2006. ISBN: 0471471631.
- [54] The MathWorks, Inc. *Simscape*. URL: <http://se.mathworks.com/products/simscape> (visited on 10/07/2015).
- [55] Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. “The OpenModelica modeling, simulation, and software development environment”. In: *Simulation News Europe* 44 (2005), pp. 8–16.
- [56] Johan Åkesson, Magnus Gäfvert, and Hubertus Tummescheit. “JModelica—an Open Source Platform for Optimization of Modelica Models”. In: *6th Vienna International Conference on Mathematical Modelling*. 2009.
- [57] Wolfram. *Wolfram System Modeler*. 2015. URL: <http://www.wolfram.com/system-modeler/> (visited on 10/07/2015).
- [58] Modelon. *Dymola*. URL: <http://www.modelon.com/products/dymola/> (visited on 10/07/2015).
- [59] ITI GmbH. *SimulationX*. URL: <https://www.simulationx.com/> (visited on 10/07/2015).
- [60] Maplesoft™. *MapleSim*. URL: <http://www.maplesoft.com/products/maplesim/> (visited on 10/07/2015).
- [61] Peter Aronsson. “Automatic Parallelization of Equation-Based Simulation Programs”. PhD thesis. Linköping University, PELAB - Programming Environment Laboratory, 2006.
- [62] Håkan Lundvall, Kristian Stavåker, Peter Fritzson, and Christoph Kessler. “Automatic Parallelization of Simulation Code for Equation-based Models with Software Pipelining and Measurements on Three Platforms”. In: *Computer Architecture News. Special Issue MCC08 – Multi-Core Computing* 36.5 (Dec. 2008). DOI: 10.1145/1556444.1556451.
- [63] Joshua D. Carl, Gautam Biswas, Sandeep Neema, and Ted Bapty. “An Approach to Parallelizing the Simulation of Complicated Modelica Models”. In: *Proceedings of the 2014 Summer Simulation Multiconference*. SummerSim ’14. Monterey, California: Society for Computer Simulation International, 2014, 17:1–17:8.

-
- [64] Mahder Gebremedhin, Afshin Hemmati Moghadam, Peter Fritzson, and Kristian Stavaker. “A Data-Parallel Algorithmic Modelica Extension for Efficient Execution on Multi-Core Platforms”. In: *Proceedings of the 9th International Modelica Conference*. Munich, Germany, Sept. 2012. DOI: 10.3384/ecp12076393.
- [65] Martina Maggio, Kristian Stavåker, Filippo Donida, Francesco Casella, and Peter Fritzson. “Parallel Simulation of Equation-based Object-Oriented Models with Quantized State Systems on a GPU”. In: *Proceedings of the 7th International Modelica Conference*. Como, Italy, Sept. 2009.
- [66] Kristian Stavåker and Peter Fritzson. “Generation of Simulation Code from Equation-Based Models for Execution on CUDA-Enabled GPUs”. In: *Proceedings of the 3rd Swedish Workshop on Multi-Core Computing*. 2010.
- [67] Per Ostlund, Kristian Stavaker, and Peter Fritzson. “Parallel Simulation of Equation-based Models on CUDA-enabled GPUs”. In: *Proceedings of the 9th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing*. POOSC ’10. Reno, Nevada: ACM, 2010, 5:1–5:6. ISBN: 978-1-4503-0546-4. DOI: 10.1145/2039312.2039317.
- [68] Kristian Stavåker, D. Rolls, J. Guo, Peter Fritzson, and S.-B. Scholz. “Compilation of Modelica array computations into single assignment C for efficient execution on CUDA-enabled GPUs”. In: *Proceedings of the 3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, EOOLT 2010 - In Conjunction with MODELS 2010*. Proceedings of the 3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, EOOLT 2010 - In Conjunction with MODELS 2010. 2010, pp. 81–90.
- [69] Robert Hooke and T. A. Jeeves. ““ Direct Search” Solution of Numerical and Statistical Problems”. In: *J. ACM* 8.2 (Apr. 1961), pp. 212–229. ISSN: 0004-5411. DOI: 10.1145/321062.321069.
- [70] William C. Davidon. “Variable Metric Method for Minimization”. In: *SIAM Journal on Optimization* 1.1 (1991), pp. 1–17. DOI: 10.1137/0801001.
- [71] Virginia Torczon and Michael W Trosset. “From evolutionary operation to parallel direct search: Pattern search algorithms for numerical optimization”. In: *Computing Science and Statistics* (1998), pp. 396–401.
- [72] J. A. Nelder and R. Mead. “A Simplex Method for Function Minimization”. In: *The Computer Journal* 7.4 (1965), pp. 308–313. DOI: 10.1093/comjnl/7.4.308.

- [73] W. Spendley, G. R. Hext, and F. R. Himsworth. “Sequential Application of Simplex Designs in Optimisation and Evolutionary Operation”. In: *Technometrics* 4.4 (1962), pp. 441–461. ISSN: 00401706. DOI: 10.1080/00401706.1962.10490033.
- [74] John E Dennis Jr and Virginia Torczon. “Direct search methods on parallel machines”. In: *SIAM Journal on Optimization* 1.4 (1991), pp. 448–474. DOI: 10.1007/978-3-642-48417-9_2.
- [75] Donghoon Lee and Matthew Wiswall. “A parallel implementation of the simplex function minimization routine”. In: *Computational Economics* 30.2 (2007), pp. 171–187. DOI: 10.1007/s10614-007-9094-2.
- [76] Gerhard Fritsch and Horst Müller. “Parallelization of a minimization problem for multi-processor systems”. In: *Conpar 81*. Springer, 1981, pp. 453–463.
- [77] M. J. Box. “A New Method of Constrained Optimization and a Comparison With Other Methods”. In: *The Computer Journal* 8.1 (1965), pp. 42–52. DOI: 10.1093/comjnl/8.1.42.
- [78] JA Guin. “Modification of the complex method of constrained optimization”. In: *The Computer Journal* 10.4 (1968), pp. 416–417.
- [79] Petter Krus. “Simulation Based Optimization for System Design”. In: *International Conference on Engineering Design*. Stockholm, Sweden, Aug. 2003.
- [80] Marcus Pettersson, Johan Andersson, and Petter Krus. “Methods for Discrete Design Optimization”. In: *Statistics* 2009 (2004), pp. 05–29.
- [81] Russ C Eberhart and Yuhui Shi. “Comparing inertia weights and constriction factors in particle swarm optimization”. In: *Evolutionary Computation, 2000. Proceedings of the 2000 Congress on*. Vol. 1. IEEE. 2000, pp. 84–88.
- [82] J. Kennedy and R. Eberhart. “Particle swarm optimization”. In: *Neural Networks, 1995. Proceedings., IEEE International Conference on*. Vol. 4. 1995, 1942–1948 vol.4.
- [83] JF. Schutte and AA. Groenwold. “A Study of Global Optimization Using Particle Swarms”. In: *Journal of Global Optimization* 31.1 (2005), pp. 93–108. ISSN: 0925-5001. DOI: 10.1007/s10898-003-6454-x.
- [84] Shu-Kai S. Fan and Yi-Yin Chiu. “A decreasing inertia weight particle swarm optimizer”. In: *Engineering Optimization* 39.2 (2007), pp. 203–228. DOI: 10.1080/03052150601047362.
- [85] JC Bansal, PK Singh, Mukesh Saraswat, Abhishek Verma, Shimpi Singh Jadon, and Ajith Abraham. “Inertia weight strategies in particle swarm optimization”. In: *Nature and Biologically Inspired Computing (NaBIC), 2011 Third World Congress on*. IEEE. 2011, pp. 633–640.

- [86] JF Schutte, BJ Fregly, RT Haftka, and AD George. *A parallel particle swarm optimizer*. Tech. rep. DTIC Document, 2003.
- [87] Byung-Il Koh, Alan D George, Raphael T Haftka, and Benjamin J Fregly. “Parallel asynchronous particle swarm optimization”. In: *International Journal for Numerical Methods in Engineering* 67.4 (2006), pp. 578–595. DOI: 10.1002/nme.1646.
- [88] David E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. 2. Reading, MA, USA: Addison-Wesley Longman Publishing Co., 1989. ISBN: 0201157675.
- [89] Rainer Storn and Kenneth Price. “Differential evolution—a simple and efficient heuristic for global optimization over continuous spaces”. In: *Journal of global optimization* 11.4 (1997), pp. 341–359. DOI: 10.1023/A:1008202821328.
- [90] ©Intel Corporation. *Intel® Threading Building Blocks User Guide*. URL: https://www.threadingbuildingblocks.org/docs/help/tbb_userguide/ (visited on 06/10/2015).
- [91] Krishna Kavi, Hyong-Shik Kim, Ben Lee, and AR Hurson. “Shared Memory And Distributed Shared Memory Systems: A Survey”. In: *Emphasizing Distributed Systems* 53 (2000), p. 55.
- [92] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. “Greedy Algorithms”. In: *Introduction to Algorithms*. 2nd ed. MIT Press, 2001. Chap. 16, pp. 370–404. ISBN: 0-262-03293-7.
- [93] Narendra Karmarkar and Richard M. Karp. *The Differencing Method of Set Partitioning*. Tech. rep. Berkeley, CA, USA: University of California at Berkeley, 1983.
- [94] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Burlington, MA: Morgan Kaufmann, 2008.
- [95] Cristiano Cardoso Locateli, Paulo Leonel Teixeira, Edson Roberto De Pieri, Petter Krus, and Victor Juliano De Negri. “Digital hydraulic system using pumps and on/off valves controlling the actuator”. In: *8th FPNI Ph. D Symposium on Fluid Power*. American Society of Mechanical Engineers. Lappeenranta, Finland, June 2014. DOI: 10.1115/FPNI2014-7839.
- [96] The Mathworks, Inc. *S-Functions*. 2015. URL: <http://se.mathworks.com/help/rtw/s-functions.html> (visited on 06/02/2015).
- [97] Isak Demir. “Multibodysimulering och Analys av en Smalgångstruck”. MA thesis. Linköping University, Fluid and Mechatronic Systems, Aug. 2011.
- [98] Erik Israelsson. “Modellering och simulering av hydraulik för användning i hardware-in-the-loop”. MA thesis. Linköping University, Fluid and Mechatronic Systems, June 2012.

- [99] Petter Krus. “Robust Modelling Using Bi-Lateral Delay Lines for Real Time and Faster Than Real Time System Simulation”. In: *ASME 2009 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. American Society of Mechanical Engineers. 2009, pp. 131–138.
- [100] *Getting Started with LabVIEW*. National Instruments. June 2012.
- [101] *Viking GUI - User Manual*. Prevas. Oct. 2010.
- [102] National Instruments. *Using the NI Veristand™ Model Framework*. National Instruments Corporation. Austin, USA, 2015. (Visited on 06/02/2015).
- [103] Modelica Association Project Function Mock-up Interface. *Functional Mock-up Interface for Model Exchange and Co-Simulation*. Version 2.0. July 25, 2015. (Visited on 10/09/2015).
- [104] Petter Krus. “An Automated Approach for Creating Component and Subsystem Models for Simulation of Distributed Systems”. In: *Proceedings of the Ninth Bath International Fluid Power Workshop*. Bath, England, 1996.
- [105] P. Krus and J. Ölvander. “Performance index and meta-optimization of a direct search optimization method”. In: *Engineering Optimization* 45.10 (2013), pp. 1167–1185. DOI: 10.1080/0305215X.2012.725052.

Papers

The articles associated with this thesis have been removed for copyright reasons. For more details about these see:

<http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-122754>