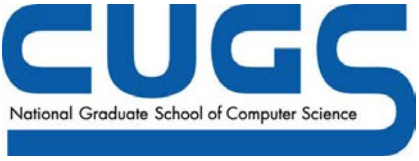
Linköping Studies in Science and Technology

Dissertations. No. 1666

Contributions to Simulation of Modelica Models on Data-Parallel Multi-Core Architectures

by

Kristian Stavåker



Department of Computer and Information Science
Linköping University
SE-581 83 Linköping, Sweden

Linköping 2015

ISBN 978-91-7519-068-6
ISSN 0345-7524
Printed by LiU Tryck 2015

URL: http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-116338

Abstract

Modelica is an object-oriented, equation-based modeling and simulation language being developed through an international effort by the Modelica Association. With Modelica it is possible to build computationally demanding models; however, simulating such models might take a considerable amount of time. Therefore techniques of utilizing parallel multi-core architectures for faster simulations are desirable. In this thesis the topic of simulation of Modelica on parallel architectures in general and on graphics processing units (GPUs) in particular is explored. GPUs support code that can be executed in a data-parallel fashion. It is also possible to connect and run several GPUs together which opens opportunities for even more parallelism. In this thesis several approaches regarding simulation of Modelica models on GPUs and multi-core architectures are explored.

In this thesis the topic of expressing and solving partial differential equations (PDEs) in the context of Modelica is also explored, since such models usually give rise to equation systems with a regular structure, which can be suitable for efficient solution on GPUs. Constructs for PDE-based modeling are currently not part of the standard Modelica language specification. Several approaches on modeling and simulation with PDEs in the context of Modelica have been developed over the years. In this thesis we present selected earlier work, ongoing work and planned work on PDEs in the context of Modelica. Some approaches detailed in this thesis are: extending the language specification with PDE handling; using a software with support for PDEs and automatic discretization of PDEs; and connecting an external C++ PDE library via the functional mockup interface (FMI).

Finally the topic of parallel skeletons in the context of Modelica is explored. A skeleton is a predefined, generic component that implements a common specific pattern of computation and data dependence. Skeletons provide a high degree of abstraction and portability and a skeleton can be customized with user code. Using skeletons with Modelica opens up the possibility of executing heavy Modelica-based matrix and vector computations on multi-core architectures. A working Modelica-SkePU library with some minor necessary compiler extensions is presented.

This work has been supported by the European ITEA2 OPENPROD project (Open Model-Driven Whole-Product Development and Simulation Environment), the European ITEA3 MODRIO project (Model Driven Physical Systems Operation) and by the National Graduate School of Computer Science (CUGS)

Populärvetenskaplig sammanfattning

Modelica är ett objektorienterat, ekvationsbaserat modellerings- och simuleringsspråk som utvecklas via den internationella organisationen the Modelica Association. Med Modelica är det möjligt att bygga beräkningskrävande modeller vilket kan leda till långa simuleringstider. Därför är metoder för att utnyttja parallella flerkärniga arkitekturer för snabbare simuleringar önskvärda. I denna avhandling utforskas området simulering av Modelicamodeller på parallella arkitekturer i allmänhet och på grafikbearbetningsenheter (GPUs) i synnerhet. GPU-kod kan köras data-parallellt. Det är också möjligt att ansluta och köra flera GPUs tillsammans vilket öppnar upp möjligheter för ännu mer parallellism. I denna avhandling utforskas flera metoder avseende simulering av Modelicamodeller på GPUs och multi-core arkitekturer.

I denna avhandling utforskas också ämnet att uttrycka och lösa partiella differentialekvationer (PDE:er) i Modelica. Modeller innehållande PDE:er ger vanligtvis upphov till ekvationssystem med en regelbunden data-parallel struktur, som lämpar sig för effektiv lösning på grafikprocessorer. Konstruktioner för PDE-baserad modellering ingår för närvarande inte i språkspecifikationen för Modelicastandarden. Flera metoder för modellering och simulering av PDE:er med Modelica har utvecklats genom åren. I denna avhandling presenterar vi utvalda tidigare arbeten, pågående arbeten, och planerade arbeten med PDE:er med Modelica. Några av metoderna som beskrivs i denna avhandling är: utvidga språkspecifikationen med PDE-hantering; stöd för PDE:er och automatisk diskretisering av PDE:er med hjälp av speciell programvara; och att ansluta ett externt C++ PDE bibliotek via det så kallade functional mockup interfacet (FMI).

Slutligen studerar vi ämnet parallella skelett tillsammans med Modelica. Ett skelett är en fördefinierad, generisk programkomponent som implementerar ett gemensamt specifikt mönster av beräkning och databeroende. Skelett ger en hög grad av abstraktion och ett skelett kan skräddarsys med användarkod. Att använda skelett tillsammans med Modelica öppnar upp möjligheten att utföra tunga Modelicabaserade matris- och vektorberäkningar på flerkärniga arkitekturer. Ett fungerande Modelica-SkePU bibliotek tillsammans med några mindre kompilatorutvidgningar presenteras.

Acknowledgements

First of all I would like to express my greatest gratitude to my main supervisor Professor Peter Fritzson for accepting me as a PhD student and for giving me guidance over the years. I would also like to thank Professor Christoph Kessler my secondary supervisor, Professor Kristian Sandahl who is head of the PELAB research group, Åsa Kärrman, an administrator at PELAB, and Anne Moe, coordinator of graduate studies administration at IDA. I would like to thank Erik Hansson, Bernhard Thiele and Nicolas Melot for help with reviewing my thesis and giving suggestions on improvements. I also like to thank my sister Kirsti Stavåker for help with English language review. I would also like to thank my other colleagues at PELAB, past and present, for a good working environment and for their support and help. I would like to thank the EMCL research group at Heidelberg University, Germany (previously at Karlsruhe University of Technology) for hosting me at their research group for three months in early 2013 and to CUGS for partially financing this stay. I am also very thankful for the kind help I have been given when it comes to administrative issues from a number of IDA administrators over the years (none mentioned, none forgotten!). I would like to thank my parents Eeva Partanen and Leif Stavåker as well as Johan, Elmer and Melker. I dedicate this thesis to my grandmother Kerstin Stavåker.

Contents

List of Figures

List of Tables

Part I

Prologue

Chapter 1

Introduction

This chapter begins with the motivation for investigating the research problems presented in this thesis work. The chapter continues with statement of the research questions, a brief description of the research process, a summary to the contributions of this thesis and the delimitations concerning the models used for generating code. The chapter closes with a list of publications and a section with the thesis outline.

1.1 Motivation

By using the equation-based object-oriented modeling language Modelica [66, 37, 38] it is possible to model large (in the sense of giving rise to many equations) and complex physical systems from various application domains (such as mechatronics, power generation, wind power plants, multi-body systems, hydraulics, automotive applications, power-train systems, etc.). Large and complex models will typically result in large differential and algebraic equation systems. Numerical solution of large systems of differential equations, which in this context equates to simulation, can be quite time consuming. It is therefore relevant to investigate how parallel multi-core architectures can be used to speedup simulation. This has also been a research goal in our research group Programming Environments Laboratory (PELAB) at Linköping University for several years, see for instance [21, 55, 19, 59]. This work involves both the actual code generation process and (modifying) the simulation runtime system. Several different parallel architectures have been targeted, such as for Intel multi-cores, STI[1] Cell BE, and Graphics Processing Unit (GPU). In this thesis the main focus is on GPUs. GPUs can be used to perform general purpose scientific and engineering computing in addition to their use for graphics processing. The theoretical processing power of GPUs has surpassed that of CPUs due to the highly parallel structure of GPUs. GPUs are, however, only good at solving certain problem

[1] An alliance between Sony, Toshiba, and IBM

types that are primarily data-parallel.

In this thesis the topic of PDEs in the context of Modelica is also explored. One reason is that PDE-systems often result in large systems of equations of a regular structure that can be suitable for efficient execution on GPUs. However, PDE language constructs for modeling with PDEs are currently not a part of the Modelica standard language specification. Several approaches for modeling with PDEs in the context of Modelica have been developed over the years. See for instance [76, 34, 93, 78, 77, 58, 57]. In this thesis previous, ongoing, and planned work on PDEs in the context of Modelica is presented. Extending the Modelica language specification to support formulation of PDEs; using software with support for PDEs and automatic discretization of PDEs, and connecting an external C++ PDE library via the Functional Mockup Interface (FMI) are some approaches detailed in this thesis.

Lastly, the thesis explores the topic of parallel skeletons in the context of Modelica. A SkePU [45, 46, 86, 85, 8, 28, 29] skeleton is a predefined, generic component that implements a common specific (parallel) pattern of computation and data dependence. Skeletons provide a high degree of abstraction and portability and a skeleton can be customized with user code. Using skeletons with Modelica opens the way for efficiently executing certain kinds of heavy Modelica-based matrix and vector computations on multi-core architectures.

The topic of execution of Modelica models on multi-core architectures is what binds this thesis together.

1.2 Research Questions

The main research questions of this work are set out below.

- Is it possible to simulate Modelica models with GPU architectures? Will such simulations run at sufficient speed compared to simulation on other architectures, for instance single- and multi-core CPUs? Are GPUs beneficial for performance? What challenges are there in terms of hardware limitations, memory limitations, etc.?
- What is the current state of modeling using PDEs in the context of Modelica? What previous research has been done in this area and what are the strengths and weaknesses of this previous research? What are the strengths and weaknesses of the approach of connecting an external (finite element) solver to the Modelica environment via functional mockup interface?
- What is the current state of skeleton programming in the context of Modelica? What previous research has been done in this area and

what are the strengths and weaknesses of this previous research? What are the strengths and weaknesses of the approach of implementing a Modelica-SkePU skeleton compilation and library approach?

1.3 Research Process

The following general research steps have been carried out in preparing this thesis.

- Literature study of background theory
- Literature study of earlier work
- Theoretical derivations and design
- Implementation in the open-source OpenModelica compiler and runtime system combined with measurements of execution times for various models
- Implementation of Modelica library code and accompanying external C/C++ code
- Presentation of papers at workshops and conferences and publication of proceedings for reviews, comments and valuable feedback
- Research visits at external research groups as well as attending various summer schools and tutorials

The research methodology used in this work is the traditional system oriented computer science research method, that is, in order to validate our hypotheses prototype implementations are built. The prototypes are used to simulate Modelica models on both serial and parallel architectures, and the simulation times are then compared. In this way speedup can be calculated.

Regarding research methodology, the ACM Task Force on the core of computer science has suggested three different paradigms for conducting research within the discipline of computing: theory, abstraction (modeling), and design [22]. The first discipline is rooted in mathematics, the second discipline is rooted in experimental scientific methods, and the third discipline is rooted in engineering and consists of stating requirements, defining the specification, designing the system, implementing the system and finally testing and evaluating the system. All three paradigms are considered to be equally important. Computer science and engineering consist of a mixture of all three paradigms.

All implementation tasks in this thesis have been conducted in the open-source OpenModelica development environment [71]. OpenModelica is an open-source implementation of a Modelica compiler, simulator and development environment, and its development is supported by the Open Source

Modelica Consortium (OSMC). See Section 2.2 for more information. External C/C++ code and Modelica model code has also been written as a part of work with this thesis.

1.4 Contributions

The following represent the main contributions to this work:

- Methods for compilation and simulation of Modelica models on GPUs
- Methods of keeping Modelica array constructs unexpanded through the compilation process, thereby increasing scalability and simplifying mapping to GPUs
- Methods of connecting Modelica code with PDE solving code via FMI
- A Modelica-SkePU library together with some minor compiler extensions to support parallel computational skeletons for use with Modelica

1.5 Delimitations

The main delimitations concern the models selected for which code is generated. Only (mainly) a subset of possible Modelica models have been investigated:

- Models that are purely continuous with respect to time.
- Models that can be reduced to Ordinary Differential Equation (ODE) systems.
- Models where the values of all constants and parameters are known at compile time.

1.6 List of Publications

This thesis is mainly based on the following publications.

- *Publication 1* Håkan Lundvall, Kristian Stavåker, Peter Fritzson, Christoph Kessler. *Automatic Parallelization of Simulation Code for Equation-based Models with Software Pipelining and Measurements on Three Platforms.* MCC'08 Workshop, Ronneby, Sweden, November 27-28, 2008. [43]
- *Publication 2* Martina Maggio, Kristian Stavåker, Filippo Donida, Francesco Casella, 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 (Modelica'2009), Como, Italy, September 20-22, 2009. [60]

- *Publication 3* Kristian Stavåker, Daniel Rolls, Jing Guo, Peter Fritzson, Sven-Bodo Scholz. *Compilation of Modelica Array Computations into Single Assignment C for Efficient Execution on Compute Unified Device Architecture (CUDA)-enabled GPUs.* 3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, Oslo, Norway, October 3, 2010. [52]

- *Publication 4* Per Östlund, Kristian Stavåker, Peter Fritzson. *Parallel Simulation of Equation-Based Models on CUDA-Enabled GPUs.* POOSC Workshop, Reno, Nevada, October 18, 2010. [74]

- *Publication 5* Kristian Stavåker, Peter Fritzson. *Generation of Simulation Code from Equation-Based Models for Execution on CUDA-Enabled GPUs.* MCC'10 Workshop, Gothenburg, Sweden, November 18-19, 2010. [53]

- *Publication 6* Afshin Hemmati Moghadam, Mahder Gebremedhin, Kristian Stavåker, Peter Fritzson. *Simulation and Benchmarking of Modelica Models on Multi-Core Architectures with Explicit Parallel Algorithmic Language Extensions.* MCC'11 Workshop, Linköping, Sweden, November 23-25, 2011. [15]

- *Publication 7* Mahder Gebremedhin, Afshin Hemmati Moghadam, Peter Fritzson, Kristian Stavåker. *A Data-Parallel Algorithmic Modelica Extension for Efficient Execution on Multi-Core Platforms.* In Proceedings of the 9th International Modelica Conference (Modelica'2012), Munich, September 3-5, 2012. [56]

- *Publication 8* Kristian Stavåker, Staffan Ronnås, Martin Wlotzka, Vincent Heuveline, Peter Fritzson. *PDE Modeling with Modelica via FMI Import of HiFlow3 C++ Components.* SIMS'2013 Workshop, 54rd SIMS Conference on Simulation and Modeling, Bergen, Norway, October 16-18, 2013. [50]

- *Publication 9* Chen Song, Kristian Stavåker, Martin Wlotzka, Peter Fritzson, Vincent Heueveline. *PDE Modeling with Modelica via FMI Import of HiFlow3 C++ Components with Parallel Multi-Core Simulations.* SIMS'2014 Workshop, 55th SIMS Conference on Simulation and Modeling, Aalborg, Denmark, October 21-22, 2014. [49]

Other publications (pre-PhD studies) by the author not covered in this thesis.

- *Publication X* Adrian Pop, Kristian Stavåker, Peter Fritzson. *Exception Handling for Modelica.* In Proceedings of the 6th International Modelica Conference (Modelica'2008), Bielefeld, Germany, March.3-4, 2008. [14]

- *Publication Y* Kristian Stavåker, Adrian Pop, Peter Fritzson. *Compiling and Using Pattern Matching in Modelica.* In Proceedings of the 6th International Modelica Conference (Modelica'2008), Bielefeld, Germany, March.3-4, 2008. [51]

1.7 Thesis Outline

Since the publications listed in the previous section includes contributions by several persons it is important to state which parts have been done by the author of this thesis and which parts have been done by others.

1.7.1 Part I

Part I includes the thesis prologue: motivation, research questions, research process, contributions, delimitations, list of publications, thesis outline, background, and previous research. This part has been entirely written by the author of this thesis.

1.7.2 Part II

Part II includes material from the author's licentiate thesis [80]. This part is based on publications 1,2,3,4,5,6 and 7.

- **Chapter 5** This chapter is mainly based on Publication 1 which contains (updated) material from Håkan Lundvall's licentiate thesis [55] as well as new material about targeting the Cell BE architecture for simulation of equation-based models. The author did the actual mapping to the Cell BE processor. The author was highly involved and co-authored the paper.

- **Chapter 6** This chapter is mainly based on Publication 2. In this paper ways of using the Quantized State System (QSS) simulation method with NVIDIA GPUs were investigated. The author implemented the OpenModelica backend QSS code generator. The author was highly involved and co-authored the actual paper.

- **Chapter 7** This chapter is a summary of Publication 4. The chapter describes the work of creating a task graph of the model equation system and then scheduling this task graph for execution. The implementation work was done by Per Östlund and is described in his master's thesis [73]. The author co-authored the paper, held the paper presentation and supervised master's thesis work.

- **Chapter 8** This chapter is mainly based on Publication 3. The chapter discusses compiling Modelica array constructs into an intermediate language, Single Assignment C (SAC), from which highly efficient code can be generated for instance for execution with CUDA-enabled GPUs. The author has been highly involved with the design, experimental setup and measurements, and co-authored the paper.
- **Chapter 9** This chapter is mainly based on Publication 6. This chapter addresses compilation and benchmarking of the algorithmic subset of Modelica, primarily to OpenCL executed on GPUs and Intel multi-cores. Implementation and measurements were performed by two master's students (Mahder Gebremedhin and Afshin Hemmati Moghadam), supervised by the author. The publication was mainly authored by the master's students, but with contributions by the author.
- **Chapter 10** This chapter describes preliminary results of ways to keep the Modelica array equations unexpanded through the compilation process. The author was highly involved with this work and conducted a prototype implementation supporting this for a subset of Modelica, as well as authoring the complete chapter.

1.7.3 Part III

Part III includes chapters on PDE Modeling in the context of Modelica and is based on Publications 8 and 9. The author of this thesis was highly involved in this work: taking the initial initiative to travel to the Karlsruhe Institute of Technology Germany to start collaboration, studying the finite element method and was heavily involved both in the implementation work and in writing the two workshop publications. The author of this thesis was the Modelica expert (including compilation and runtime issues) while the EMCL research group provided the mathematical expertise regarding PDE solving and the HiFlow3 software.

1.7.4 Part IV

Part IV includes a chapter on parallel skeleton pattern programming in the context of Modelica. The author of this thesis carried out all implementation work for the Modelica-SkePU library and compiler extensions, with advice from the co-authors, and wrote the test cases and conducted measurements as well.

1.7.5 Part VI

Part VI presents an epilogue: conclusions and future work chapters. Completely written by the author of this thesis.

Chapter 2

Background

This chapter starts with an introduction of the Modelica modeling language and the open-source OpenModelica compiler. The chapter then continues with an introduction of some mathematical concepts and a description of the general compilation process of Modelica code.

2.1 The Modelica Modeling and Simulation Language

Modelica is a modeling language for equation-based, object-oriented mathematical modeling that is being developed through an international effort via the Modelica Association [66, 38]. Since Modelica is an equation-based language it supports modeling in an acausal form. This is in contrast to a conventional programming language for which the user would first have to manually transform the model equations into causal (assignment) statement form, also called causalization. However, with Modelica it is possible to write equations directly in the model code and letting the compiler in question taking care of the causalization. When writing Modelica models, it is also possible to utilize high-level concepts such as object-oriented modeling and component composition. An example of a Modelica model is provided below in Listing 2.1.

The model in Listing 2.1 describes a simple circuit consisting of various components as well as a source and ground. Several components are instantiated from various classes (Resistor class, Capacitor class, etc.) and these are then connected together with connect clauses. The connect is an equation construct since it expands into one or more equations. Subsequently a Modelica compiler can be used to compile this model into code that can be linked with a runtime system for simulation, where the main runtime part consists of a numerical solver; see Section 2.3.2. All the object-oriented

structure is typically removed at the beginning of the compilation process and the connect equations are expanded into standard equality equations.

Listing 2.1: *A Modelica model for a simple electrical circuit.*

```
model Circuit
  Resistor R1(R=10);
  Capacitor C(C=0.01);
  Resistor R2(R=100);
  Inductor L(L=0.1);
  VsourceAC AC;
  Ground G;
equation
  connect(AC.p, R1.p);
  connect(R1.n, C.p);
  connect(C.n, AC.n);
  connect(R1.p, R2.p);
  connect(R2.n, L.p);
  connect(L.n, C.n);
  connect(AC.n, G.p);
end Circuit;
```

Modelica and Equation-Based Object-Oriented (EOO) languages in general support the following concepts:

- **Equations**
- **Models/Classes**
- **Objects**
- **Inheritance**
- **Polymorphism**
- **Acausal Connections**

Continuous-time differential and/or algebraic equations make it possible to model continuous-time systems. There are also discrete equations available for modeling hybrid systems, i.e., systems with both continuous-time and discrete-time parts. The Modelica language has a uniform design meaning that everything, e.g., models, packages, real numbers, etc., are classes. A Modelica class can be of different specialized kinds of classes denoted by different class keywords such as model, class, record, connector, package, etc. From the Modelica class, objects can be instantiated. Just like in the C++ and Java languages, classes in Modelica can inherit behavior and data fields from each other. To conclude, Modelica supports imperative, declarative and object-oriented modeling and programming resulting in a complex compilation process that places a high burden on the compiler constructor.

2.2 The OpenModelica Development Environment

OpenModelica is an open source implementation of a Modelica compiler, simulator and development environment for industrial, research, and education purposes. It is developed and supported by an international organization, the Open Source Modelica Consortium (OSMC) [71]. OpenModelica consists of several subsystems including the OpenModelica Compiler (OMC) and other tools such as OMNotebook, OMShell, OMEdit, OMOptim, ModelicaML, etc., that form an environment for creating and simulating Modelica models. The OpenModelica Compiler is easily extensible; a different code generator can for instance be plugged-in at a suitable place. The OpenModelica User Guide [72] states:

- *The short-term goal is to develop an efficient interactive computational environment for the Modelica language, as well as a rather complete implementation of the language. It turns out that with support of appropriate tools and libraries, Modelica is very well suited as a computational language for development and execution of both low level and high level numerical algorithms, e.g. for control system design, solving nonlinear equation systems, or to develop optimization algorithms that are applied to complex applications.*

- *The longer-term goal is to have a complete reference implementation of the Modelica language, including simulation of equation based models and additional facilities in the programming environment, as well as convenient facilities for research and experimentation in language design or other research activities.*

2.3 Mathematical Concepts

Here an overview is provided of some of the mathematical theory that will be used later in the thesis. For more details see for instance [24].

2.3.1 ODE and DAE Representation

Central concepts in the field of equation-based languages are ODE and Differential Algebraic Equation (DAE) systems. A DAE representation can be described as follows.

$$\underline{0} = \underline{f}(t, \underline{\dot{x}}(t), \underline{x}(t), \underline{y}(t), \underline{u}(t), \underline{p})$$

- t time
- $\underline{\dot{x}}(t)$ vector of differentiated state variables

- $\underline{x}(t)$ vector of state variables
- $\underline{y}(t)$ vector of algebraic variables
- $\underline{u}(t)$ vector of input variables
- $\underline{p}$ vector of parameters and/or constants

In an ODE system of equations the vector of state derivatives $\underline{\dot{x}}$ is explicitly computed by equation right-hand sides. In the compilation process, as a middle step we will typically arrive at a DAE system from the transformed Modelica model after all the object-oriented structure has been removed and expansions have been made; see Section 2.6.

2.3.2 ODE and DAE Numerical Integration Methods

This section describes some of the numerical integration methods available for numerically solving an ODE or DAE system.

Euler Integration Method

The simplest method for numerically solving an ODE system is the Euler method described below, where $\underline{x}$ is the state vector, $\underline{u}$ is the input vector, $\underline{p}$ is a vector of parameters and constants, and $\underline{t}$ represents time.

$$\underline{\dot{x}}(t_n) \approx \frac{\underline{x}(t_{n+1}) - \underline{x}(t_n)}{t_{n+1} - t_n} \approx \underline{f}(t_n, \underline{x}(t_n), \underline{u}(t_n), \underline{p})$$

The derivative is approximated as the difference of the state values between two time points divided by the difference in time (this can easily be derived by studying a graph). The above equation gives the following iteration scheme.

$$\underline{x}(t_{n+1}) \approx \underline{x}(t_n) + (t_{n+1} - t_n) \cdot \underline{f}(t_n, \underline{x}(t_n), \underline{u}(t_n), \underline{p})$$

Runge-Kutta Integration Method

The explicit Runge-Kutta numerical integration method is a multi-stage scheme. The generic s-stage explicit Runge-Kutta method is given below, where Δt represents a time step.

$$\begin{aligned}
&\underline{k_1} = \underline{f}(t, \underline{x}(t_n)) \\
&\underline{k_2} = \underline{f}(t + c_2 \cdot \Delta t, \underline{x}(t_n) + \Delta t a_{21} \underline{k_1}) \\
&\underline{k_3} = \underline{f}(t + c_3 \cdot \Delta t, \underline{x}(t_n) + \Delta t (a_{31} \underline{k_1} + a_{32} \underline{k_2})) \\
&\ldots \\
&\underline{k_s} = \underline{f}(t + c_s \cdot \Delta t, \underline{x}(t_n) + \Delta t (a_{s1} \underline{k_1} + \ldots + a_{s,s-1} \underline{k_s})) \\
&\underline{x}(t_{n+1}) = b_1 \underline{k_1} + \ldots + b_s \underline{k_s}
\end{aligned}$$

The values of the constants are given by the Runge-Kutta table below (given a value of s).

0					
c_2	a_{21}				
c_3	a_{31}	a_{32}			
...					
c_s	a_{s1}	a_{s2}	...	$a_{s,s-1}$	
	b_1	b_2	...	b_{s-1}	b_s

We also have the following necessary condition.

$c_j = \sum j = 1i - 1a_{ij}$

DASSL Solver

DASSL stands for Differential Algebraic System Solver. It implements the backward differentiation formulas of orders one through five. [27] The nonlinear system (algebraic loop) at each time-step is solved by Newton's method. This is the main solver used in the OpenModelica compiler. Input to DASSL are systems in DAE form F(t,y,y')=0, where F is a function and y and y' are vectors, moreover, initial values for y and y' are given.

2.3.3 Partial Differential Equations (PDEs)

A differential equation that contains unknown multivariable functions and their partial derivatives is called a PDE. PDEs are used in many different areas such as fluid flow, electrodynamics, heat distribution, elasticity, electrostatics, sound distribution, and quantum mechanics. See [76, 61] and Chapter 10 for more information.

A PDE for the function $u(x_1, \cdots, x_n)$ is an equation of the form:

$$F\left(x_1, \ldots, x_n, u, \frac{\partial u}{\partial x_1}, \ldots, \frac{\partial u}{\partial x_n}, \frac{\partial^2 u}{\partial x_1 \partial x_1}, \ldots, \frac{\partial^2 u}{\partial x_1 \partial x_n}, \ldots\right) = 0.$$

Some different notation can be used:
$u_x = \frac{\partial u}{\partial x}$
and
$u_{xy} = \frac{\partial u}{\partial y \partial x} = \frac{\partial}{\partial y}\left(\frac{\partial u}{\partial x}\right)$

2.3.4 PDE Solving Software

Here an overview is provided of some software and libraries that are available for modeling and solving PDEs.

HiFlow3

HiFlow3 [10] is a multi-purpose finite element software providing powerful tools for efficient and accurate solution of a wide range of problems modeled by PDEs. Based on object-oriented concepts and the full capabilities of C++ the HiFlow3 project follows a modular and generic approach for building efficient parallel numerical solvers. It provides highly capable modules dealing with mesh setup, finite element spaces, degrees of freedom, linear algebra routines, numerical solvers, and output data for visualization. Parallelism - as the basis for high performance simulations on modern computing systems - is introduced at two levels: coarse- grained parallelism by means of distributed grids and distributed data structures, and fine-grained parallelism by means of platform-optimized linear algebra back-ends.

COMSOL Multiphysics

One well-known software for PDE solving and simulation is the COMSOL Multiphysics [2] system from the company COMSOL (previously known as FEMLAB). It is a finite element analysis, solver and simulation software. It can be used for various engineering problems, most notably for entering and solving coupled systems of PDEs. Not only can the PDEs be entered directly but it is enough to enter the so-called weak form; see Chapter 10 for more information. It is possible to interface COMSOL Multiphysics with Matlab.

Maple and MapleSim by MapleSoft

Maple and MapleSim from MapleSoft can be used together in order to model and solve PDEs. Maple is good at symbolic and numerical mathematics, visualization and programming. MapleSim (which supports Modelica) is good at systems engineering, multi-domain modeling and code generation. A four step approach with custom components is as follows.

1. Launch Maple from within MapleSim using the custom component template feature.
2. Enter and develop the PDEs using Maple.
3. Generate a MapleSim component with your PDEs (an automatic discretization is performed, thus no PDEs will remain).
4. Use the newly generated component in MapleSim.

Initial and boundary conditions must also be set. The discretization phase -- going from PDE to ODE system -- can greatly be influenced. See [5, 77, 78] for more information.

2.4 Causalization of Equations

As mentioned earlier in this chapter, systems that consist of a mixture of implicitly formulated algebraic and differential equations are called DAE systems. Converting an implicit DAE system to an equivalent explicit-sorted ODE system is an option (we know in which order and by which equation a variable should be computed). This is an important task for a compiler of an equation-based language. For more details see for instance [24]. Two simple rules can determine which variable to solve from which equation:

- If an equation only has a single unknown variable then that equation should be used to solve for that variable. It could be a variable for which no solving equation has yet been found.
- If an unknown variable only appears in one equation, then use that equation to solve for it.

2.4.1 Sorting Example

f1,...,f5 is used to denote expressions containing variables. Initially all equations are assumed to be in acausal form. This means that the equal sign should be viewed as an equality sign rather than an assignment sign. The structure of an equation system can be captured in a so-called incidence matrix. Such a matrix lists the equations as rows and the unknowns in these equations as columns. In other words if equation number i contains variable number j then entry *(i,j)* in the matrix contains an 1 otherwise 0. The best one can hope for is to be able to transform the incidence matrix into glsblt form, that is a triangular form but with "squares" on the diagonal representing sets of equations that needs to be solved together (algebraic loops).

$f1(z3, z4) = 0$
$f2(z2) = 0$
$f3(z2, z3, z5) = 0$
$f4(z1, z2) = 0$
$f5(z1, z3, z5) = 0$

The above equations will result in the sorted equations with the solved for variables underlined:

$f2(\underline{z2}) = 0$
$f4(\underline{z1}, z2) = 0$
$f3(z2, \underline{z3}, \underline{z5}) = 0$
$f5(z1, \underline{z3}, \underline{z5}) = 0$
$f1(z3, \underline{z4}) = 0$

Note that we have an algebraic loop since *z3* and *z5* have to be solved together. The corresponding matrix transformation is given below. The matching of the variables with an equation to compute that variable is shown in Figure 2.1 and Figure 2.2.

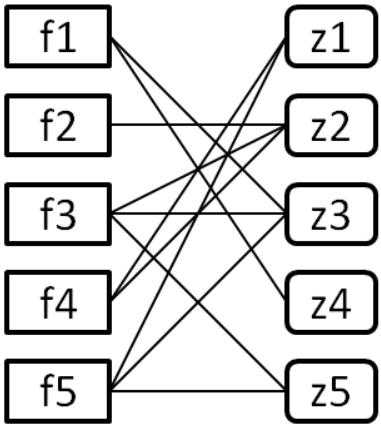


Figure 2.1: *Equation system dependencies before matching.*

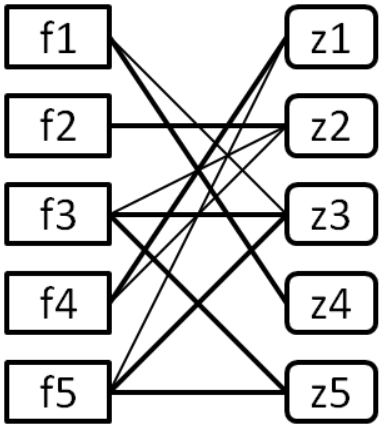


Figure 2.2: *Equation system dependencies after matching.*

$$\begin{pmatrix} & z1 & z2 & z3 & z4 & z5 \\ f1 & 0 & 0 & 1 & 1 & 0 \\ f2 & 0 & 1 & 0 & 0 & 0 \\ f3 & 0 & 1 & 1 & 0 & 1 \\ f4 & 1 & 1 & 0 & 0 & 0 \\ f5 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} & z1 & z2 & z3 & z4 & z5 \\ f1 & 0 & 0 & 1 & \underline{1} & 0 \\ f2 & 0 & \underline{1} & 0 & 0 & 0 \\ f3 & 0 & 1 & 1 & 0 & \underline{1} \\ f4 & \underline{1} & 1 & 0 & 0 & 0 \\ f5 & 1 & 0 & \underline{1} & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} & z2 & z1 & z3 & z5 & z4 \\ f2 & \underline{1} & 0 & 0 & 0 & 0 \\ f4 & 1 & \underline{1} & 0 & 0 & 0 \\ f3 & 1 & 0 & \underline{1} & \underline{1} & 0 \\ f5 & 0 & 1 & \underline{1} & \underline{1} & 0 \\ f1 & 0 & 0 & 1 & 0 & \underline{1} \end{pmatrix}$$

This algorithm is usually divided into two steps: *1. solve matching problem* and *2. find strong components (and sort equations).*

2.4.2 Sorting Example with Modelica Model

An example Modelica model is shown in Listing 2.2.

Listing 2.2: *Modelica model used for sorting example.*

```
model NonExpandedArray1
  Real x;
  Real y;
  Real z;
  Real q;
  Real r;
equation
  2.3232*y + 2.3232*z + 2.3232*q + 2.3232*r = der(x);
  der(y) = 2.3232*x + 2.3232*z + 2.3232*q + 2.3232*r;
  2.3232*x + 2.3232*y + 2.3232*q + 2.3232*r = der(z);
  der(r) = 2.3232*x + 2.3232*y + 2.3232*z + 2.3232*q;
  2.3232*x + 2.3232*y + 2.3232*z + 2.3232*r = der(q);
end NonExpandedArray1;
```

The above model will result in the following matrix.

$$\begin{pmatrix} & x & y & z & q & r \\ eq1 & 1 & 0 & 0 & 0 & 0 \\ eq2 & 0 & 1 & 0 & 0 & 0 \\ eq3 & 0 & 0 & 1 & 0 & 0 \\ eq4 & 0 & 0 & 0 & 0 & 1 \\ eq5 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

In sorted form:

$$\begin{pmatrix} & x & y & z & q & r \\ eq1 & 1 & 0 & 0 & 0 & 0 \\ eq2 & 0 & 1 & 0 & 0 & 0 \\ eq3 & 0 & 0 & 1 & 0 & 0 \\ eq5 & 0 & 0 & 0 & 1 & 0 \\ eq4 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

2.4.3 Conversion to Causal Form in Two Steps

Here the matching algorithm and Tarjan's algorithm for transforming an equation system into causal form are described in more details.

Step 1: Matching Algorithm

Assign each variable to exactly one equation (matching problem), find a variable that is solved in each equation. Then perform the matching algorithm, which is the first part of sorting the equations into Block Lower Triangular (BLT) form. See Listing 2.3.

Listing 2.3: *Matching algorithm pseudo code.*

```
assign(j) := 0, j=1,2,..,n
for <all equations i=1,2,..,n>
    vMark(j) := false, j=1,2,..,n
    eMark(j) := false, j=1,2,..,n
    if not pathFound(i), "singular"
end for

function success = pathFound(i)
    eMark(i) := true
    if <assign(j)=0 for one variable j in equation i> then
        success := true
        assign(j) := i
    else
        success := false
        for <all variable j of equation i
                with vMark(j) = false>
            vMark(j) := true
            success := pathFound(assign(j))
            if success then
                assign(j) := i
                return
            end if
        end for
    end if
end
```

Step 2: Tarjan's Algorithm

Find sets of equations that have to be solved simultaneously. This is the second part of the BLT sorting. It takes the variable assignments and the incidence matrix as input and identifies strong components, i.e. subsystems of equations that are mutually dependent. See Listing 2.4.

Listing 2.4: *Tarjan's algorithm pseudo code.*

```
i = 0 % global variable
number = zeros(n,1) % global variable
lowlink = zeros(n,1) % root of strong component
<empty stack> % stack is global

for w = 1:n
    if number(w) == 0 % call the recursive procedure
        strongConnect(w) % for each non-visited vertex
    end if
end for

procedure strongConnect(v)
    i = i+1
    number(v) = i
    lowlink(v) = i
    <put v on stack>
    for <all w directly reachable from v>
     if number(w) == 0 %(v,w) is a tree arc
          strongConnect(w)
          lowlink(v) = min(lowlink(v),lowlink(w))
     else if number(w)<number(v)%(v,w) frond/cross link
        if <w is on stack>
         lowlink(v) = min(lowlink(v), number(w))
           end if
     end if
    end for

    if lowlink(v) == number(v) %v root of a strong component
     while <w on top of stack satisfies number(w)>=number(v)>
        <delete w from stack and put w in current component>
     end while
    end if
end
```

2.4.4 Algebraic Loops

An algebraic loop is a set of equations that cannot be causalized to explicit form, they need to be solved together using a numerical algorithm. In each iteration of the solver loop this set of equations has to be solved together, i.e. a solver call is made in each iteration. Newton iteration could for instance be used if the equations are nonlinear.

2.5 Compiler Structure

In this section the basic principles behind compilers and compiler construction are outlined. Basically a compiler is a program that reads a program written in a source language and translates it into a program in a target language. Before a program can be run it must[1] be transformed by a compiler into a form that can be executed by a computer[2]. A compiler should also report any errors in the source program that it detects during the translation process. See Figure 2.3 for the various compilation steps. See [18] for more information.

- **Front-end.** The front-end typically includes lexical analysis and parsing. That is, from the initial program code an internal abstract syntax tree is created by collecting groups of characters into tokens (lexical analysis) and building the internal tree (syntax analysis).
- **Middle-part.** The middle-part typically includes semantic analysis (checking for type conflicts, etc.), intermediate code generation and optimization.
- **Code Generation.** Code generation is the process of generating code from the internal representation. Parallel executable code generation is the main focus of this thesis.

2.6 Compilation and Simulation of Modelica Models

The main translation stages of the OpenModelica compiler can be seen in Figure 2.4. The compilation process of Modelica code differs quite a bit from that process for typical imperative programming languages such as C, C++ and Java. This is because Modelica is a complex language that mixes several programming styles and especially due to the fact that it is a declarative equation-based language. Here a brief overview is provided of the compilation process for generating sequential code, as well as the simulation process. For a more detailed description the interested reader is referred to for instance [24]. The Modelica model is first parsed by the parser, making use of a lexer as well; this is a fairly standard procedure. The Modelica model is then elaborated/instantiated by a front-end module that involves among other things, removal of all object-oriented structure, type checking of language constructs, etc. The output from the front-end is a lower level intermediate form, a data structure with lists of equations, variables, functions, algorithm sections, etc. This internal data structure will

[1] Except those programs that are written directly in binary code form.
[2] There are also interpreters that execute a program directly at runtime.

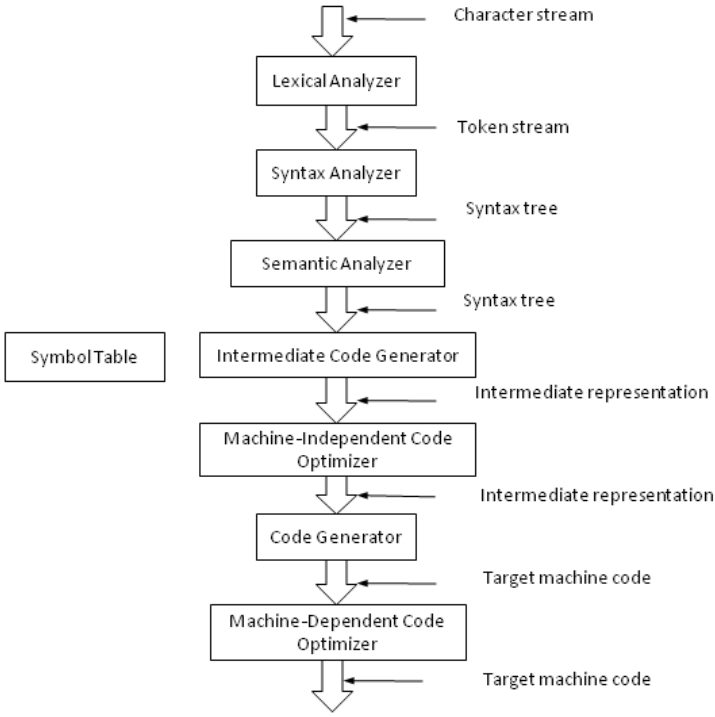


Figure 2.3: *Main compilation phases in a typical compiler.*

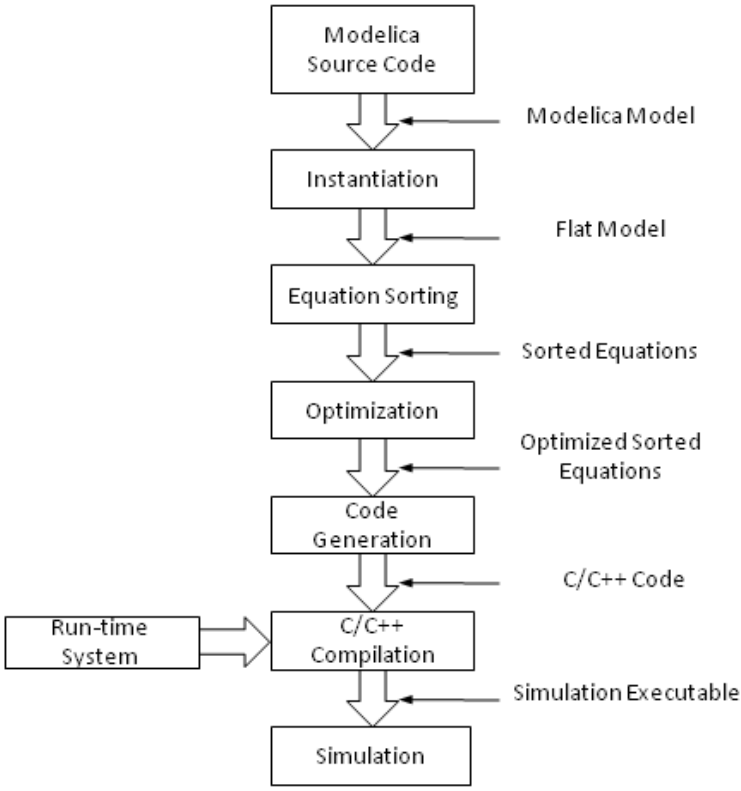


Figure 2.4: *Main compilation phases in a typical Modelica compiler.*

then be used, after several optimization and transformation steps, as a basis for generating the simulation code.

There is a major difference between the handling of time-dependent equations and the handling of time-independent algorithms and functions. Modelica assignments and functions are mapped into assignments and functions respectively in the target language. Regarding the equation handling, several steps are taken. This involves among other things symbolic index reduction (that includes symbolic differentiation) and a topological sorting according to the data flow dependencies between the equations and conversion into single-assignment form. In some cases the result of the equation processing is an ODE system in single-assignment form and in some cases a DAE system is the result. Many Modelica compilers including OpenModelica always reduce the system (through index reduction) to an ODE (in other words index 1). The actual runtime simulation consists mainly of solving this ODE or DAE system using a numerical integration method, such as the ones described earlier (Euler, Runge-Kutta or DASSL). Several C-code files are produced as output from the OpenModelica compiler. These files will be compiled and linked together with a runtime system, which will result in a simulation executable. One of the output files is a source file containing the bulk of the model-specific code, for instance a function for calculating the right-hand side f in the sorted equation system. Another source file contains the compiled Modelica functions. There is also a makefile generated and a file with initial values of the state variables and of constants/parameters along with other settings that can be changed at runtime, such as time step, simulation time, etc.

2.7 Multi-Core Computing

A computing component with two or more independent cores (that can read and execute program instructions) is called a multi-core processor. Many application domains make use of multi-core processors: general-purpose, embedded, network, digital signal processing (DSP), and graphics. The term multi-CPU is used to describe multiple physically separate (on different chips) CPUs. When the number of cores is exceptionally high in a multi-core architecture the terms massively multi-core and many-core are often used.

Multi-core computing has become a hot topic since Moore's law has ended in practice [64]. Increasing the operating frequency is no longer enough to gain performance. There are three factors in play that causes this.

- **The Power Wall**: A factorial increase in operating frequency leads to exponential growth in power consumption.
- **The Memory Wall**: The processor is usually much faster than the memory. [32]

- **The Instruction Level Parallelism Wall**: It is getting more and more difficult to find enough parallelism in a single instruction to keep a processor core busy or to make use of a processor core.

In order to obtain speedup use of multi-core computing is needed. Software developers need to learn how to program these architectures. A lot of research has been carried out on how to abstract away difficulties with the programming of multi-cores. The programming APIs OpenMP and PThreads are both well-known. For for even a higher level of abstraction: see for instance the Liquid Metal Programming Language (LIME) and X10 [64]. One idea is to use some part of the chip for special purposed functions, accelerators. Accelerators well-known today include: GPU, Field Programmable Gate Arrays (FPGA), etc. The main idea is to decompose the application into a CPU-executable part and an accelerator-executable part. [64]

Flynn's classical taxonomy [75] is often used to classify different parallel architectures.

- Multiple Instruction Multiple Data (MIMD)
- Single Instruction Multiple Data (SIMD)
- Multiple Instruction Single Data (MISD)
- Single Instruction Single Data (SISD). A conventional CPU falls into this category.

Several issues arise when designing and developing parallel applications.

- *Partitioning*: The problem at hand needs to be decomposed, in other words, a large number of smaller tasks need to defined in order to arrive at a fine-grained decomposition of the problem.
- *Communication*: The communication needed between the smaller tasks need to be defined.
- *Agglomeration*: It might be justifiable to merge some of the smaller tasks together in order to decrease communication.
- *Mapping*: It needs to be specified where each task should be executed on a specific parallel architecture.

2.8 Graphics Processing Units (GPUs)

This section is based on [35, 69, 70, 6]. In this thesis the main focus is on NVIDIA GPUs since they have represented the most wide-spread GPU architecture during this work and are also available at our research group

used for our measurements. Two other makers of GPU worth mentioning are AMD [1] and ARM [4]. The main goal of GPUs was initially to accelerate the rendering of graphic images in memory frame buffers intended for output to a display, graphic rendering in other words. A GPU is a specialized circuit designed to be used in personal computers, game consoles, workstations, smartphones, and embedded systems. The highly parallel structure of modern GPUs make them more effective than general-purpose CPUs for data-parallel algorithms. The same program is executed for each data element. In a personal computer there are several places where a GPU can be present. It can for instance be located on a video card, or on the motherboard, or in certain CPUs, on the CPU die. Several series of GPU cards have been developed by NVIDIA, the three most notable are mentioned below.

- *The GeForce GPU computing series.* The GeForce 256 was launched in August 1999. In November 2006 the G80 GeForce 8800 was released, which supported several novel innovations: support of C, the single-instruction multiple-thread (SIMT) execution model, shared memory and barrier synchronization for inter-thread communication, a single unified processor that executed vertex, pixel, geometry, computing programs, etc.
- *The Quadro GPU computing series.* The goal with this series of cards was to accelerate digital content creation (DCC) and computed-aided design (CAD).
- *The Tesla GPU computing series.* The Tesla GPU was the first dedicated general purpose GPU.

The appearance of programming frameworks such as CUDA from NVIDIA minimizes the programming effort required to develop high performance applications on these platforms. A whole new field of General-Purpose Computing on Graphics Processing Units (GPGPU) has emerged. Another software platform for GPUs (as well as for other hardware architectures) is OpenCL, which will be described in Section 2.8.3.

2.8.1 The Fermi Architecture

The Fermi architecture is the successor to the Tesla architecture. A scalable array of multi-threaded Streaming Multiprocessors (SMs) is the most notable feature of the architecture. Each of these streaming multiprocessors subsequently contains Scalar Processors (SPs), resulting in a large number of computing cores that can compute a floating point or integer instruction per clock for a thread. Some synchronization between the streaming multiprocessors is possible via the global GPU memory but no formal consistency model exists between them. Thread blocks are distributed by the GigaThread global scheduler to the different streaming multiprocessors. The GPU is connected to the CPU via a host interface. Each scalar processor

contains a fully pipelined integer arithmetic unit (ALU) and floating point unit (FPU). Each streaming multiprocessor has 16 load/store units and four special function units (SFU) that execute transcendental instructions (such as sin, cosine, reciprocal, and square root). The Fermi architecture supports double precision.

Memory Hierarchy

There are several levels of memory available as described below.

- Each scalar processor has a set of registers and accessing these typically requires no extra clock cycles per instruction (except for some special cases).
- Each streaming multiprocessor has an on-chip memory. This on-chip memory is shared and accessible by all the scalar processors on the streaming multiprocessor in question, which greatly reduces off chip traffic by enabling threads within one thread block to interact. The on-chip memory of 64KB can be configured either as 48 KB of shared memory with 16 KB of L1 cache or as 16 KB of shared memory with 48 KB of L1 cache.
- All of the streaming multiprocessors can access a L2 cache.
- The Fermi GPU has 6 GDDR5 DRAM memory of 1 GB each.

2.8.2 CUDA

Compute Unified Device Architecture or CUDA is a parallel programming model and software and platform architecture from NVIDIA [69]. It was developed in order to overcome the challenge with developing application software that transparently scales the parallelism of NVIDIA GPUs but at the same time maintains a low learning curve for programmers familiar with standard programming languages such as C. CUDA comes as a minimal set of extensions to C. CUDA provides several abstractions for data-parallelism and thread parallelism: a hierarchy of thread groups, shared memories, and barrier synchronization. With these abstractions it is possible to partition the problem into coarse-grained subproblems that can be solved independently in parallel. These subproblems can then be further divided into smaller pieces that can be solved cooperatively in parallel as well. The idea is that the runtime system only needs to keep track of the physical processor count. CUDA, as well as the underlying hardware architecture has become more and more powerful and increasingly powerful language support has been added. Some of the CUDA release highlights from [69] are summarized below.

CUDA Programming Model

The parallel capabilities of GPUs are well exposed by the CUDA programming model. Host and device are two central concepts in this programming model. The host is typically a CPU and the device is one or more NVIDIA GPUs. The device operates as a coprocessor to the host and CUDA assumes that the host and device operates separate memories, host memory and device memory. Data transfer between the host and device memories takes place during a program run. All the data that is required for computation by the GPUs is transfered by the host to the device memory via the system bus. Programs start running sequentially on the host and kernels are then launched for execution on the device. CUDA functions, kernels, are similar to C functions in syntax but the big difference is that a kernel, when called, is executed N times in parallel by N different CUDA threads. It is possible to launch a large number of threads to perform the same kernel operation on all available cores at the same time. Each thread operates on different data. The example in Listing 2.5 is taken from the CUDA Programming Guide [6].

Listing 2.5: *CUDA kernel example, taken from [6].*

```
// Kernel definition
__global__ void vecAdd(float* A, float* B, float* C)
{
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}

int main()
{
    // Kernel invocation
    vecAdd<<<1, N>>>(A, B, C);
}
```

The main function is run on the host. The global keyword states that the *vecAdd* function is a kernel function to be run on the device. The special <<< ... >>> construct specifies the number threads and thread blocks to be run for each call (or a execution configuration in the general case). Each of the threads that execute *vecAdd* performs one pair-wise addition and the thread ID for each thread is accessible via the *threadIdx* variable. Threads are organized into thread blocks (which can be organized into grids). The number of threads in a thread block is restricted by the limited memory resources. On the Fermi architecture a thread block may contain up to 512 threads. After each kernel invocation, blocks are dynamically created and scheduled onto multiprocessors efficiently by the hardware. Within a block threads can cooperate among themselves by synchronizing their execution and sharing memory data. Via the *syncthreads* function call it is possible to specify synchronization points. When using this barrier all threads in a block must wait for the other threads to finish.

Thread blocks are further organized into one-dimensional or two-dimensional grids. One important thing to note is that all thread blocks should execute independently, in other words they should be allowed to execute in any order. On the Fermi architecture it is possible to run several kernels concurrently in order to occupy idle streaming multiprocessors; with older architectures only one kernel could run at a time thus resulting in some streaming multiprocessors being idle.

CUDA employs an execution mode called SIMT (single-instruction, multiple-thread) which means that each scalar processor executes one thread with the same instruction. Each scalar thread executes independently with its own instruction address and register state on one scalar processor core on a multiprocessor. On a given multiprocessor the threads are organized into so called warps, which are groups of 32 threads. It is the task of the SIMT unit on a multiprocessor to organize the threads in a thread block into warps, and this organization is always done in the same way with each warp containing threads of consecutive, increasing thread IDs starting at 0. Optimal execution is achieved when all threads of a warp agree on their execution path. If the threads diverge at some point, they are executed in serial and when all paths are complete they converge back to the same execution path.

2.8.3 OpenCL

OpenCL (Open Computing Language) is a framework that has been developed in order to be able to write programs that can be executed across heterogeneous platforms. Such platforms could consist of CPUs, GPUs, Digital Signal Processors (DSPs), and other processors. It has been adopted into graphics card drivers by AMD, ATI and NVIDIA among others. OpenCL consists of, among other things, APIs for defining and controlling the platforms and a language for writing kernels (C-like language). Both task-based and data-based parallelism is possible with OpenCL. OpenCL shares many of its computational interfaces with CUDA and is similar in many ways. [70]

2.8.4 OpenACC

OpenACC (Open Accelerators) [7] is a programming standard that has been developed in order to be able to write programs that can be executed across heterogeneous CPU/GPU systems. The standard has been developed by the companies Cray, CAPS, Nvidida and PGI. OpenACC is similar to OpenMP in the sense that the user can annotate the code with compiler directives. There are also several runtime API functions for device management, etc..

Chapter 3

Previous Research

This chapter describes earlier research that has been conducted mainly at our research group PELAB regarding methods for compiling and simulating equation-based models on multi-core architectures.

There are three main approaches to parallelism with equation-based models.

- **Explicit Parallel Programming Constructs in the Language**: The language is extended with language constructs for expressing parts that should be simulated/executed in parallel. It is up to the application programmer to decide which parts will be executed in parallel. This approach is touched upon in Chapter 8 and in [25].
- **Coarse-Grained Explicit Parallelization Using Components**: The application programmer decides which components of the model can be simulated in parallel. This is described in more details in Section 3.6 below.
- **Automatic (Fine-grained) Parallelization of Equation-Based Models**: It is entirely up to the compiler to make sure that parallel executable simulation code is generated. This is the main approach that is investigated in this thesis.

The automatic parallelization approach can be further divided using the following classification.

- **Parallelism over the method**: With this approach one adopts the numerical solver to exploit parallelism over the method. But this can lead to numerical instability.
- **Parallelism over time**: The goal of this approach is to parallelize the simulation over the simulation time. This method is difficult to implement, since with a continuous time system each new solution of

the system depends on preceding steps, often the immediately preceding step.

- **Parallelism of the system**: With this approach the model equations are parallelized. This means the parallelization of the right-hand side of an ODE system.

3.1 Early Work with Compilation of Mathematical Models to Parallel Executable Code

In [19] certain methods of extracting parallelism from mathematical models are described. In this work searches for parallelism were performed on three levels.

- **Equation System Level**: Equations are gathered into strongly connected components. The goal is to try to identify tightly coupled equation systems within a given problem and separate and solve them independently of each other. A dependency analysis is performed and an equation dependence graph is created using the equations in the ordinary differential equation system as vertices where arcs represent dependencies between equations. From this graph the strongly connected components are extracted. This graph is then transformed into an equation system task graph. A solver is attached to each task in the equation system task graph.
- **Equation Level**: Each equation forms a separate task.
- **Clustered Task Level**: Each sub-expression is viewed as a task. This is the method that has been used extensively in other research work. See Section 3.4 below on task scheduling and clustering.

3.2 Task Scheduling and Clustering Approach

In [21] the method of exploiting parallelism from an equation-based Modelica model via the creation and then the scheduling of a task graph of the equation system was extensively studied.

3.2.1 Task Graphs

A task graph is a directed acyclic graph (DAG) for representing the equation system. There are costs associated with the nodes and edges. It can be described by the following tuple.

$$G = (V, E, c, \tau)$$

- V is the set of vertices (nodes) representing the tasks in the task graph.
- E is the set of edges. An edge $e = (v_1, v_2)$ indicates that node v_1 must be executed before v_2 and send data to v_2.
- $c(e)$ gives the cost of sending the data along an edge $e \in E$.
- $\tau(n)$ gives the execution cost for each node $v \in V$.

An example of a task graph is shown in Figure 3.1.

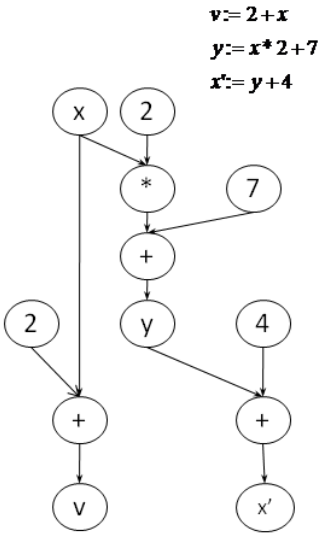


Figure 3.1: *An example of a task graph representation of an equation system.*

The following steps are taken.

- **Building a Task Graph**: A fine-grained task graph is built, at the expression level.
- **Merging**: An algorithm is applied that tries to merge tasks that can be executed together in order to make the graph less fine grained. Replication might also be applied to further reduce execution time.
- **Scheduling**: The fine-grained task graph is then scheduled using a scheduler for a fixed number of computation cores.
- **Code Generation**: Finally code generation is performed. The code generator takes the merged tasks from the last step and generates the executable code.

3.2.2 Modpar

Modpar is the name of the OpenModelica code generation back-end module that was developed in [21] and performs automatic parallelization for a subset of Modelica models. Its use is optional: via flags one can decide whether to generate serial and parallel executable code. The internal structure of Modpar consists of a task graph building module, a task merging module, a scheduling module, and a code generation module.

3.3 Inlined and Distributed Solver Approach

In [55] the work with exploiting parallelism by creating an explicit task graph was continued. A combination of the following three approaches was taken:

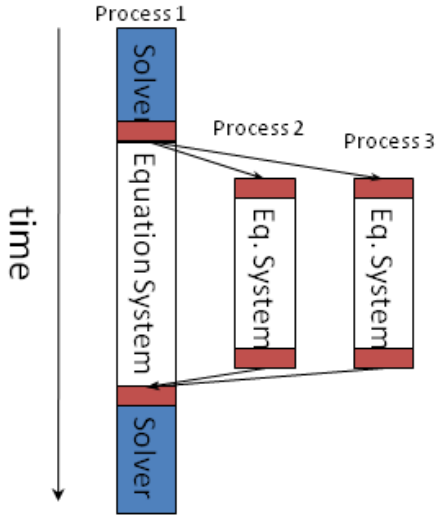


Figure 3.2: *Centralized solver running on one computational core with the equation system distributed over several computational cores.*

- The stage vectors of a Runge-Kutta solver are evaluated in parallel within a single time step. The stage vectors correspond to the various intermediate calculations in Section 2.3.2.
- The evaluation of the right-hand side of the equation system is parallelized.
- A computation pipeline is generated such that processors early in the

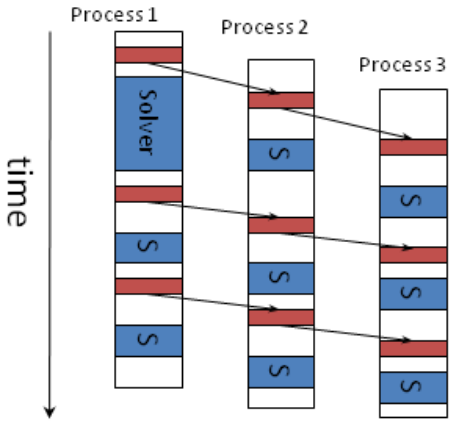


Figure 3.3: *Centralized solver running on several computational cores with an equation system distributed over several computational cores as well.*

pipeline can carry on with subsequent time steps while the end of the pipeline still computes the current time step.

Figure 3.2 shows the traditional approach with a centralized solver and the equation system computed in parallel over several computational cores. Figure 3.3 instead shows the distributed solver approach. Figure 3.4 shows an inlined Runge-Kutta solver, where the computation of the various stages overlap in time.

3.4 Distributed Simulation using Transmission Line Modeling

Technologies based on TLM have been developed for quite some time at Linköping University, for instance in the HOPSAN simulation package developed for mechanical engineering and fluid power applications [44]. It is also used in the SKF TLM-based co-simulation package [16]. Work has also been carried out on introducing distributed simulation based on TLM technology in Modelica [59]. The idea is to let each component in the model solve its own equations, in other words we have a distributed solver approach where each component is numerically isolated from the other components. Each component and each solver can then have its own fixed time step, which produces high robustness and also opens up potential for parallel execution over multi-core platforms. Time delays are introduced between different components to counter the real physical time propagation which produces a physically accurate description of wave propagation in the system.

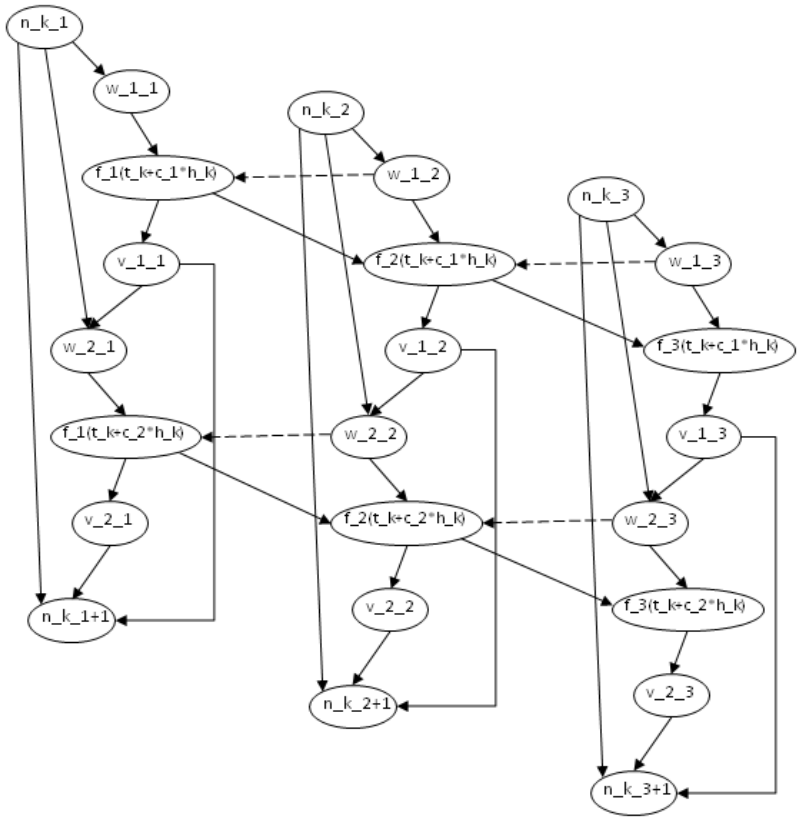


Figure 3.4: *Two-stage inlined Runge-Kutta solver distributed over three computational cores [55].*

Mathematically, a transmission line can be described in the frequency domain by the four-pole equation. Transmission line modeling is illustrated in Figure 3.5.

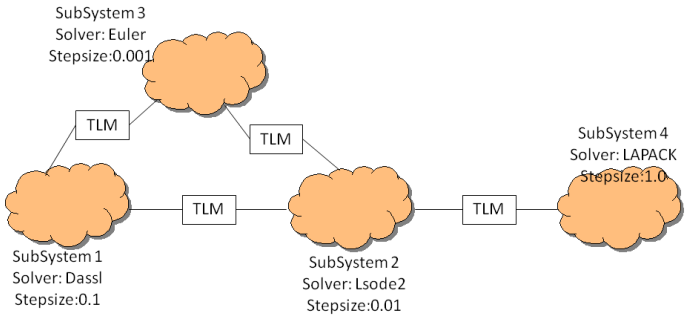


Figure 3.5: *Transmission Line Modeling (TLM) with different solvers and step sizes for different parts of the model [59].*

3.5 PDE Modeling with Modelica

In [34] a Modelica library is described with basic building blocks for solving one-dimensional PDE with spatial discretizations based on the method of lines or finite volumes. Although this approach is attractive due to its simplicity, it is not clear how it could be extended to higher dimensions, without increasing the complexity significantly. Another approach is described in [93], which extends the modeling language with primitives for geometry description and boundary/initial conditions, and uses an external pre-processing tool to convert the PDE model to a DAE based on the method of lines. In both of these two works, the PDE system is expanded early on in the compilation process. In this way, important information about the PDE structure is lost, information that could have been used for mesh refinement and adjustment of the runtime solver. Another similar option is to use the commercial MapleSim environment [5]: It means writing the PDEs in a Maple component, to export this component to DAE form using a discretization scheme and using the resulting component in MapleSim, which supports the Modelica language. An overview of how to use Maple and MapleSim together for PDE modeling can be found in [78]. This method again has the same drawback, arising from the loss of information regarding the original model. See Chapter 10 for more.

3.6 Related Research in other Research Groups

For work on parallel differential equation solver implementations in a broader context than Modelica see for instance [84, 62, 40, 83].

In [90, 91, 36] a different and more complete implementation of the QSS method for the OpenModelica compiler is described. This work interfaces the OpenModelica compiler and enables the automatic simulation of large-scale models using QSS and the PowerDEVS runtime system. The interface allows the user to simulate a Modelica model even without any knowledge of DEVS and/or QSS methods. In this work discontinuous systems are also handled, something that is not dealt with in the work in Chapter 5.

SUNDIALS from the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory has been *"implemented with the goal of providing robust time integrators and nonlinear solvers that can easily be incorporated into existing codes"* [81]. PVODE is included in the SUNDIALS package for equation solving on parallel architectures. Interfacing this solver with OpenModelica could be a promising subject of future work.

In the HPC-OpenModelica project [3] implementation of parallel computing capabilities is being performed in OpenModelica. The project has three partners: Bosch Rexroth (BR), ITI GmbH Dresden (ITI) and TU Dresden. The first part of the project is a precise analysis of OMC and simulation runtime systems in order to reveal slow algorithms or inefficient memory management. The second part of the project is the implementation of the automatic parallelization in OMC. A goal of the project is to be able to efficiently simulate heavy machinery. The approach with task graph parallelization as well as parallel time integration are used in the project.

Part II

Parallel Simulation of Equation-Based Models on Graphics Processing Units

Chapter 4

Simulation of Equation-Based Models on the CELL BE Processor Architecture

This chapter is based on Publication 1 that mainly presented two areas: a summary of our previous approaches (work in our group) of extracting parallelism from equation-based models (this was covered somewhat in Chapter 2 of this thesis) and an investigation of using the STI[1] Cell BE architecture [23] for simulation of equation-based Modelica models. A prototype implementation of the parallelization approaches with task graph creation and scheduling for the Cell BE processor architecture was presented for the purpose of demonstration and feasibility. It was a hard-coded implementation of an embarrassingly parallel flexible shaft model. The generated parallel C/C++ code (from the OpenModelica compiler) was manually re-targeted to the Cell BE processor architecture. Some speedup was gained but no further work has been carried out since then. This work is included in this thesis since it holds some relevance regarding the work on generating code for NVIDIA GPUs.

This chapter is organized as follows. The chapter begins with a description the Cell BE processor architecture. We then discuss the above-mentioned hard-coded implementation. We provide the measurements that were given in the paper. Finally, we conclude with a discussion section where we address the measurement results, suitability of the Cell BE architecture for simulation of equation-based Modelica models and our implementation.

[1] An alliance between Sony, Toshiba, and IBM

4.1 The Cell BE Processor Architecture

The Cell BE Architecture is a single-chip multiprocessor consisting of one Power Processor Element (PPE) and 8 so-called Synergistic Processor Elements (SPE). The PPE runs the top level thread and coordinates the SPEs, which are optimized for running compute-intensive applications. Each SPE has its own small local on-chip memory for both code and data but the SPEs and PPE do not share on-chip memory. To transfer data between the main memory and the SPEs and between the different SPEs, DMA transfers (which can run asynchronously) are used. In conclusion, the main features of the Cell BE processor architecture are the following.

- One main 64-bit PPE processor (PowerPC) Power Processor Element, 2 hardware threads good at control tasks, task switching and OS-level code and SIMD unit VMX
- 8 SPE processors (RISC with 128bit SIMD operations). Good at compute-intensive tasks, small local memory 256KB (code and data)
- No direct access to main memory, DMA transfers used (for SPEs only)
- Internal communication: Signals, mailboxes interface to main memory (off chip, 512 MB and more)

4.2 Implementation

Here the hand-coded implementation for demonstration and feasibility studies is described. The equations converted to statement form for computation are divided into 6 different subsets and in the PPE 6, threads are created and loaded with 6 different program handlers. The PPE then uses the mailbox facility to send out a pointer to a control block in main memory to each SPE which is then used to transfer a copy of the control block via DMA to its local store. The SPEs will use the pointers in the control block to fetch and store data from the main storage, and when sending and synchronizing between different SPEs. Next the initial data is read by each SPE for the different vectors *x'* (state variable derivatives), x (state variables), u (input variables) and p (constants and parameters) into local store. Then comes the actual iteration of the solver (that runs on the PPE) in N time steps where new values of the state variables *x(t+h)* are calculated at each step (the values *x(t+h)* associated with each SPE). DMA transfers are used if SPEs need to send and receive data between them. Data is sent back from the SPEs to the main memory buffer at the end of each iteration step (or at the end of some iteration steps, in a periodic manner). After all threads have finished the PPE will write this data to a results file. In order to exploit the full performance potential of the Cell BE processor, the SIMD instructions of the SPEs need to be leveraged (but only inter-SPE parallelism and DMA

parallelism was utilized). This requires vectorization of the (generated) code (or for instance keeping the array equations unexpanded throughout the compilation process). DMA transfers have the advantage that an SPE in some cases can continue to execute while the transfer is underway. For code from large examples, data distribution across a cluster of several Cell BE processors is needed. Another alternative is a time consuming overlay of multiple program modules in the SPE local store.

Listing 4.1: *SimpleShaft Modelica model.*

```
model ShaftElement
  import Modelica.Mechanics.Rotational;
  extends Rotational.Interfaces.TwoFlanges;
  Rotational.Inertia inertia1;
  SpringDamperNL springDamper1(c=5,d=0.11);
equation
  connect(inertia1.flange_b,springDamper1.flange_a);
  connect(inertia1.flange_a, flange_a);
  connect(springDamper1.flange_b,flange_b);
end ShaftElement;

model FlexibleShaft
  import Modelica.Mechanics.Rotational;
  extends Rotational.Interfaces.TwoFlanges;
  parameter Integer n(min=1) = 3;
  ShaftElement shaft[n];
equation
  for i in 2:n loop
    connect(shaft[i-1].flange_b,shaft[i].flange_a);
  end for;
  connect(shaft[1].flange_a,flange_a);
  connect(shaft[n].flange_b,flange_b);
end FlexibleShaft;

model ShaftTest
  FlexibleShaft shaft(n=100);
  Modelica.Mechanics.Rotational.Torque src;
  Modelica.Blocks.Sources.Step c;
equation
  connect(shaft.flange_a,src.flange_b);
  connect(c.y,src.tau);
end ShaftTest;
```

4.3 Measurements

The Modelica model used for the measurements is shown in Listing 4.1. Running the entire flexible shaft example, 100000 iteration steps on the Cell BE processor (with 6 SPUs as mentioned earlier) took about 31.4 seconds (from start of the PPU main function to the end of the PPU main function). The final writing of the results to results files is not included in this measurement. The relative speedup is shown in Figure 4.1, compared

Table 4.1: *Measurements of running the flexible shaft model on six threads (on Cell BE).*

Thread	$T_{tot}(s)$	$T_{DMA}(s)$	% DMA
1	31.39	2.49	7.9%
2	31.39	12.28	39.1%
3	31.39	11.10	35.4%
4	31.39	12.25	39.0%
5	31.39	11.04	35.2%
6	31.38	4.39	13.9%

to running with 6 SPUs to one SPU. However, it is not straightforward to define relative speedup since the Cell BE architecture is a heterogeneous architecture, and this measurement should be taken with some caution.

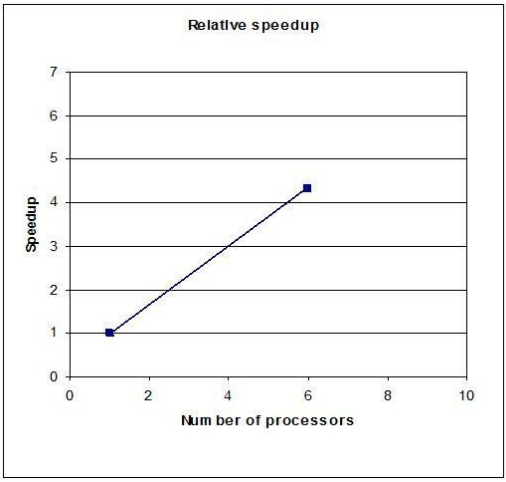


Figure 4.1: *Relative speedup of running the flexible shaft model on the Cell BE architecture with 6 threads.*

4.4 Discussion

From Table 4.1 several things can be concluded. Threads 2 to 5 spent more than a third of the execution time on DMA transfers, but threads 1 and 6 did not spend significant time performing DMA transfers. The total execution time of about 31.4 seconds is not good. On a 4 core Intel Xeon with hyper-threading the same example took 11.35 seconds (using one core) and on SGI Altix 3700 Bx2 it took 22.59 seconds (using one processor). Another issue is the fact that on our Cell BE version double precision calculations take about 7 times more time than single precision (this was improved in the

next version of the Cell BE processor). In conclusion, this implementation was crude and it seems that the memory transfers prevent any performance gains. A model with larger and heavier calculations at each time step might have worked better. This will be discussed further in Chapter 12 of this thesis.

Chapter 5

Simulation of Equation-Based Models with Quantized State Systems on Graphics Processing Units

This chapter discusses the use of the Quantized State Systems (QSS) [48] simulation algorithm as a way of exploiting parallelism for simulating Modelica models on NVIDIA GPUs. This chapter is mainly based on Publication 2. In that paper a method was described that made it possible to translate a restricted class of Modelica models to parallel QSS-based simulation code. The OpenModelica compiler was extended with a back-end module that automatically generates CUDA-based simulation code. Some performance measurements of an example model on the Tesla architecture [82] was performed. The QSS method replaces classical time slicing, i.e. quantization of the time variable, by a quantization of the state variables in an ODE system. This is an alternative way for numerically solving ODE systems of equations. The QSS integration method is a Discrete Event System (DEVS) method. However, no further work on this implementation was done after the paper was published. The goal was two-fold: to investigate the possibility of parallelization of the QSS algorithm per se together with the chosen architecture, and to investigate the parallel performance of the QSS integration method via automatically generated CUDA code. It was first suggested in [48] that QSS could be suitable for parallel execution. The set of models have been restricted to only a subset of valid Modelica models: continuous time, time-invariant systems (with no events); index-1 DAE; initial values of

states and values of parameters known at compile time, and inserted into the generated code as numbers; and no implicit systems of nonlinear equations to be solved numerically.

This chapter is organized as follows. A description is provided of the restricted set of Modelica models that code is generated from. The quantized state systems numerical solution method is then introduced. The following section contains an explanation of why the QSS method is suitable for parallel execution. Then a summary of the implementation work that was described in the paper follows. The chapter is then continued with a measurements section and finally concluded with a discussion section. Neither the NVIDIA GPU architecture nor the CUDA programming model will be covered here, although this was covered in the paper, since we have already presented this in Chapter 2 of this thesis.

Note that a more extensive and more recent work on simulating Modelica models with the QSS method can be found in [36]. That PhD thesis contains several approaches. First an algorithm was developed for extracting all the necessary information from a hybrid dynamical model and it could then be simulated within the DEVS simulation framework. The implementation work was performed in OpenModelica and the PowerDEVS environment was used as well. In the next approach an automated translation of a Modelica model to the μ-Modelica specification was implemented in OpenModelica, without using PowerDEVS. A stand-alone QSS-solver can then be used and this was tested with two large, hybrid, representative smart-grid models and compared with classical solvers. Finally a generic way to model and solve the load-balancing problem of a parallel QSS simulation was presented and analyzed.

5.1 Quantized State Systems (QSS)

The QSS numerical solution method was introduced in [24], where the authors suggested that it could be suitable for parallel execution. Here the main characteristics of this method are described. Note that in this work we are using the Ziegler DEVS (library) approach, and it is possible to discretize QSS without using the specific Ziegler DEVS library approach as described in [36].

Time slicing is by far the most commonly used approach for numerically solving a set of ODEs on a digital computer. But instead of discretization of the time, the state variable values can be discretized. This is what is done in the QSS method. QSS is actually a set of algorithms that have in common that they are intended to discretize the state variables and solve the system of equations. The classical approach is as follows.

Given that the state value at time t_k is equal to x, what is the state value at time $t_{k+1} = t_k + \Delta t$?

With QSS one instead tries to answer the following question.

Given that the state has a value of x_k at time t, what is the earliest time instant, at which the state assumes a value of $x_{k+h} = x_k + \Delta x$?

In other words, a QSS algorithm calculates the earliest time instant at which this state variable shall reach either the next higher or the next lower discrete level in a set of values. The currently available QSS algorithms are not yet as sophisticated as the classical numerical ODE solvers since the QSS method is relatively new.
A limited boundary error exists when transforming a continuous time system into a discrete one, i.e.:

$$\dot{x} = f(x, u) \longrightarrow \dot{x} = f(q, u)$$

Here the state vector x becomes a quantized state vector q where state values are in the corresponding set. The quantized state vector is a vector of discretized states and each state varies according to an hysteretic quantization function. When simulating a system with the QSS algorithm a variable-step technique is applied. The algorithm is asynchronous: it adjusts the time instant at which the state variable is re-evaluated to the speed of change of that state variable. In other words, different state variables will update their values independently of each other. This approach can be seen in Figure 5.1. Each state variable has an associated DEVS subsystem. The dependency between state variables and derivative equations decides the interconnection between subsystems. When the hysteretic quantization threshold is reached the events of the DEVS model are fired.

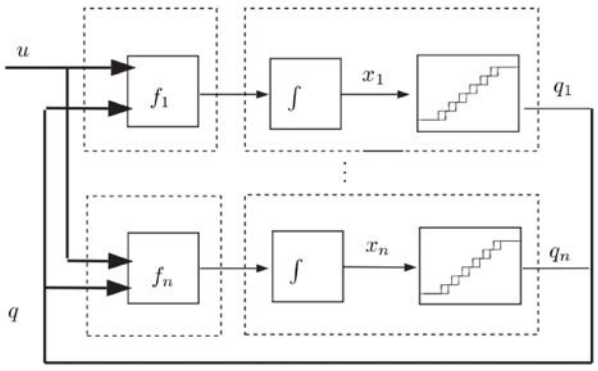


Figure 5.1: *Updating state variable values with the QSS method [60].*

The simulation method consists of three main steps:

- Search the DEVS subsystem that is the next to perform an internal transition, according to its internal time and to the derivative value. Suppose that the event time is t_{next} and the associated state variable is x_i. If $t_{next} > t_{inputevent}$ then set $t_{next} = t_{inputevent}$ and perform the input change.
- Advance the simulation time from current time to t_{next} and execute the internal transition function of the model associated to x_i or the input change associated to u_i.
- Propagate the new output event produced by the transition to the connected state variable DEVS models.

5.2 Restricted Set of Modelica Models

The set of models has been restricted to a subset. This is mainly because the Modelica language is used to describe many different classes of systems and it was deemed suitable to limit work in order to obtain results within reasonable time. The restricted set of models is described below.

- Continuous time, time-invariant systems (with no events)
- Index-1 DAE (If the index is greater than 1 the index reduction algorithm should be used before processing the model)
- Initial values of states and values of parameters known at compile time, and inserted into the generated code as numbers
- No implicit systems of nonlinear equations to be solved numerically

A constant QSS numerical integration step was used, unchanged for all the state variables but a different quantization step can be used for each state variable, with minor modification to the code.

5.3 Implementation

It was noted in [48] that *"due to the asynchronous behavior, the DEVS models can be implemented in parallel in a very easy and efficient way"*. The QSS algorithm is naturally amenable to be parallelized due to the possibility of separately computing the derivatives of the state variables and the time events schedule. The method will first be described in general terms and then the actual code that is generated is discussed. A Modelica model and the generated CUDA code can be found in Appendix A.

- The derivatives of the state variables are computed using the model equations (assuming that the initial values of the state variables are known). MIMD execution model.

- The time of the new event is calculated. Since all the computing threads execute the same code on different data SIMD parallelism can be completely exploited.

- A time advance is made. If the values of one of the inputs changes or if one of the bounds of the quantized state function is reached, a new event is registered. Each value of the state variables is then re-computed and the quantized integrators are updated. Since the same code executes on different data elements. SIMD parallelism can be exploited here as well.

Good performance can be achieved via the definition of a state vector array since the NVIDIA Tesla architecture requires all the computing cores in the same group to compute the same instruction at the same time. Each derivative state value is calculated within a separate thread but the SIMD style code is not performing well here since the code to compute such values are different for each state variable. Instead a MIMD style code is needed.

When all threads finish, the derivative values have been calculated. The next time event for each variable is then calculated by the threads, executing the same portion of code. Since every thread executes the same code on a different data portion, this part of the code should be able to execute in SIMD fashion. Finally, the next time event of the QSS simulation is determined and processed.

In conclusion, the system advancement part takes full advantage of the hardware capabilities but the derivative calculation part of the code is not completely parallel with this approach.

The OpenModelica compiler was extended with a back-end module that generates QSS, CUDA-based simulation code. The module took the equation system immediately after the matching and index reduction phases and generated the CUDA code.

Figure 5.2 shows the internal call chain in the compiler for obtaining CUDA code. The newly added module is the GPUpar module; the other phases were described in the background chapter of this thesis. In the GPUpar module different kernel and header files are generated. As input GPUpar takes the DAELow form as well as the BLT matrix and strong components information from the equation sorting phase. Some data has to be computed from the DAELow form in order to generate the model-specific files.

- For each state variable a derivative function is generated. This function contains the algorithm for the time derivative computation. If there is a dependency with other equations they are also added to the derivative function (in statement form of course).

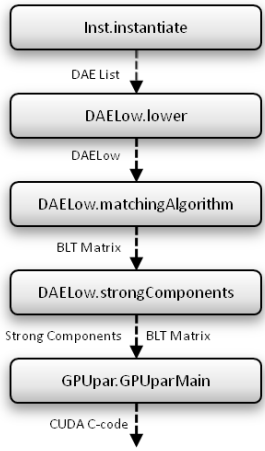


Figure 5.2: *Internal call chain in the OpenModelica compiler to obtain CUDA code.*

- For each output variable an output function for computing the output values is generated. If there is a dependency with other equations they are also added to this function (in statement form).
- From the list of variables in the DAELow form, initial variable (and parameter) values must be gathered

When additional equations that depend on the single derivative/output equation are present, we get a subtree with the main equation as the root node. An existing function (DAELow.markStateEquations) was slightly modified to handle this problem. The equations are sorted by using information obtained in the sorting phase. All the equations are also brought into solved form (explicit form) by calling Exp.solve. By traversing the list of variables, the initial values are gathered in a rather straight-forward manner. To solve the problem with the variables being stored in different arrays in the generated code - xd (derivatives), x (state variables), y (output variables), u (input variables) and p (parameters), an environment is created at the beginning of the GPUpar module that contains a mapping between each variable/parameter and the array name plus the index number in this array. In order to find the correct array and index to print for a given variable, this environment is then used when the CUDA C-code is generated.

5.4 Measurements

The test model used for the measurements is depicted in Figure 5.3. The circuit consists of a generator voltage that comprises N ..1 different branches.

Table 5.1: *Execution times and speedup with the GeForce 8600.*

	parallel [s]	sequential [s]	speedup
8 state variables	6.26	7.07	1.129
16 state variables	8.04	10.27	1.277
32 state variables	27.02	45.55	1.685
64 state variables	103.18	507.38	4.917

Table 5.2: *Execution times and speedup with the C1060 using one cluster for the derivatives of the state variables calculation.*

	parallel [s]	sequential [s]	speedup
8 state variables	1.06	5.71	5.387
16 state variables	8.11	9.07	1.118
32 state variables	22.91	47.30	2.065
64 state variables	208.76	711.00	3.406

Each of them is composed of a resistor with resistance R=N and of a capacitor with capacitance C=N. The last branch is made up of the resistor with resistance R=N and a capacitor with capacitance C together with a resistor with resistance R in parallel. The only input of the system, in the following referred as u, is the voltage V, that is supposed to be a square wave with rise time and fall time of 1s and voltage of 1 volt.

The initial time is obtained at the beginning of the program, before memory allocation. The end time is measured when the simulation stops with the same function call, and the difference between them is divided by a *CLOCKSPERSEC* constant to compare architectures with different clock periods. The parallel algorithm is compared to the sequential one, where a single thread is executed on the graphic card and takes care of the computation sequentially. The results with the NVIDIA Tesla GeForce 8600 can be seen in Table 5.1. Table 5.2 shows the results with the NVIDIA Tesla C1060 when just one cluster is used to compute the derivative values, while Table 5.3 reports the data with the same graphic card when all the available clusters are used. In Figure 5.4 a summary of the obtained speedup values is presented.

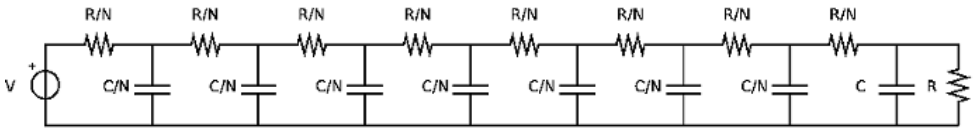


Figure 5.3: *Simple circuit used as test model.*

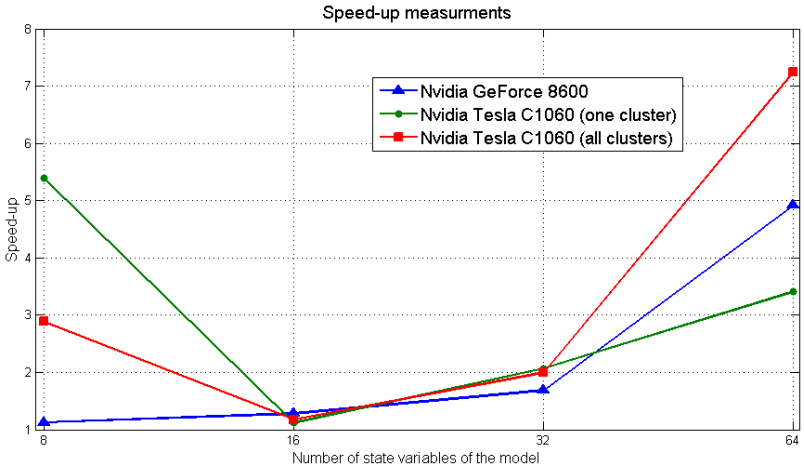


Figure 5.4: *Speedup measurements: comparison between a GeForce 8600 and an NVIDIA Tesla C1060 with increasing number of state variables.*

Table 5.3: *Execution times and speedup with the C1060 using all the clusters for the derivatives of the state variables calculation.*

	parallel [s]	sequential [s]	speedup
8 state variables	1.98	5.71	2.884
16 state variables	7.73	9.07	1.173
32 state variables	23.73	47.30	1.993
64 state variables	98.09	711.00	7.248

5.5 Discussion

In this work methods of simulating equation-based models using the QSS method with NVIDIA GPUs were investigated, which was a first attempt at this, with some minor success. However, it is worth noting that the work done in [90] and [91] is more promising for the future. The simulations reported in this paper were very slow, compared to simulating the same model using a normal CPU, for instance. This has to do with the memory latency of copying data back and forth to the GPU device.

A problem with 256 state variables requires more than $(5 * 64 + 1 * 32) * 256/8[Bytes] = 11[MB]$, while a case with 1024 state variables would require 43[MB]. The side effects of the diverging branches have to be further reduced.

Chapter 6

Simulation of Equation-Based Models on Graphics Processing Units Utilizing Task Graph Creation

This chapter is mainly based on Publication 4, which is based on [73]. This paper demonstrated that it is possible to automatically generate parallel simulation code for pure continuous-time models that can be reduced to an ordinary differential equation system without algebraic loops, and where the initial values of all variables and parameters are known at compile time. A back-end module was implemented for the OpenModelica compiler and measurements were performed; a relative speedup of 4.6 was obtained for one of the models. The method for finding parallelism in this work is by creating a large task graph from the equation system, merging this coarse-grained task graph into larger tasks and then scheduling this task graph for execution with NVIDIA GPUs. Methods of efficiently using the available memory space on the architecture are also presented, which is an important issue that is further discussed in Chapter 12. Other ways of using the CUDA architecture more efficiently are also discussed.

6.1 Case Study

The model is taken from [37](page 584). The model models the one-dimensional wave equation that is given by a partial differential equation of the following form:

$$\frac{\partial^2 p}{\partial t^2} = c^2 \frac{\partial^2 p}{\partial x^2}$$

Here $p = p(x,t)$ is a function of both space and time and we consider a duct of length 10 where we let $-5 \leq x \leq 5$. We discretize the problem in the spatial dimension and approximate the spatial derivatives using the difference approximation:

$$\frac{\partial^2 p}{\partial t^2} = c^2 \frac{p_{i-1} + p_{i+1} - 2p_i}{\delta x^2}$$

where $p_i = p(x_1 + (i-1)\delta x)$ on an equidistant grid and δx is a small change in distance. The Modelica model in Listing 6.1 is obtained.

Listing 6.1: *Modelica model WaveEquationSample.*

```
model WaveEquationSample
  parameter Real L = 10 "Length of duct";
  parameter Integer n = 30 "Number of sections";
  parameter Real dL = L/n "Section length";
  parameter Real c = 1;
  Real[n] p(start = fill (0,n));
  Real[n] dp(start = fill (0,n));
equation
  p[1] = exp(-(-L/2)^2);
  p[n] = exp(-(L/2)^2);
  dp = der(p);
  for i in 2:n-1 loop
    der(dp[i])=c^2*(p[i+1]-2*p[i]+p[i-1])/dL^2;
  end for;
end WaveEquationSample;
```

6.2 Implementation

The general compilation and simulation process of Modelica models is described in Chapter 2. In this implementation, some changes to the normal compilation process are made. This is depicted in Figure 6.1. A new module was implemented as a small MetaModelica package that exports a task graph to an external C++ module, which then manipulates the task graph and finally generates the CUDA code. This module is invoked with a sorted equation system as input. A task graph is then created from the equation system. A task graph is described in chapter 3. Rough approximations of costs for the tasks are used, with the cost of unary and binary operations set to 1 and the cost of special functions set to 4 (which should reflect the

fact that a streaming multiprocessor has eight scalar processors but only two special function units). The cost of communication is set to 100, which should reflect the latencies to the global memory. After the task graph has been created a merging algorithm is applied to it. This merging algorithm is further described in [73].

The merged task graph is then sent to the scheduling phase. We need to determine the order in which the tasks should be executed and whether they should be executed in parallel on different streaming multiprocessors. This is a two-step process, in the first step the nodes in the merged task graph are scheduled with the so-called critical path algorithm and then the tasks in each node are scheduled. In one node the tasks are scheduled using a first-in, first-out queue with the tasks to be scheduled. An example of this approach can be seen in Figure 6.2. There is also a third approach used by the scheduler. The scheduler tries to find nodes that are operationally equivalent to other nodes. If they are operationally equivalent they are scheduled to be executed in parallel on the same streaming multiprocessor (SIMD style execution). A processor schedule is the result of the scheduling, an example can be seen in Figure 6.3. From the figure we can see that the processor schedule contains execution paths and execution path lists where an execution path is a list of task executed in order. The goal is to execute one execution path on one streaming multiprocessor and in one streaming multiprocessor we should hopefully (if there are several execution paths) execute in SIMD mode. We can run several blocks in parallel by using the following technique (remember that we do not know the order in which the different blocks are going to execute); we never execute more blocks than there are streaming multiprocessors (to avoid dead-locks) and we synchronize via the global memory. If it is the case that a task has a dependency with a task that is scheduled on another processor, the scheduler inserts signals and locks into the schedule and determines which data should be sent where. In addition to this, special execution paths for communication are inserted into the schedule. After that, code is generated from the schedule. This is done by iterating through the processors' schedule one processor at a time one execution path at a time. Memory coalescing is used to reduce long off-chip DRAM latencies. 16 variables are read at a time from the device memory (the size of a coalesced read of 32-bit variables). These variables are then moved to where they should be in the shared memory on a streaming multiprocessor.

6.3 Runtime Code and Generated Code

The actual simulation function is shown in the code Listing 6.2. A fourth-order Runge-Kutta method was used, both for the GPU-based implementation as well as the normal CPU implementation. In the code below there is a main for-loop that corresponds to the main simulation loop. The function

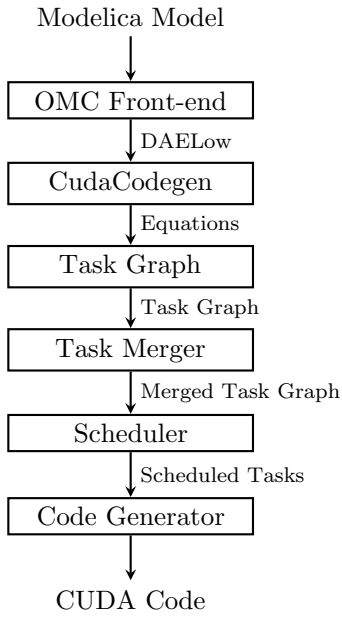


Figure 6.1: *The process of compiling a Modelica model to CUDA code.*

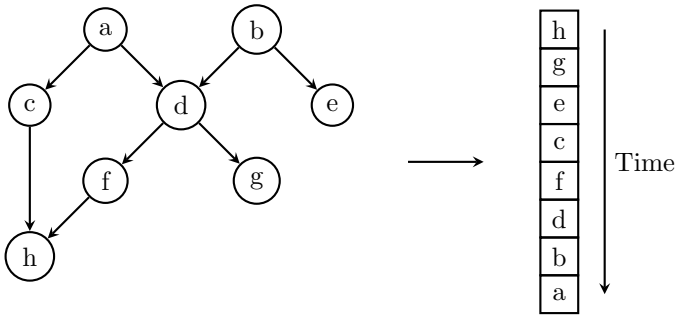


Figure 6.2: *An example of the task scheduling algorithm.*

execute_tasks corresponds to the computation of one of the stages in the Runge-Kutta solver scheme. This is done in parallel by launching kernels for the GPU. The various *step_and_increment* functions handle advancing the step and adding vectors together. Note that we have four calls to *execute_tasks* since we have a fourth-order Runge-Kutta solver scheme. In each iteration, device-to-host copying is performed for the vectors with state variables, which is time-consuming.

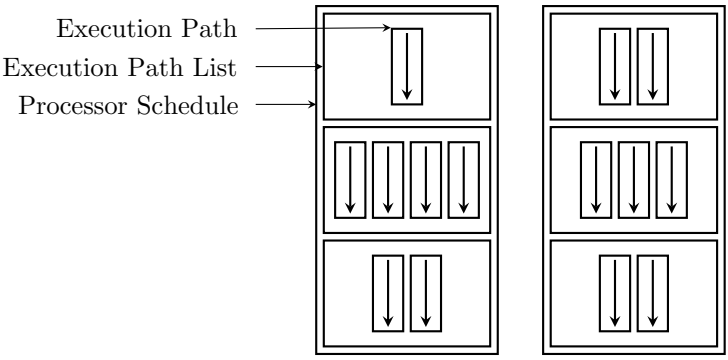


Figure 6.3: *An example of a schedule for two processors.*

Listing 6.2: *Main CUDA simulation loop, based on a 4-stage Runge-Kutta solver.*

```
//Determine the size of the shared memory needed.
int shmem_size = 100 * sizeof(real);

for(int step = 0; step < steps; ++step)
{
  //Move the pointers of the result arrays forward.
  r_dx += DERIVATIVES;
  r_x += STATES;
  r_y += ALGEBRAICS;

  //Execute the tasks, call integration kernel.
  execute_tasks<<<7, 20, shmem_size>>>(d_dx, d_x,
    d_y, d_c, d_l, t);
  step_and_increment1<<<2, 32>>>(d_x, d_old_x,
    d_dx, d_k, half_h);

  //Increment the time by half a time step.
  t += half_h;

  //Do two more steps of the RK4 method.
  execute_tasks<<<7, 20, shmem_size>>>(d_dx, d_x,
    d_y, d_c, d_l, t);
  step_and_increment2<<<2, 32>>>(d_x, d_old_x,
    d_dx, d_k, half_h);
  execute_tasks<<<7, 20, shmem_size>>>(d_dx, d_x,
    d_y, d_c, d_l, t);
  step_and_increment3<<<2, 32>>>(d_x, d_old_x,
    d_dx, d_k, h);

  //Increment the time again with half a time step.
  t += half_h;

  //Do the final integration.
  execute_tasks<<<7, 20, shmem_size>>>(d_dx, d_x,
    d_y, d_c, d_l, t);
  step_and_integrate<<<2, 32>>>(d_x, d_old_x,
    d_dx, d_k, h_div_6);
```

```
  //Save the new values.
  cudaMemcpy(r_x, d_x, STATES * sizeof(real),
  cudaMemcpyDeviceToHost);
  cudaMemcpy(r_dx, d_dx, DERIVATIVES * sizeof(real),
  cudaMemcpyDeviceToHost);
  cudaMemcpy(r_y, d_y, ALGEBRAICS * sizeof(real),
  cudaMemcpyDeviceToHost);
}
```

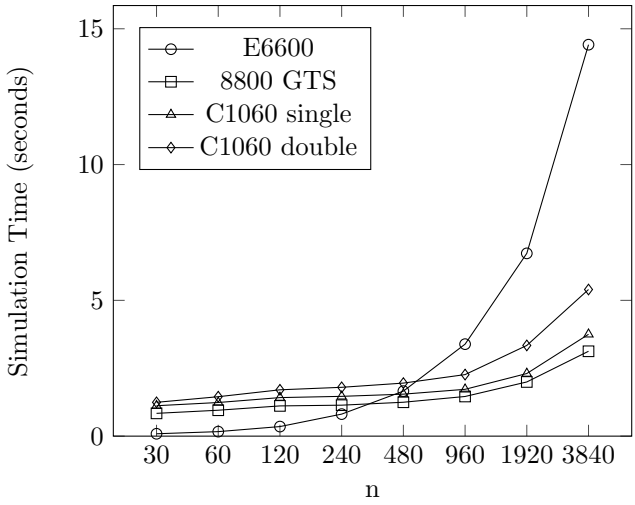


Figure 6.4: *Execution time for the WaveEquationSample Modelica model as a function of the number of sections.*

6.4 Measurements

The specifications for the two GPU cards used can be seen in Table 6.1. The CPU used was an Intel Core 2 Duo E6600 with 2.4 GHz clock frequency. Note though that only one core was used. Table 6.2 shows seconds spent in different parts of the simulation of the test model WaveEquationSample and the graph in Figure 6.4 shows the execution time for the sample model as a function of the number of sections.

Table 6.1: *Specifications for the used GPUs.*

	GeForce 8800 GTS	Tesla C1060
Streaming Multiprocessors	12	30
Scalar Processors	96	240
Scalar Processor Clock (MHz)	1200	1300
Single Precision GFLOPS	346	933
Double Precision GFLOPS	N/A	78
Memory Amount (MB)	320	4096
Memory Interface	320-bit	512-bit
Memory Clock (MHz)	800	800
Memory Bandwidth (GB/s)	64	102
PCIe Version	1.0	2.0 (1.0 used)
PCIe Bandwidth (GB/s)	4	8 (4 used)
CUDA Compute Capability	1.0	1.3

Table 6.2: *Seconds spent in the different parts of the simulation of the WaveEquationSample Modelica model.*

	8800 GTS	C1060 single	C1060 double
Task Execution	0.164	0.592	0.389
Shared Mem Allocation	1.440	1.426	2.287
Integration	0.417	0.400	0.445
Memory Transfers	1.104	1.332	2.278

6.5 Discussion

Some general conclusions can be drawn from the measurements.

- A relative speedup of 4.6 with 3840 sections was obtained using single-precision calculations and comparing the GeForce 8800 GTS to the Intel E6600 CPU.
- Actual computations take a small amount of time while memory transactions take most of the time.
- The simulation times on the CPU approximately double when the model size is doubled. The simulation times for the GPUs instead rise slower. However, the GPUs need many thread blocks with many threads to fully utilize their power.
- The computation per variable is low for the model used. If the model would have had more computations per variable we would most likely have seen a larger performance increase when using a GPU.

Chapter 7

Compilation of Modelica Array Computation into Single Assignment C for Execution on Graphics Processing Units

This chapter is mainly based on Publication 3. In that paper the possibility of compiling Modelica array equations into an intermediate language, SAC [79], was investigated. SAC is a language with C-like syntax but allows Matlab-style programming on n-dimensional arrays. The Single Assignment C to C Compiler (SAC2C) compiler can generate highly efficient code and several auto-parallelizing back-ends have been developed. These back-ends include the generation of POSIX-thread based code for shared memory multi-cores and CUDA-based code for GPUs. A future plan was to enhance the OpenModelica compiler with capabilities to detect and compile data-parallel Modelica array equations and/or algorithmic array operations into SAC WITH-loops. A SAC WITH-loop is the most important construct in the SAC language. In the paper however, only a feasibility study was conducted. As a first step calls to SAC array operations in the code generated from OpenModelica were inserted manually, and as a second step parts of the OpenModelica runtime system were rewritten in SAC code. The paper was about unifying three technologies OpenModelica, SAC2C and CUDA. Measurements of this new integrated runtime system with and without CUDA were performed as well as stand-alone measurements of CUDA code generated with SAC2C.

This chapter is organized as follows. The chapter begins with a description of SAC and its main characteristics. This is followed by a description of the actual implementation work performed. Some of the measurements from the paper are provided after this and the chapter is concluded with a discussion.

7.1 Single Assignment C (SAC)

SAC combines a C-like syntax with Matlab-style programming on n-dimensional arrays. The highly optimizing SAC2C compiler can generate high performance code from SAC due to its functional underpinnings. Most constructs in SAC, however, are inherited from C and the overall design policy is that a C style construct should behave in the same way as it does in C. However, the strong and explicit support of non-scalar data structures is a major difference between SAC and C. In C the programmer has to allocate and deallocate memory as needed and sharing of data structures is explicit via the use of pointers. In SAC there is no notion of pointers. Allocation, reuse and deallocation of memory is handled by the compiler and runtime system and arrays can be passed to and returned from functions in the same way as scalar values. Memory is reused as soon as possible and array updates are performed in place whenever possible. This is ensured by the compiler and runtime system.

SAC comes with a very versatile data-parallel programming construct, the WITH-loop. Here this construct will only be briefly discussed. A modarray WITH-loop takes the general form set out in Listing 7.1.

Listing 7.1: *General form of SAC modarray WITH-loop.*

```
with {
( lower1 <= idx_vec < upper1) : expr1;
...
( lowern <= idx_vec < uppern) : exprn;
} : modarray(array)
```

Here idxvec is an identifier, and *lower_i* and *upper_i* denote expressions for which for any i *lower_i* and *upper_i* should evaluate to vectors of identical length. *expri* denote arbitrary expressions that should evaluate to arrays of the same shape and the same element type. Such a WITH-loop defines an array of the same shape as an array whose elements are either computed by one of the expressions or copied from the corresponding position of the array. As we shall see, the goal is to map Modelica array equations into SAC WITH-loops.

7.2 Implementation

The performed implementation work is discussed here by using an example model and showing the various compilation steps by the use of this model.

The model used is the WaveEquationSample model introduced in Section 6.2 of this thesis. After letting the compiler flatten this model the system of equations in Listing 7.2 is obtained.

Listing 7.2: *Instantiated equation code from the WaveEquationSample Modelica model.*

```
p[0] = exp(-(-L / 2.0) ^ 2.0);
p[n-1] = exp(-(L / 2.0) ^ 2.0);
der(p[0]) = p[0];
...
der(p[n-1]) = p[n-1];
der(dp[0]) = 0;
der(dp[1]) = c^2.0 * ((p[2]+(-2.0*p[1]+p[0])) * dL^-2.0);
...
der(dp[n-2]) = c^2.0 * ((p[n-1]+(-2.0*p[n-2]+p[n-3])) * dL^-2.0);
der(dp[n-1]) = 0;
```

In Listing 7.3 four expressions are defined (where $0 \leq Y \leq n-1$ and $2 \leq X \leq n-3$). The expressions correspond to the various right-hand side expressions present in the equation system in Listing 7.2.

Listing 7.3: *Definition of expressions from the flattened equation code from the WaveEquationSample Modelica model.*

```
Expression 1.
p[Y]

Expression 2.
c^2.0*((p[2] + (-2.0*p[1] + p[0]))*dL^-2.0)

Expression 3.
c^2.0*((p[X+1] + (-2.0*p[X] + p[X-1]))*dL^-2.0)

Expression 4.
c^2.0*((p[n-1] + (-2.0*p[n-2] + p[n-3]))*dL^-2.0)
```

The generated C++ code from OpenModelica will then have the structure as seen in the pseudo code in Listing 7.4.

Listing 7.4: *Generated equation code from the WaveEquationSample Modelica model.*

```
void functionODE(...) {
  // Initial code
  tmp0 = exp((-pow((L / 2.0), 2.0)));
  tmp1 = exp((-pow(((-L) / 2.0), 2.0)));

  stateDers[0 ... (NX/2)-1] = Expression 1;

  stateDers[NX/2] = Expression 2;

  stateDers[(NX/2 + 1) ... (NX - 2)] = Expression 3;
```

```
  stateDers[NX-1] = Expression 4;
}
```

The code in *functionODE* is rewritten into SAC code, which can be seen in Listing 7.5.

Listing 7.5: *SAC with-loop corresponding to the generated equation code from the WaveEquationSample model.*

```
with {
  ([0] <= iv < [NX/2]) : Expression 1;

  ([NX/2] <= iv <= [NX/2]) : Expression 2;

  ([NX/2] < iv < [NX - 1] : Expression 3;

  ([NX-1] <= iv <= [NX-1]) : Expression 4;
} : modarray(stateVars);
```

A second approach that was tried in the paper was to rewrite the actual simulation loop in SAC. In the first approach we make at least one call to a SAC binary at each time step, which in the following measurements section is shown to be very time consuming. A simple Euler loop was written in SAC, which can be seen in Listing 7.6, where *functionODE* is the same as earlier.

Listing 7.6: *A main Euler simulation loop written in SAC.*

```
while (time < stop)
{
  states = states * timestep * derivatives;
  derivatives = functionODE(states, c, 1, dL);
  time = time + timestep;
}
```

7.3 Measurements

All the experiments were run under CentOS Linux with an Intel Xeon 2.27 GHz processor with 24 GB of RAM, 32 kB of L1 cache, 256 kB of L2 cache per core and 8 MB of processor level 3 cache. SAC2C measurements were run with version 16874, C compiler GCC 4.5 and revision 5625 of OpenModelica were used. Figure 7.1 shows time measurements for the modified generated code from the WaveEquationSample Modelica model with functionODE implemented purely in C and in SAC respectively.

Figure 7.2 shows time measurements for the WaveEquationSample Modelica model with the modified solver loop.

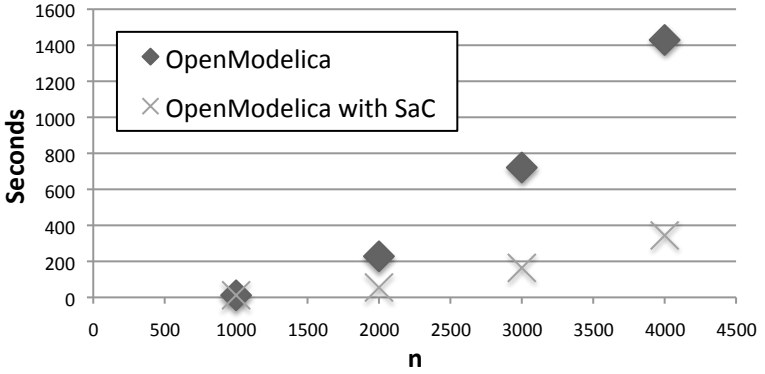


Figure 7.1: *The WaveEquationSample model run for different numbers of sections (n) with functionODE implemented as pure OpenModelica-generated C++ code, as OpenModelica-generated C++ code with functionODE implemented in SAC. Start time 0.0, stop time 1.0, step size 0.002 and without CUDA.*

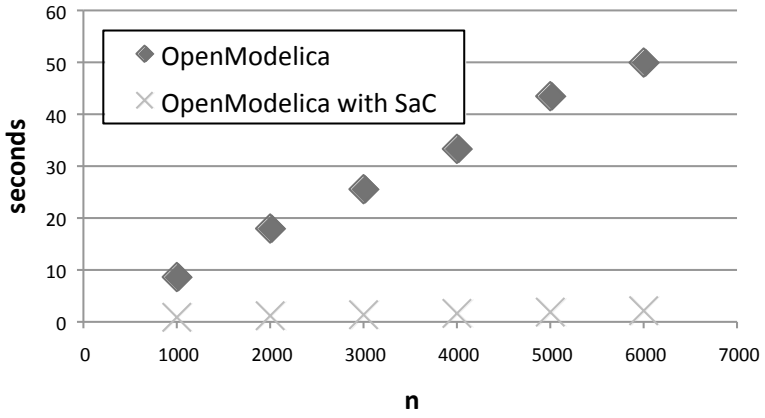


Figure 7.2: *The WaveEquationSample model run for different numbers of sections (n) with functionODE and Euler loop implemented as pure OpenModelica-generated C++ code and as OpenModelica-generated C++ code with functionODE and Euler loop implemented in SAC. Start time 0.0, stop time 10.0, step size 0.002 and without CUDA.*

7.4 Discussion

In this work ways were investigated to make use of the efficient execution of array computations that SAC and SAC2C offer, in the context of Modelica and OpenModelica. It is common to have large arrays of state variables in Modelica code, as in the model used in this chapter. It was demonstrated that it is possible to generate C++ code from OpenModelica that can call compiled SAC binaries for execution of heavy array computations. It was also shown that it is possible to rewrite the main simulation loop of the runtime solver in SAC. In conclusion, the potential was shown for the use of SAC as an intermediate language and runtime language was successfully demonstrated. It at least has potential for code fragments that the OpenModelica compiler can identify as potentially data-parallel. With the new implementation of handling of unexpanded Modelica arrays in the OpenModelica compiler, this work has future promise.

Chapter 8

Extending the Algorithmic Subset of Modelica with Explicit Parallel Programming Constructs for Multi-Core Simulation

This chapter is based on Publication 6. More or less all the previous work described in this thesis has focused on automatic parallelization of equation-based models. That is, it is entirely up to the compiler for finding and analyzing parallelism. In this chapter however, a different approach is investigated and several extensions to the algorithmic part of the Modelica language are introduced. In other words it is up to the end-user modeler to express which parts of a model that should be simulated in parallel, and corresponding OpenCL code is generated. The new constructs include parallel variables, parallel functions, parallel for-loops, etc. It is important to note that these new language constructs are not part of the official language specification. They have instead been added to the OpenModelica compiler for experimental reasons. A similar approach was taken with the NestStepModelica implementation [25].

8.1 Implementation

In this section some of the new language constructs are briefly introduced.

8.1.1 Parallel Variables

Recall that all data to be used on the GPU (the device) must be copied explicitly by the programmer. A special keyword has therefore been added that specifically tells the compiler that a variable should be allocated on the device. An example of this is given in Listing 8.1.

Listing 8.1: *Example of parallel variables in Modelica.*

```
function parvar
    Integer m = 1000;
    Integer A[m,n];
    Integer B[m,n];
    parallel Integer pA[m,m];
    parallel Integer pB[m,m];
end parvar;
```

The first three variables are located in the normal host memory while the last two matrices will be allocated on the device. The copying of data between the host memory and the device memory can then be performed in a normal fashion. The assignments $A := B$, $pA := A$, $B := pB$ and $pA := pB$ would all be valid within the function parvar.

8.1.2 Parallel Functions

With the help of the keyword *parallel*, parallel functions can be defined in Modelica. They correspond to OpenCL functions defined in kernel files or CUDA device functions. Such functions are for distributed independent parallel execution. A *parallel* function must be called from another *parallel* function, from a *kernel* function (see below) or from inside a *parfor* loop. Parallel functions can neither include *parfor*-loops nor declarations of *parallel* variables (since a *parallel* function is already executing on the GPU device).

8.1.3 Kernel Functions

Kernel functions correspond to CUDA global functions and to OpenCL kernel functions. They can be called from serial host code and are entry functions for parallel execution. They cannot, however, be called from the body of a *parfor* loop or from other *kernel* functions. They cannot have *parfor* loops in their bodies nor can they have any explicit parallel variables. They are defined by using the kernel keyword; see Listing 8.2.

Listing 8.2: *Example of kernel function in Modelica.*

```
kernel function arrayElemWiseMultiply
  input Integer A[:];
  input Integer B[:];
  output Integer C[:];
  Integer id;
algorithm
  id := globalThreadId();
  C[id] := multiply(A[id],B[id]);
end arrayElemWiseMultiply;
```

In the function above a parallel function multiply is called. Note the kernel keyword before the function.

8.1.4 Parallel For-Loops

A parallel for-loop is written using the *parfor* keyword and is a loop meant to be executed in a parallel fashion using a device, e.g. a GPU. There are some constraints necessary to make this work. First of all, all variables referenced inside a loop must be *parallel* variables. Secondly, one iteration cannot have a loop-carried dependency to another iteration. An example of a function with a parallel loop is given in Listing 8.3.

Listing 8.3: *Example of parallel for-loops in Modelica.*

```
function parMatrixMult
  input Integer m;
  input Integer A[m,m];
  input Integer B[m,m];
  output Integer C[m,m];
//parallel counterparts of the variables
  parallel Integer pm;
  parallel Integer[m,m] pA;
  parallel Integer[m,m] pB;
  parallel Integer[m,m] pC;
//Integer temp
  parallel Integer ptemp;
algorithm
  pm := m;
  pA := A;
  pB := B;
  parfor i in 1:m loop
    for j in 1:pm loop
          ptemp := 0;
          for h in 1:pm loop
            ptemp := multiply(pA[i,h],pB[h,j])+ptemp;
          end for;
          pC[i,j] := ptemp;
        end for;
  end parfor;
    // copy back C. No other copy back needed
    C := pC;
end parMatrixMult;
```

In the code above the function multiply is a parallel function. Note that the variables referenced inside the loop are all parallel variables. It is enough to specify the *parfor* keyword for the outermost loop, the inner loops will all be considered as parallel loops. The iterations of the loop specified with the *parfor* keyword are equally distributed among available processors. If there are more iterations than threads, some threads might perform more than one iteration.

8.1.5 OpenCL Functionalities

Some additional features have been added for management and execution of parallel operations.

- *oclbuild(String)* Builds an OpenCL source file and returns an OpenCL program object.
- *oclkernel(oclprogram, string)* The first argument is a previously built OpenCL program and the second argument is a kernel. The function creates an OpenCL kernel object.
- *oclsetargs(oclkernel,...)* This function takes a previously created kernel object and a variable number of arguments. It sets each argument to one in the kernel definition.
- *oclexecute(oclkernel)* Executes the specified kernel.

8.1.6 Synchronization and Thread Management

There are also features for managing threads and synchronizations. They are briefly described below.

- *globalThreadId()* This function can only be called from a parallel or kernel function and it returns the global ID of the current thread.
- *localThreadId()* This function can only be called from a parallel or kernel function and it returns the local ID of the current thread (not finalized).
- *globalBarrier()* A global barrier that makes sure that all threads reach this point before any thread is allowed to continue.
- *localBarrier()* Used to synchronize all local threads (not finalized).

8.2 Measurements

Measurements from simulating two models from the implemented benchmark suite are presented in this section. All simulations where run with time step 0.2, with the DASSL solver, start time 0.0 and with a stop time of 0.2

seconds (it makes sense with the same time step and duration since we are working with purely algorithmic models). The time measurement is taken as the difference from when the simulation loop starts and the simulation loop finishes.

- *Matrix Multiplication.* A M×K matrix C is produced from multiplying an M×N matrix A by an N×K matrix B. A considerable speedup has been achieved as a result of parallel simulation of this model on parallel platforms since this model presents a very high level of data-parallelism.
- *LU Decomposition.* The Gaussian Elimination method is used to decompose a matrix to lower and upper triangular forms, which can be used for solving a system of linear equations Ax=B. The size of the problem was successively increased by increasing the values of the parameters M, N, and K of matrices A and B (both matrices had the same size).
- *Stationary Heat Conduction.* This model models the transformation of energy in stationary surfaces. Thermal energy transfers from surfaces with higher temperatures to surfaces with lower temperatures. A parameter N determines the size of the models, which refers to the size of the surface and an equidistant grid.

Used for executing sequential code (generated by the old OpenModelica compiler) was Intel Xeon E5520 CPU, with 16 cores, each with 2.27 GHz clock frequency. For executing parallel code by our new code generator the same CPU was used along with with the NVIDIA Fermi-Tesla M2050 GPU. The simulation execution times are used as results to give us information regarding the following considerations. The measurements were performed to validate that the code generator generates efficient parallel code and to ensure that the Modelica language extensions are successfully targeted toward the OpenCL architecture. The simulation time plots can be seen in Figure 8.1, Figure 8.2 and Figure 8.3 respectively.

8.3 Discussion

In this chapter some novel language constructs for the algorithmic part of the Modelica language have been presented and discussed. Several measurements were provided from a benchmark test suite, MPAR, containing models using these new language constructs. The models contain heavy computations over large matrices. Considerable speedups compared to normal CPU execution were obtained. It is important to once more note that these language constructs are not part of the official Modelica language specification, rather they have been implemented in the OpenModelica compiler for experimental purposes.

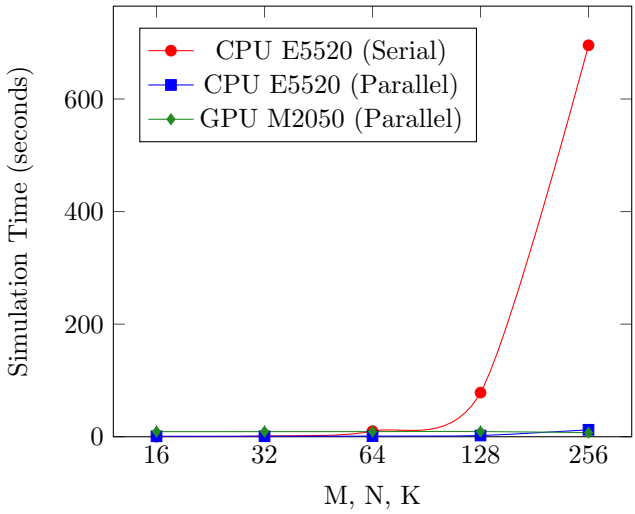


Figure 8.1: *Simulation Time Plot for Matrix Multiplication Model as a Function of Model Parameter M,N, and K.*

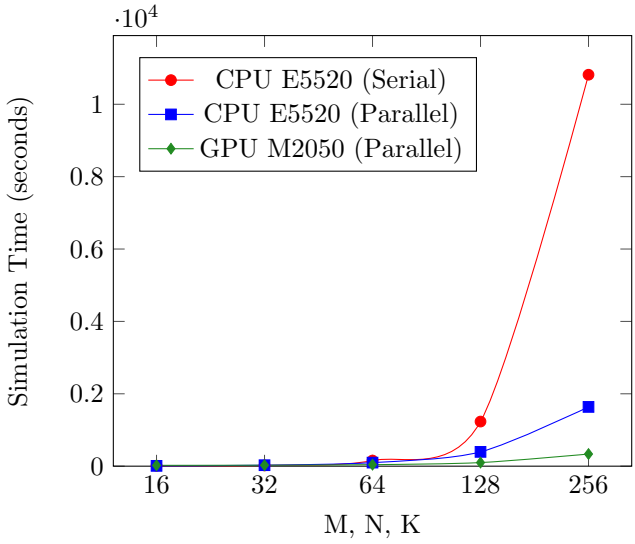


Figure 8.2: *Simulation Time Plot for LU Decomposition Model as a Function of Model Parameters M,N, and K.*

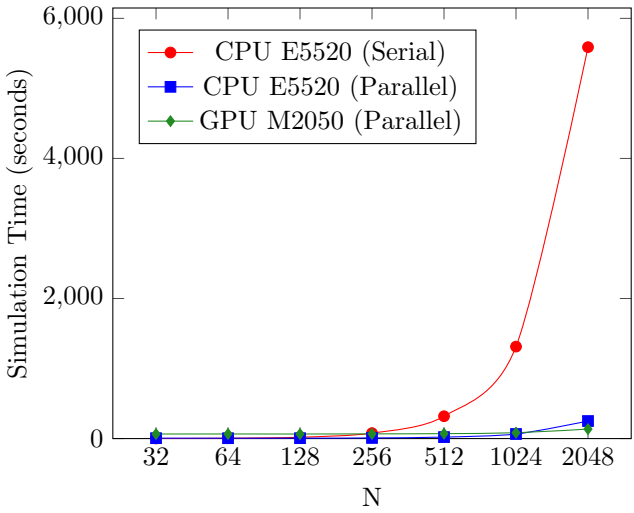


Figure 8.3: *Simulation Time Plot for Heat Conduction Model as a Function of Model Parameter N.*

Chapter 9

Compilation of Unexpanded Modelica Array Equations

Traditionally most Modelica compilers have expanded arrays into scalars, and array equations into scalar equations, with one equation for each array element. This has advantages in providing specific symbolic manipulation for each array element equation. However, this approach also has serious disadvantages when trying to exploit data parallelism from arrays. In this chapter an approach is investigated to keep arrays unexpanded throughout the Modelica compilation process in order to facilitate exploiting data parallelism, for instance for efficient execution on GPUs. The main approach is to avoid expanding array operations and to combine many references of array elements into references of whole arrays or array slices.

9.1 Introduction

The present OpenModelica Compiler (OMC) handles array-related constructs such as arrays of state variables, array equations, and for-equations by expanding them into scalar variables and scalar equations. Work has been done resulting in a preliminary design and prototype to provide functionality to keep arrays of state variables and array equations unexpanded throughout the compilation process. This functionality is activated by a compilation flag. Several changes in the OMC compilation process are needed to support unexpanded arrays. This involves changes in more or less all parts of the compilation process (in the front-end, in the middle parts and in the code generation part). The most difficult task involves changes in the equation-sorting and equation-processing parts of the compilation process. The focus in this chapter is on the equation sections of Modelica; array

constructs in algorithmic sections of Modelica are discussed in Chapter 8. Keeping array equations and arrays of state variables unexpanded opens up the possibility of generating more efficient code for execution on parallel architectures, such as on GPUs. The equation handling parts in the OpenModelica compilation process were briefly described in Chapter 2. By keeping array equations and array variables unexpanded, equation sorting should become faster since the number of equations will be lower. Keeping array equations unexpanded results in smaller matrices and data structures thus leading to less memory consumption and lower compilation time. Keeping array equations unexpanded is also beneficial for normal serial code since the number of statements that are generated in the final executable code is reduced. The compiler could for instance, generate a for-loop instead of many assignments statements. Finally, note that in the traditional approach when array equations are expanded, at the code-generation phase the information about which equations belong together is lost, thus missing opportunities for generation of data-parallel code. Modelica models containing operations over large arrays of state variables often originate from models derived from a discretization of a partial differential equation, one such model was presented in Listing 6.1. See also Chapter 10 regarding PDEs.

This chapter begins with a description of the problem with the current approach given in Section 9.2. The following Section 9.3 and Section 9.4 contain descriptions of some initial algorithms that should be applied to array equations and array for-equations so that they can be handled easier in the equation-sorting phase which is described in Section 9.5. Finally, the chapter is concluded with a discussion in Section 9.6.

9.2 Problems with Expanding Array Equations

The above mentioned loss of array-related information makes it more difficult for the compiler to generate data-parallel code. The following are three problems with the current approach of expanding array equations.

- The current equation matching algorithm in the OMC back-end assumes that array equations and array variables have all been expanded into scalar equations and scalar variables, thus leading to large data structures, substantial memory consumption and long compilation times as a result.
- At the code-generation phase all the original explicit array operations in the model are lost.
- The dimension sizes of arrays and the number of equations related to specific array equations are lost.

9.3 Splitting For-Equations with Multiple Equations in their Bodies

The following can be concluded from the Modelica Language Specification [66] regarding for-equations: since the solution order of the equations in a Modelica model is not specified it does not matter if we split a for-equation into multiple for-equations with one equation each (with the current approach for-equations are expanded into one equation for each array element and merged into the large model equation system; moreover, the order could change). For example, in Listing 9.2 and Listing 9.3 the for-equation in the model is transformed into several for-equations.

9.3.1 Algorithm

An algorithm in pseudo code is shown in Listing 9.1.

Listing 9.1: *Splitting array for-equations with multiple equations in the body.*

```
for each equation in the for-equation body do
  create a separate for-equation containing the equation,
    use the same head
```

9.3.2 Examples

An example of using this splitting approach is given below in Listing 9.2 and 9.3.

Listing 9.2: *Modelica model containing a for-equation with multiple equations in the body.*

```
model TestModel
  parameter Integer n = 4;
  Real u[n](start = 1.0);
  Real x[n](start = 1.0);
equation
    for y in 1:n loop
    der(u[y])=-0.167;
    der(x[y])=80;
  end for;
end TestModel;
```

The model in Listing 9.2 can be transformed into the model in Listing 9.3.

Listing 9.3: *Modelica model containing for-equations with one equation each in their bodies.*

```
model TestModel
  parameter Integer n = 4;
```

```
  Real u[n](start = 1.0);
  Real x[n](start = 1.0);
equation
    for y in 1:n loop
    der(u[y])=-0.167;
  end for;
  for y in 1:n loop
    der(x[y])=80;
  end for;
end TestModel;
```

9.4 Transforming For-Equations and Array Equations into Slice Equations

In order to have all equations in a standard uniform format it is advantageous to transform all for-equations[1] and array equations into slice equations. Such equations have a simple form where all array references have the shape of indexed array slices.

9.4.1 Algorithm

An algorithm in pseudo code for conversion of for-equations and array equations to slice equations is shown in Listing 9.4. The reason for this conversion into a uniform slice equation form is to simplify index reduction, BLT sorting, etc. There are a few forms: equations containing whole arrays (without indexing), array slices, array elements, or for-equations. Single scalar array elements can be eliminated by combining them into slices. For-equations can be transformed into array equations with slices. Whole arrays can be trivially converted to a slices that are the same as the array (e.g. from 1 to end). Overlapping array slices are not handled in the current algorithm but the algorithm can be extended with the following approach: partition the slices into smaller slices so that the overlapping part becomes its own slice.

Listing 9.4: *Splitting array for-equations with multiple equations in the body.*

```
for each equation do
  case equation is for-equation
    for each for-equation iterator do
      case iterator used as an array index
              replace by computing an array slice of
                  the array indexing using the dimension data
                      from the for-equation head

              remove for-equation head, use only body equation
```

[1] Another approach could be to not transform for-equations. For instance algorithmic for-loops in algorithm sections are currently handled as they are.

```
      otherwise
        replace by expanding into an array constructor with
        iterator(s) using the dimension data,
          from the for-equation head

        remove for-equation head, use only body equation
  otherwise
    for each variable reference (if not already slice reference)
      create a slice reference using the dimension information
      for that variable.
```

9.4.2 Examples

Several examples of using this approach are given below. The model in Listing 9.5 can be transformed into the model in Listing 9.6.

Listing 9.5: *Modelica TestModel model containing for-equations.*

```
model TestModel
  parameter Integer n = 4;
  Real u[n](start = 1.0);
  Real x[n](start = 1.0);
equation
    for y in 1:n loop
    der(u[y])=-0.167;
  end for;

  for y in 1:n loop
    der(x[y])=80;
  end for;
end TestModel;
```

Listing 9.6: *Modelica TestModel model containing slice-equations.*

```
model TestModel
  parameter Integer n = 4;
  Real u[n](start = 1.0);
  Real x[n](start = 1.0);
equation
  der(u[1:n])=-0.167;
  der(x[1:n])=80;
end TestModel;
```

The model in Listing 9.7 can be transformed into the model in Listing 9.8.

Listing 9.7: *Modelica WaveEquationSample model containing one array equation one for-equation.*

```
model WaveEquationSample
  import Modelica.SIunits;
  parameter SIunits.Length L = 10 "Length of duct";
```

```
  parameter Integer n = 30 "Number of sections";
  parameter SIunits.Length dl = L/n "Section length";
  parameter SIunits.Velocity c = 1;
  SIunits.Pressure[n] p(each start = 1.0);
  Real[n] dp(start = fill(0,n));
equation
  p[1] = exp(-(-L/2)^2);
  p[n] = exp(-(L/2)^2);
  dp = der(p); // Array equation
  for i in 2:n-1 loop // for-equation header
    der(dp[i]) = c^2*(p[i+1] - 2 * p[i] + p[i-1])/dL^2;
  end for;
end WaveEquationSample;
```

Listing 9.8: *Modelica WaveEquationSample model containing slice-equations.*

```
model WaveEquationSample
  import Modelica.SIunits;
  parameter SIunits.Length L = 10 "Length of duct";
  parameter Integer n = 30 "Number of sections";
  parameter SIunits.Length dl = L/n "Section length";
  parameter SIunits.Velocity c = 1;
  SIunits.Pressure[n] p(each start = 1.0);
  Real[n] dp(start = fill(0,n));
equation
  p[1] = exp(-(-L/2)^2);
  p[n] = exp(-(L/2)^2);
  dp[1:n] = der(p[1:n]);
  der(dp[2:n-1]) = c^2*(p[3:n] - 2 * p[2:n-1] + p[1:n-2])/dL^2;
end WaveEquationSample;
```

9.5 Matching and Sorting of Unexpanded Array Equations

Here an outline of matching and sorting (see Chapter 2) of unexpanded array equations is presented for at least a subset of possible models. The algorithm assumes that the model is balanced (the same number of equations and variables) and that there are no overlapping array reference slices (array slices are non-overlapping if each array element belongs to at most one slice). Handling overlapping array reference slices requires certain modifications to the algorithm: partition the slices into smaller slices so that the overlapping part becomes its own slice.

9.5.1 Algorithm

The algorithm is divided into several steps, shown in Listing 9.9, 9.10, 9.11, 9.12, 9.13 and 9.14.

Listing 9.9: *Step 1: Generate a list with one set for each equation, containing all variable references in the equation.*

```
  for each equation do
    for each variable reference do
      case not array slice reference
        insert the index of variable in the set for the equation.
        add a minus (-) sign if state,
          no minus sign if derivative of state.
      otherwise array slice reference
        insert the index of variable in the set for the equation
          store the array slice reference.
        add a minus (-) sign if state,
          no minus sign if derivative of state.
```

Listing 9.10: *Step 2: Detect overlapping array reference slices.*

```
create an array of lists of booleans, one list of booleans
  for each variable (empty lists to begin with)

for each set in the list from step 1 do
  for each variable reference in the set do
    if the variable reference index is negative then skip
    else
       if variable reference has a slice that overlaps
       with another slice in the list from the array
         (with the same variable name)
         then return
       else insert a variable reference with slice
         information into the list for the correct array entry

if overlapping slices we can not continue with
  the remaining steps
```

Listing 9.11: *Step 3: Check to make sure that the model is balanced.*

```
Check if the number of equations equals the number of variables
  if not the algorithm can not continue with the remaining steps
```

Listing 9.12: *Step 4: Building the incidence matrix (see Chapter 2).*

```
for each set in the list from step 1 do
  create a new row in the matrix
  for each variable reference in the set do
    if the variable reference is not negative insert into row
    else do nothing
```

Listing 9.13: *Step 5: Extended version of matching.*

```
Do the matching as usual but now we also need to check
  that the dimension on the left side equals the
    dimension on the right side.
```

Listing 9.14: *Step 6: Equation sorting.*

```
As before, no major changes needed.
```

9.5.2 Examples

Several examples of the algorithm in the previous section are given in this section.

Example 1

The Modelica model for the first example is shown in Listing 9.15.

Listing 9.15: *Example 1: Modelica Model NonExpandedArray1.*

```
model NonExpandedArray1
  parameter Integer p=500;
  Real x[p];
  Real y[p];
  Real z[p];
  Real q[p];
  Real r[p];
equation
  2.3232*y + 2.3232*z + 2.3232*q + 2.3232*r = der(x);
  der(y) = 2.3232*x + 2.3232*z + 2.3232*q + 2.3232*r;
  2.3232*x + 2.3232*y + 2.3232*q + 2.3232*r = der(z);
  der(r) = 2.3232*x + 2.3232*y + 2.3232*z + 2.3232*q;
  2.3232*x + 2.3232*y + 2.3232*z + 2.3232*r = der(q);
end NonExpandedArray1;
```

eq	
1	x[1:n], -y[1:n], -z[1:n], -q[1:n], -r[1:n]
2	-x[1:n], y[1:n], -z[1:n], -q[1:n], -r[1:n]
3	-x[1:n], -y[1:n], z[1:n], -q[1:n], -r[1:n]
4	-x[1:n], -y[1:n], -z[1:n], -q[1:n], r[1:n]
5	-x[1:n], -y[1:n], -z[1:n], q[1:n], -r[1:n]

The above model has no overlapping slices and the model is balanced. The above model will result in the following matrix.

eq	$der(x[1:n])$	$der(y[1:n])$	$der(z[1:n])$	$der(q[1:n])$	$der(r[1:n])$
1	1	0	0	0	0
2	0	1	0	0	0
3	0	0	1	0	0
4	0	0	0	0	1
5	0	0	0	1	0

=> Sorting =>

eq	$der(x[1:n])$	$der(y[1:n])$	$der(z[1:n])$	$der(q[1:n])$	$der(r[1:n])$
1	1	0	0	0	0
2	0	1	0	0	0
3	0	0	1	0	0
5	0	0	0	1	0
4	0	0	0	0	1

Example 2

The Modelica model for the second example is given in Listing 9.16.

Listing 9.16: *Example 2: Modelica model WaveEquationSample.*

```
model WaveEquationSample
  import Modelica.SIunits;
  parameter SIunits.Length L = 10 "Length of duct";
  parameter Integer n = 30 "Number of sections";
  parameter SIunits.Length dl = L/n "Section length";
  parameter SIunits.Velocity c = 1;
  SIunits.Pressure[n] p(each start = 1.0);
  Real[n] dp(start = fill(0,n));
equation
  p[1] = exp(-(-L/2)^2);
  p[n] = exp(-(L/2)^2);
  dp = der(p);
  for i in 2:n-1 loop
    der(dp[i]) = c^2*(p[i+1] - 2 * p[i] + p[i-1])/dL^2;
  end for;
end WaveEquationSample;
```

=> Transform for-equation =>

Listing 9.17: *Example 2: Modelica model WaveEquationSample after transformation.*

```
model WaveEquationSample
  import Modelica.SIunits;
  parameter SIunits.Length L = 10 "Length of duct";
  parameter Integer n = 30 "Number of sections";
  parameter SIunits.Length dl = L/n "Section length";
  parameter SIunits.Velocity c = 1;
  SIunits.Pressure[n] p(each start = 1.0);
  Real[n] dp(start = fill(0,n));
equation
  p[1] = exp(-(-L/2)^2);
  p[n] = exp(-(L/2)^2);
  dp = der(p);
  der(dp[2:n-1]) = c^2*(p[3:n] - 2 * p[2:n-1] + p[1:n-2])/dL^2;
end WaveEquationSample;
```

eq	
1	-p[1]
2	-p[n]
3	p[1:n], -dp[1:n]
4	dp[2:n-1], -p[3:n], -p[2:n-1], -p[1:n-2]

In this model there are no overlapping slices and the model is balanced. The above model will result in the following incidence matrix. The columns dp[1] and dp[n] result from the check at step 2, which is an extended version.

eq	$der(dp[2:n-1])$	$der(p[1:n])$	$dp[1]$	$dp[n]$
1	0	0	0	0
2	0	0	0	0
3	0	1	0	0
4	1	0	0	0

=> This causes Pantelides algorithm[48] to detect that equations 1 and 2 must be derived to obtain an equation for dp[1] and dp[n], where p[1] and p[n] are dummy states =>

eq	$der(dp[2:n-1])$	$der(p[1:n])$	$dp[1]$	$dp[n]$	$p[1]$	$p[n]$
1	0	0	0	0	1	0
2	0	0	0	0	0	1
3	0	1	0	0	0	0
4	1	0	0	0	0	0
5	0	0	1	0	0	0
6	0	0	0	1	0	0

=> Sorting =>

eq	$der(dp[2:n-1])$	$der(p[1:n])$	$dp[1]$	$dp[n]$	$p[1]$	$p[n]$
4	1	0	0	0	0	0
3	0	1	0	0	0	0
5	0	0	1	0	0	0
6	0	0	0	1	0	0
1	0	0	0	0	1	0
2	0	0	0	0	0	1

Example 3

The Modelica model for the third example is given in Listing 9.18.

Listing 9.18: *Example 3: Modelica model ArraySlice1.*

```
class ArraySlice1
  Real a[4];
equation
```

```
  a[{1,3}] = a[{2,4}];
  a[1]=time;
  a[4]=1;
end ArraySlice1;
```

eq	
1	a[1], a[2]
2	a[3],a[4]
3	a[1]
4	a[4]

The model above has no overlapping slices and the model is balanced. The above model will result in the following matrix.

eq	$a[1]$	$a[2]$	$a[3]$	$a[4]$
1	1	1	0	0
2	0	0	1	1
3	1	0	0	0
4	0	0	0	1

=> Sorting =>

eq	$a[1]$	$a[2]$	$a[3]$	$a[4]$
3	1	0	0	0
1	1	1	0	0
4	0	0	0	1
2	0	0	1	1

9.6 Discussion

In this chapter we discussed how Modelica array equations can be kept unexpanded throughout the compilation process. Previously such equations were expanded into scalar equations and handled as normal scalar equations. However, if array equations can be kept unexpanded there are several benefits. We get a faster compilation process, the equation-sorting phase in particular becomes faster since we have smaller and fewer data structures to process. We can also more easily compile more efficient and faster code. Keeping array equations unexpanded makes it much easier to detect data parallelism when compiling code for parallel architectures such as GPUs.

Part III

Partial Differential Equation Modeling with Modelica via FMI Import of C++ Components

Chapter 10

Partial Differential Equation Modeling with Modelica via FMI Import of C++ Components

10.1 Introduction

Numerical simulation of models that couple PDEs and DAEs in the context of the Modelica modeling and simulation language [66, 37, 38] is discussed. Modelica originated around the idea of solving complex, coupled dynamical systems, which can be described by systems of ODE or DAE. Up to now, there has been only limited support for working with PDEs, despite the fact that the number of Modelica users in academia and in industry has grown significantly lately. One of the first attempts to incorporate PDE support into Modelica is described in [76, 54], and in Chapter 8 of [38], which investigates two different approaches: (1) expressing the PDEs using a combination of new language constructs and a supporting Modelica PDE library using the method-of-lines; (2) exporting the PDE part to an external PDE FEM C++ tool that solves the PDE part of the total problem. Based on this work, an experimental implementation of PDE support was added to the OpenModelica [71] compiler. However, this implementation has not been maintained, even though there have recently been discussions in the OpenModelica community about re-activating these features. Only one simple PDE operator is currently in the Modelica language specification: spatial distribution for 1D PDEs.

In [34] a Modelica library with basic building blocks for solving one-dimensional PDEs with spatial discretizations based on the method of lines or finite vol-

umes is described. Although this approach is attractive due to its simplicity, it is not clear how it could be extended to higher dimensions without significantly increasing the complexity. Another approach is described in [93], which extends the modeling language with primitives for geometry description and boundary/initial conditions, and uses an external pre-processing tool to convert the PDE model to a DAE based on the method of lines. In both of these two works, the PDE system is expanded early on in the compilation process. In this way important information of the PDE structure is lost, information that could have been used for mesh refinement and adjustment of the runtime solver. Another similar option is to use the commercial MapleSim environment [5]. This involves writing the PDEs in a Maple component, exporting this component to DAE form using a discretization scheme and using the resulting component in MapleSim, which supports the Modelica language. An overview of how to use Maple and MapleSim together for PDE modeling can be found in [78]. Apart from the cost for licenses, this method again has the same drawback, arising from the loss of information regarding the original model.

In this work, a way to allow for PDE modeling with Modelica by importing C++ components, written with the HiFlow3 multi-purpose finite element software HiFlow3[10], into Modelica using the FMI [9] import is proposed. FMI is a standard for model exchange and co-simulation between different tools. FMI supports only C but with correct linking it is possible to execute with C++ code. The OpenModelica [71] development environment is used but the same approach can be adapted to other Modelica environments. A similar approach was used in [58, 57], which describes a simulation of the energy supply system of a building using Dymola, ANSYS CFD and the TISC co-simulation environment. Some of the products used in that work are proprietary however, whereas our environment is based on open standards and open source software. Furthermore, the ‘‘model import’’ approach for the coupling between components is used, instead of the ‘‘co-simulation’’ approach applied in those works.

As a proof-of-concept to demonstrate Modelica/HiFlow3 integration, a coupled model that consists of solving the heat equation in a 3D domain and controlling its temperature via an external heat source have been implemented and tested. This source consists of a Modelica Proportional-Integral Derivative (PID) controller, which is taken directly from the Modelica standard library.

The work with Modelica-HiFlow3 coupling was continued with parallel computations on multi-core architectures. A model of a steel beam with a force acting upon it is used to demonstrate implementation. The elasticity deformation of the beam is demonstrated. In this work the actual beam is modeled and solved in parallel using a C++ HiFlow3 [12, 13] component whereas the

physical force acting on the beam is modeled using Modelica. It should be noted that this elasticity deformation model is just for demonstrating our Modelica/HiFlow3 approach. There are better and more powerful methods for elasticity modeling using Modelica in the general case, see for instance [20].

The method described in this work has several advantages:

1. HiFlow3 is well maintained and has strong support and capabilities for PDE modeling and solving,
2. HiFlow3 and OpenModelica are free to download and use,
3. The PDE structure is not lost but is maintained throughout the actual runtime simulation process. This allows for mesh refinement, solver runtime adjustments, etc.,
4. It is possible to mix PDE and DAE systems in the same system setting. This is also possible in [76].

10.2 Simulation Scenario 1: Heat Equation

We consider the evolution of the temperature distribution in a rectangular piece of copper. Figure 10.1 shows the setup for the system. A heat sensor is attached on the right side of the copper bar, and on the bottom there is an adjustable heat source. The system is exposed to environmental influences through time-varying boundary conditions at the top and left side. The goal is to control the temperature in the material by adjusting the heat source, so that a desired temperature is reached at the point of measurement. The regulation of the heat source is performed by a PID controller. It uses the sensor value and a reference temperature to compute the heat source strength.

In our simulation, the two entities in this system are realized by reusing existing software components. The temperature of the copper bar is computed using a HiFlow3 solver, and the ‘‘LimPID’’ controller using a model from the Modelica standard library. The components are coupled by importing the HiFlow3 solver as a Modelica model using FMI and then creating a third Modelica model, which connects these two components as well as some additional components.

10.2.1 Computing the Temperature Distribution

The evolution of the temperature distribution is modeled by the unsteady heat equation. In this subsection, the mathematical problem formulation is given, the numerical treatment of the heat equation is discussed and the discretization we used in the computations is described.

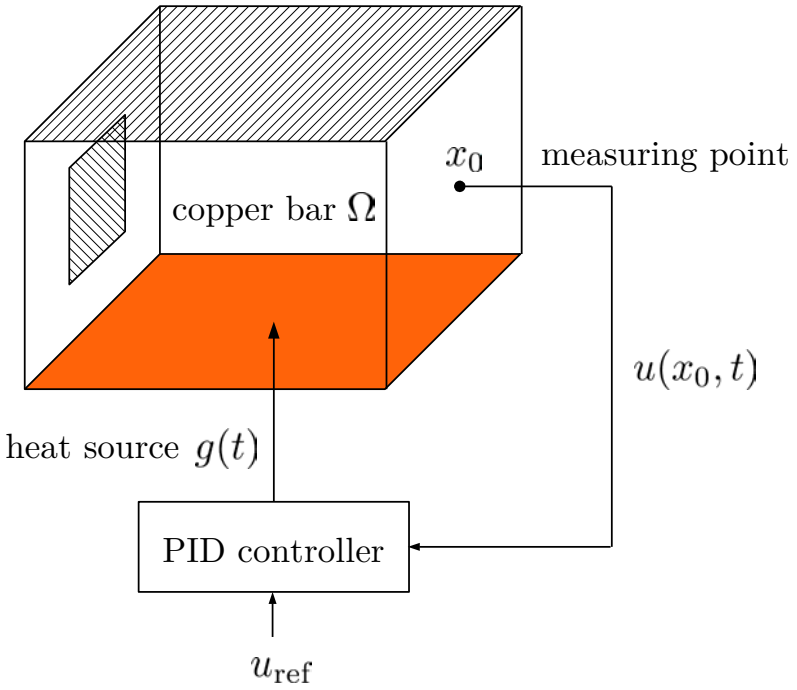


Figure 10.1: *System consisting of a copper bar connected to a temperature regulator based on a PID controller.*

Heat Equation

We consider the copper to occupy a domain

$$\Omega := (0, 0.045) \times (0, 0.03) \times (0, 0.03) \subset \mathbb{R}^3 \,,$$

where the boundary of Ω is denoted by $\partial\Omega$. The heat problem formulation for our simulation scenario is as follows:

Find a function $u : \Omega \times (0, T) \to \mathbb{R}$ as the solution to

$$\partial_t u - \alpha \Delta u = 0 \qquad \text{in } \Omega \times (0, T)\,, \tag{10.1a}$$
$$u(0) = u_0 \qquad \text{in } \Omega\,, \tag{10.1b}$$
$$u = g \qquad \text{on } \Gamma_{\text{src}} \times (0, T)\,, \tag{10.1c}$$
$$u = u_{\text{top}} \qquad \text{on } \Gamma_{\text{top}} \times (0, T)\,, \tag{10.1d}$$
$$u = u_{\text{left}} \qquad \text{on } \Gamma_{\text{left}} \times (0, T)\,, \tag{10.1e}$$
$$\frac{\partial u}{\partial n} = 0 \qquad \text{on } \Gamma_{\text{iso}} \times (0, T)\,. \tag{10.1f}$$

The unsteady heat equation (10.1a) is a parabolic PDE. Its solution, the unknown function u, describes the evolution of the temperature in the copper bar Ω during the time interval $(0, T)$. Here, $\alpha = 1.11 \times 10^{-4} [\frac{m^2}{s}]$ denotes the thermal diffusivity of the copper. u_0 in equation (10.1b) is the initial state at time $t = 0$. The sensor is placed at the point $x_0 := (0.045, 0.015, 0.015)$, where the temperature $u(x_0, t)$ is taken as the measurement value for time $t \in (0, T)$. The heat source is modeled by the Dirichlet boundary condition (10.1c). The controlled temperature $g(t)$ is prescribed for the source part of the boundary

$$\Gamma_{\text{src}} := [0, 0.045] \times [0, 0.03] \times \{0\} \subset \partial\Omega\,.$$

The environmental influence is modeled by the Dirichlet boundary conditions (10.1d) and (10.1e). At the top boundary part

$$\Gamma_{\text{top}} := [0, 0.045] \times [0, 0.03] \times \{0.03\} \subset \partial\Omega$$

and the left boundary part

$$\Gamma_{\text{left}} := \{0\} \times [0.01, 0.02] \times [0.01, 0.02] \subset \partial\Omega$$

the temperature is given by the functions $u_{\text{top}}(t)$ and $u_{\text{left}}(t)$ for $t \in (0, T)$, respectively. The rest of the boundary

$$\Gamma_{\text{iso}} := \partial\Omega \backslash \big(\Gamma_{\text{top}} \cup \Gamma_{\text{left}} \cup \Gamma_{\text{src}}\big)$$

is isolated through the homogeneous Neumann boundary condition (10.1f).

Variational Formulation

A well-established method for numerically solving PDEs like the heat equation is the finite element method. Here the numerical treatment of the problem (10.1) is briefly described. The methods of this section are taken from [31] and [17].

The finite element discretization is based on a variational formulation, which can be derived as follows. We denote by $C^k(X;Y)$ the set of all k times continuously differentiable functions from X to Y, and by $C_0^\infty(X;Y)$ the set of all smooth functions with compact support. In the common case $X = \Omega$, $Y = \mathbb{R}$, we just write $C^k(\Omega)$. Assuming that there is a classical solution

$$u \in C^1(0,T;C^2(\Omega) \cap C(\bar{\Omega}))$$

of problem (10.1), equation (10.1a) is multiplied by a test function $v \in C_0^\infty(\Omega)$ and integrated over Ω:

$$\int_\Omega (\partial_t u)v \, dx - \alpha \int_\Omega (\Delta u)v \, dx = 0 \tag{10.2}$$

Green's first identity [41] is applied to the second term of (10.2), giving

$$\begin{aligned}\int_\Omega (\Delta u)v \, dx &= \int_{\partial\Omega} \frac{\partial u}{\partial n} v \, ds - \int_\Omega \nabla u \cdot \nabla v \, dx \\ &= -\int_\Omega \nabla u \cdot \nabla v \, dx \,,\end{aligned}$$

where n is the outer unit normal on $\partial\Omega$. The boundary integral vanishes since v is zero on $\partial\Omega$. This leads to

$$\int_\Omega (\partial_t u)v \, dx + \alpha \int_\Omega \nabla u \cdot \nabla v \, dx = 0 \,. \tag{10.3}$$

For equation (10.3) to be well-defined, weaker regularity properties of u and v than in the classical context are sufficient. The problem can be formulated in terms of the Lebesgue space $L^2(\Omega)$ of square-integrable functions and the Sobolev space $H^1(\Omega)$ of functions in $L^2(\Omega)$ with square-integrable weak derivatives. We define the solution space

$$V := \{v \in H^1(\Omega) \,:\, v = 0 \text{ on } \Gamma_{\text{left}} \cup \Gamma_{\text{top}} \cup \Gamma_{\text{src}}\}$$

and the bilinear form

$$\begin{aligned} a \,&:\, H^1(\Omega) \times H^1(\Omega) \to \mathbb{R} \,, \\ a(u,v) &:= \alpha \int_\Omega \nabla u \cdot \nabla v \, dx \,. \end{aligned}$$

Note that a is symmetric, continuous and V-elliptic. We denote by $(u,v)_{L^2} = \int_\Omega uv \, dx$ the standard inner product in $L^2(\Omega)$. Now we can state a variational formulation of problem (10.1):

Find $u \in \bar{u} + C^1(0,T;V)$ as the solution of

$$(\partial_t u, v)_{L^2} + a(u,v) = 0 \quad \forall\, v \in V\,, \tag{10.4a}$$

$$u(0) = u_0\,, \tag{10.4b}$$

where $\bar{u} \in C^1(0,T;H^1(\Omega))$ is a given function fulfilling the Dirichlet boundary conditions

$$\begin{aligned} \bar{u} &= g && \text{on } \Gamma_{\text{left}} \times (0,T)\,, \\ \bar{u} &= u_{\text{top}} && \text{on } \Gamma_{\text{top}} \times (0,T)\,, \\ \bar{u} &= u_{\text{src}} && \text{on } \Gamma_{\text{src}} \times (0,T)\,. \end{aligned}$$

This variational formulation admits a unique solution u, which is called a weak solution of the heat equation.

Finite Element Discretization in Space

Let $T_h := \{K_1, ..., K_N\}$ be a triangulation of Ω with N tetrahedral cells $K_i\,(i = 1, ..., N)$. We define the finite element space of piecewise linear functions

$$V_h := \{v \in V \,:\, v\big|_K \text{ is linear } (K \in T_h)\}\,. \tag{10.5}$$

V_h has the finite dimension $n := \dim(V_h)$. We give the problem formulation for a conforming finite element approximation of (10.4):

Find $u_h \in \bar{u}_h + C^1(0,T;V_h)$ as the solution of

$$(\partial_t u_h, v_h)_{L^2} + a(u_h, v_h) = 0 \quad \forall\, v_h \in V_h\,, \tag{10.6a}$$

$$(u_h(0), v_h)_{L^2} - (u_0, v_h)_{L^2} = 0 \quad \forall\, v_h \in V_h\,. \tag{10.6b}$$

Let $\{\varphi_1, ..., \varphi_n\}$ be a basis of V_h. We define the ansatz

$$u_h(x,t) := \sum_{i=1}^{n} w_i(t)\varphi_i(x)$$

and insert it into (10.6a), yielding

$$\sum_{i=1}^{n} \dot{w}_i(\varphi_i, \varphi_j)_{L^2} + \sum_{i=1}^{n} w_i a(\varphi_i, \varphi_j) = 0 \quad (j = 1, ..., n)\,.$$

This can be written as

$$M\dot{w} + Aw = 0\,,$$

where

$$M := \big((\varphi_j, \varphi_i)_{L^2}\big)_{i,j=1,...,n} \in \mathbb{R}^{n\times n}$$

is the mass matrix and

$$A := \big(a(\varphi_j, \varphi_i)\big)_{i,j=1,...,n} \in \mathbb{R}^{n\times n}$$

is the stiffness matrix of the problem, and

$$w : (0,T) \to \mathbb{R}^n$$

is the vector of the time-dependent coefficients. From (10.6b), an initial condition for w is derived as

$$\begin{aligned}&(Mw_0)_i = (u_0, \varphi_i)_{L^2} \quad (i = 1, ..., n)\\ &\iff w_0 = M^{-1} b\,,\end{aligned}$$

where $b_i = (u_0, \varphi_i)_{L^2}$ $(i = 1, ..., n)$. Thus, the finite element discretization in space leads to the initial value problem

$$M\dot{w} + Aw = 0\,, \tag{10.7a}$$

$$w(0) = w_0\,, \tag{10.7b}$$

for the coefficient vector w.

Implicit Euler Discretization in Time

As will be discussed in Section 10.4.4, limitations in the used technology restrict us to using a relatively simple ODE solver for the time discretization. For the heat equation the implicit Euler scheme is suitable, due to its good stability properties [33]. Let $\{0 = t_0 < t_1 < ... < t_m = T\}$ be a partition of the time interval with step sizes $\delta t_k = t_{k+1} - t_k$ $(k = 0, ..., m-1)$. The implicit Euler time stepping method for problem (10.7) is defined as

$$\big[M + \delta t_k A\big] w(t_{k+1}) = M w(t_k) \tag{10.8}$$

for $k = 0, ..., m-1$. This method is convergent and has first-order accuracy in terms of the step size δt_k.

10.2.2 Proportional-Integral-Derivative (PID) Controller

A Proportional-Integral-Derivative controller (PID controller) is a form of loop feedback controller that is widely used to control industrial processes. The controller takes as input a reference value u_{ref}, which represents the desired temperature, and the measured value $u(x_0, t)$. It uses the error $e(t) := u_{\text{ref}} - u(x_0, t)$ to compute the output signal $g(t)$. As the name PID suggests, there is a proportional part that accounts for present errors, an integral part that accounts for the accumulation of past errors, and a derivative part that predicts future errors:

$$g(t) = w_{\text{P}} e(t) + w_{\text{I}} \int_0^t e(\tau)\, d\tau + w_{\text{D}} \frac{d}{dt} e(t)$$

Here, w_{P}, w_{I} and w_{D} are weight parameters. By tuning these parameters the performance of the controller can be adapted to a specific process. A

PID controller is widely regarded as the best controller when information of the underlying process is lacking, but the use of a PID controller does not guarantee optimal control. The tuning of the parameters can be done manually or by using a heuristic method such as Ziegler-Nichols or Cohen-Coon. There are also software tools available. Sometimes one or several of the parameters might have to be set to zero. For instance, because derivative action is sensitive to measurement noise, this part of the controller might have to be omitted in some situations, resulting in a PI controller. PID controllers are linear and can therefore have problems controlling non-linear systems, such as Heating, Ventilation and Air Conditioning (HVAC) systems. [47]

10.3 Simulation Scenario 2: Elasticity Equation

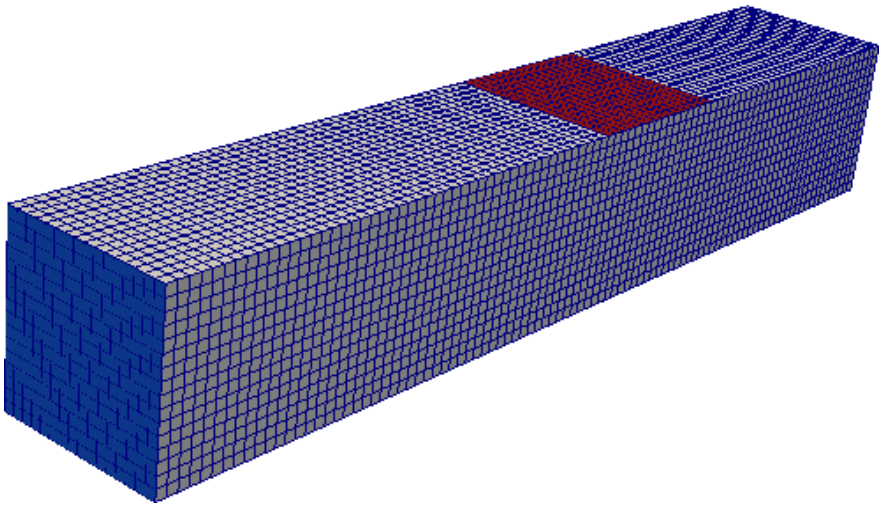


Figure 10.2: *Geometry and computational mesh for the concrete element. The fixed front end is in blue, a force is acting on the red part.*

We now consider the deformation of a rectangular building element under a load. Figure 10.2 shows the setup of the configuration. The element is fixed at both ends, and the load is modeled by an external force acting upon a part of the upper boundary.

10.3.1 Linear Elasticity Model

The element occupies a domain $\Omega \subset \mathbb{R}^3$ with boundary $\Gamma := \partial\Omega$. The behavior of the object subject to a force is described by means of the displacement $\approx$

and the stress tensor σ. Conservation of momentum leads to the equilibrium equation

$$-\nabla \cdot \big([I + \nabla u]\sigma\big) = \rho \mho \quad \text{in } \Omega\,, \tag{10.9}$$

where ρ is the density of the material, $\mho$ is a volumetric force, and I denotes the identity matrix. In our scenario, gravity is the only volumetric force, therefore $\mho = -g_z$.
According to Hooke's law for isotropic materials the stress tensor is related to the deformation tensor ϵ as

$$\sigma = 2\mu\epsilon + \lambda \mathrm{tr}(\epsilon) I\,,$$

with material parameters μ and λ, known as the Lamé elasticity constants. Assuming small deformations, we neglect nonlinear terms in the deformation tensor resulting in the linearized form [26]

$$\epsilon \approx \frac{1}{2}(\nabla u + \nabla u^\top)\,,$$

and the deformation gradient is simplified as

$$I + \nabla u \approx I\,.$$

This leads to the following problem formulation:

Find a function $\cong : \Omega \to \mathbb{R}^3$ as the solution of

$$-\mu \nabla \cdot (\nabla u + \nabla u^\top) - \lambda \nabla(\nabla \cdot u) = \rho \mho \qquad \text{in } \Omega\,, \tag{10.10a}$$

$$u = 0 \qquad \text{on } \Gamma_0\,, \tag{10.10b}$$

$$\Big[\mu(\nabla u + \nabla u^\top) + \lambda(\nabla \cdot u) I\Big] \cdot \propto = p \qquad \text{on } \Gamma_\mathrm{l}\,, \tag{10.10c}$$

$$\Big[\mu(\nabla u + \nabla u^\top) + \lambda(\nabla \cdot u) I\Big] \cdot \propto = 0 \qquad \text{on } \Gamma_\mathrm{f}\,. \tag{10.10d}$$

The homogeneous Dirichlet boundary condition (10.10b) fixes the beam at its ends. The load on the beam acts as a pressure p through the Neumann boundary condition (10.10c), and the homogeneous Neumann condition (10.10d) is imposed on the free part of the boundary.

Variational Formulation

A well-established method for numerically solving PDEs is the finite element method, which is based on a variational formulation of the system (10.10).

Assuming that there is a classical solution $u \in C^2(\Omega, \mathbb{R}^3)$ of problem (10.10), equation (10.10a) is multiplied by a test function $v \in C_0^\infty(\Omega)$ and integrated over Ω:

$$-\mu \int_\Omega \Big[\nabla \cdot (\nabla u + \nabla u^\top)\Big] \cdot v \, dx$$
$$-\lambda \int_\Omega \Big[\nabla(\nabla \cdot u)\Big] \cdot v \, dx = \int_\Omega \rho f \cdot v \, dx$$

The divergence theorem yields (note that $\nabla v = \nabla v^\top$)

$$-\mu \int_\Omega \Big[\nabla \cdot (\nabla u + \nabla u^\top)\Big] \cdot v \, dx \\ = \frac{\mu}{2} \int_\Omega (\nabla u + \nabla u^\top) : (\nabla v + \nabla v^\top) \, dx$$

and

$$-\lambda \int_\Omega \Big[\nabla(\nabla \cdot u)\Big] \cdot v \, dx = \lambda \int_\Omega (\nabla \cdot u)(\nabla \cdot v) \, dx \,,$$

where the boundary integrals are omitted since they vanish as $v = 0$ on $\partial\Omega$. This leads to

$$\int_\Omega \frac{\mu}{2} (\nabla u + \nabla u^\top) : (\nabla v + \nabla v^\top) \\ + \lambda (\nabla \cdot u)(\nabla \cdot v) \, dx = \int_\Omega \rho f \cdot v \, dx \,. \quad (10.11)$$

For equation (10.11) to be well-defined, weaker regularity properties of u and v than in the classical context are sufficient. The problem can be formulated in terms of the Lebesgue space $[L^2(\Omega)]^3$ of square-integrable functions defined on Ω and with image in $\mathbb{R}^3$, and the Sobolev space $[H^1(\Omega)]^3$ of functions in $[L^2(\Omega)]^3$ with square-integrable weak derivatives. We define the weak solution space

$$V := \{u \in [H^1(\Omega)]^3 \,:\, u = 0 \text{ on } \Gamma_0\} \,,$$

the bilinear form

$$a \,:\, [H^1(\Omega)]^3 \times [H^1(\Omega)]^3 \to \mathbb{R} \,, \\ a(u,v) := \int_\Omega \frac{\mu}{2} (\nabla u + \nabla u^\top) : (\nabla v + \nabla v^\top) \\ + \lambda (\nabla \cdot u)(\nabla \cdot v) \, dx \,,$$

and the linear form

$$l \,:\, [H^1(\Omega)]^3 \to \mathbb{R} \,, \\ l(v) := \int_\Omega \rho f \cdot v \, dx + \int_{\Gamma_0} p \cdot v \, ds \,.$$

Note that the bilinear form a is symmetric, continuous and V-elliptic. Now we can state the variational formulation of problem (10.10):

Find $u \in V$ as the solution of

$$a(u,v) = l(v) \quad \forall\, v \in V \,. \quad (10.12)$$

This variational formulation admits a unique solution, which is called the weak solution of the elasticity problem.

Finite Element Discretization

Let $T_h := \{K_1, ..., K_N\}$ be a triangulation of Ω with N tetrahedron cells $K_i\,(i = 1, ..., N)$. We define the finite element space of piecewise linear functions

$$V_h := \{v \in V \,:\, v\big|_K \text{ is linear } (K \in T_h)\}\,.$$

V_h has the finite dimension $n := \dim(V_h)$. We give the problem formulation for a conforming finite element approximation of (10.12):
Find $u_h \in V_h$ as the solution of

$$a(u_h, v_h) = l(v_h) \quad \forall\, v_h \in V_h\,. \tag{10.13}$$

Let $\{\varphi_1, ..., \varphi_n\}$ be a basis of V_h. We define the ansatz function as :

$$u_h(x) := \sum_{i=1}^{n} x_i \varphi_i(x)$$

with coefficients $x_i \in \mathbb{R}$ and insert it into (10.13), yielding

$$\sum_{i=1}^{n} x_i a(\varphi_i, \varphi_j) = b(\varphi_j) \quad (j = 1, ..., n)\,.$$

This can be written as the linear system

$$Ax = b\,, \tag{10.14}$$

where

$$A := \Big(a(\varphi_j, \varphi_i)\Big)_{i,j=1,...,n} \in \mathbb{R}^{n\times n}$$

is the stiffness matrix and

$$b := \Big(l(\varphi_i)\Big)_{i=1,...,n} \in \mathbb{R}^n$$

is the load vector. As the stiffness matrix is symmetric and positive definite [26], we employ the Conjugate Gradient (CG) method [42] for solving (10.14).

10.4 Coupled Implementation

In this section, the technologies used in the present work are introduced. The coupled simulation setup is then described, as well as its two main constituent components in more detail.

10.4.1 The HiFlow3 Finite Element Library

HiFlow3 [10] is a multi-purpose finite element software providing powerful tools for efficient and accurate solution of a wide range of problems modeled by PDEs. Based on object-oriented concepts and the full capabilities of C++ the HiFlow3 project follows a modular and generic approach for building efficient parallel numerical solvers. It provides highly capable modules for dealing with mesh setup, finite element spaces, degrees of freedom, linear algebra routines, numerical solvers and output data for visualization. Parallelism - as the basis for high performance simulations on modern computing systems - is introduced at two levels: coarse- grained parallelism by means of distributed grids and distributed data structures, and fine-grained parallelism by means of platform-optimized linear algebra back-ends.

10.4.2 The Functional Mock-Up Interface

The Functional Mock-Up Interface (FMI) [9, 67] is a tool-independent standard to support both model exchange and co-simulation of dynamic models, which can be developed with any language or tool. A model can be exported as a Functional Mockup Unit (FMU). Such an FMU consists mainly of two parts: (1) an XML file describing the interface and (2) the model functionality in compiled binary or C code form. Other tools or models that also implement the FMI can import an FMU. Initial FMI development was conducted in the European ITEA2 MODELISAR project [11].

10.4.3 Simulation Overview for Simulation Scenario 1

Figure 10.3 provides an overview of the simulation setup. To create the PDE component, an existing HiFlow3 application was reused , which solves the boundary value problem for the heat equation (10.1). In order to import this code into Modelica, it was converted into a Dynamic Shared Object (DSO), which implements the FMI functions and interface descriptions necessary to build an FMU. This FMU was then loaded via FMI into our Modelica model. The details of the PDE component are described in Section 10.4.4.

For the PID controller, we used the LimPID component from the Modelica standard library. This was connected to the PDE component in a new Modelica model, which is described in Section 10.5.

This model was then compiled with the OpenModelica compiler and executed using the OpenModelica runtime system. By choosing the Euler solver, the runtime system provides the time-stepping algorithm according to the implicit Euler scheme, and additionally solves the equations for the PID controller component at each time step. Figure 10.4 illustrates the calls made to the compiled model code on a time axis, which in turn makes calls to the PDE component.

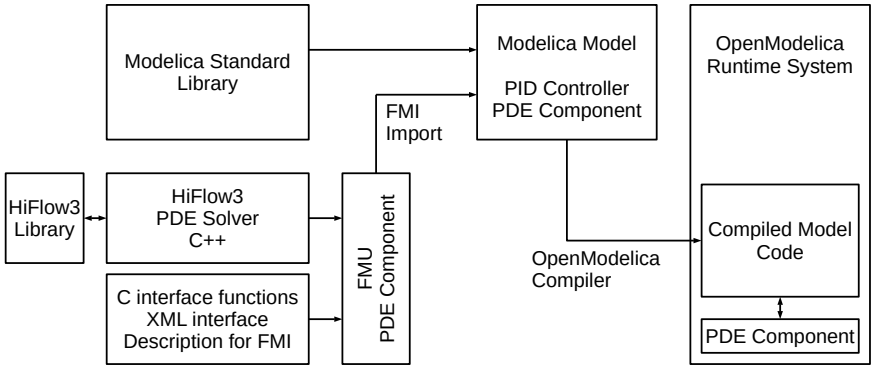


Figure 10.3: *Overview of the creation and coupling of the simulation components.*

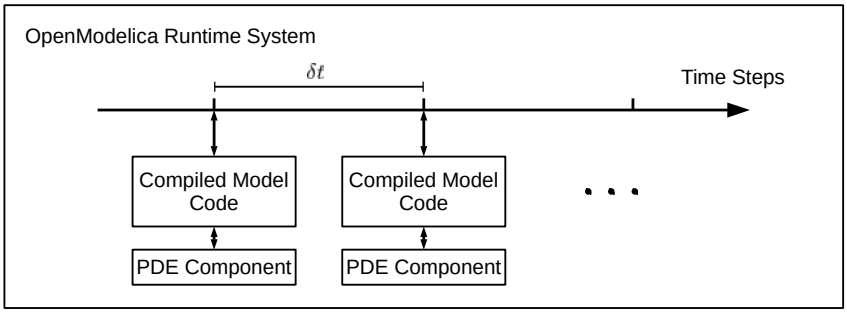


Figure 10.4: *Interaction between the OpenModelica runtime system and the coupled model with the implicit Euler solver.*

10.4.4 HiFlow3-based PDE Component - Simulation Scenario 1

The main sub-part of the PDE component is the `HeatSolver` class, which is a slightly modified version of an existing HiFlow3 application. This class uses data structures and routines from the HiFlow3 library to solve the heat problem (10.1) numerically. It uses a finite element discretization in space and an implicit Euler scheme in time as described in Sections 10.2.1 and 10.2.1.

Furthermore, this class provides functions for specifying the current time, the controlled temperature of the heat source, the top and bottom temperatures, and for retrieving the temperature at the measuring point. The top level routine of the `HeatSolver` class is its `run()` function, see Listing 13.2. This function computes the solution of the heat equation.

The triangulation T_h was then prepared in a preprocessing step and stored it in a mesh file. When the `run()` function is called for the first time, it reads the mesh file and possibly refines the mesh. It also creates the data structures representing the finite element space V_h from (10.5), the linear algebra objects representing the system matrix, the right-hand-side vector and the solution vector. Then, the `run()` function assembles the system matrix $M + \delta t_k A$ and the right-hand-side vector $Mw(t_k)$ according to (10.8). It computes the solution vector $w(t_{k+1})$ for the new time step $t = t_{k+1}$ using the conjugate gradient method [92]. The solution is stored in the VTK format [89] for visualization.

Listing 10.1: *Run function of the HeatSolver class.*

```
HeatSolver_run() {

  // if this is the first call
  if (first_call) {

   // read mesh file and eventually refine it
   build_initial_mesh();

   // initialize the finite element space and
   // the linear algebra data structures
   prepare_system();

   first_call = false;
  }

  // compute the system matrix and
  // the right-hand-side vector
  assemble_system();

  // solve the linear system
  solve_system();

  // visualize the solution
  visualize();

  // keep solution and time in memory
  CopyFrom(prev_solution,old_solution);
}
```

It is important to note that the solution vector and the current time are kept in memory inside the PDE component, since this data is required for computing the solution at the next time step. Although it has been planned for a future version, at present the FMI standard (neither 1.0 nor 2.0) do not directly support arrays, which prevents passing the solution vector back and forth between the PDE component and the Modelica environment as a parameter [67]. Although this use of mutable internal state in the PDE component might be preferable from a performance point of view, it has the drawback of making the function calls referentially opaque: two calls with the same parameters can yield different results, depending on the current internal state. This imposes a strong restriction on the solver used, which must assure that the sequence of time values for which the function `run()` is called is non-decreasing. For this reason, only the method with the simple implicit Euler solver was tested, and verified that the calls were indeed performed in this way. For more complicated solvers, such as DASSL, this requirement is no longer satisfied.

The entry point of the PDE component is the `PDE_component()` function, see Listing 10.2. This function is called within the Modelica simulation loop. When it is called for the first time, it creates a `HeatSolver` object. It sets the input values for the heat source, the temperatures at the top and

bottom boundary, and the current time. Then the `run()` function of the `HeatSolver` is called to compute the solution of the heat equation. Finally, the run counter is incremented and the measurement value is returned.

10.5 Modelica Model 1

Our Modelica model is shown schematically in Figure 10.5. It contains the PDE component, the PID controller and four source components. Two of the sources represent the environmental influences, which are given by sinusoidal functions.

Listing 10.2: *Main simulation routine of the PDE component.*

```
PDE_component(
double in_Controlled_Temp,
double in_Top_Bdy_Temp,
double in_Bottom_Bdy_Temp,
double in_Time)
{
  // create HeatSolver object if this
  // is the first call
  if (run_counter == 0)
    heat_solver = new HeatSolver();

  // set input values
  heat_solver->set_time(in_Time);
  heat_solver->set_g(in_Controlled_Temp);
  heat_solver->set_top_temp(in_Top_Bdy_Temp);
  heat_solver->set_bottom_temp(
    in_Bottom_Bdy_Temp);

  // run the HeatSolver
  heat_solver->run();

  // increment the run counter
  run_counter++;

  // return the measurement value
  return heat_solver->get_u();
}
```

They are connected to the PDE component to give the top and left boundary temperatures u_{top} and u_{left} in Equations (10.1d) and (10.1e), respectively. One source is connected to the PID controller and gives the reference value u_{ref} for the desired temperature. The fourth source is connected to the dummy state variable of the PDE component. The dummy state variable and its derivative are in the model due to the fact that the OpenModelica implementation of FMI 1.0 import does not allow for an empty state variable vector. There is however, nothing in the FMI 1.0 model exchange specification that disallows this. Additionally, the measurement value of the PDE component is connected to the input of the PID controller, and the output

signal of the PID controller is connected to the heat source input of the PDE component.

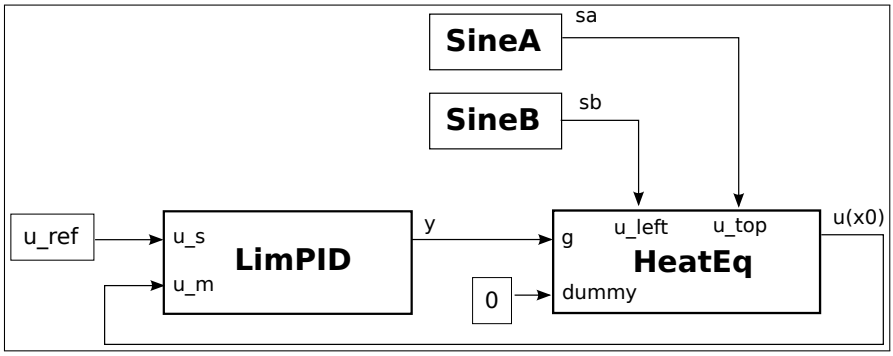


Figure 10.5: *Schematic view of the coupled Modelica model used in the simulation.*

The internal constant parameters of the components are summarized in Table 10.1.

Component Parameter	Value
LimPID	
proportional gain w_{P}	0.05
integral gain w_{I}	0.2
derivative gain w_{D}	0.0
HeatEquationFMU	
thermal diffusivity α	$1.11 \cdot 10^{-4}\,\mathrm{m^2 s^{-1}}$
SineA	
amplitude	0.5 °C
vertical offset	3.5 °C
start time	150.0 s
frequency	$0.001\ \mathrm{s^{-1}}$
SineB	
amplitude	6.0 °C
vertical offset	3.0 °C
start time	350.0 s
frequency	$0.002\ \mathrm{s^{-1}}$

Table 10.1: *Internal parameters of the components in the simulation model.*

10.6 Modelica Model 2

Our (relatively simple) Modelica model is shown in Listing 10.3. It contains the PDE component and the force variable. The variables $hfBlock.stateVar$ and $hfBlock.der\_stateVar$ don't do anything useful but are in the model because the OpenModelica runtime system needs at least one state variable to operate.

Listing 10.3: *Modelica model.*

```
// ------------
// ElasticitySolver
// FMI application with
// HiFlow^3 block for PDE solving.
//
// Authors:
// Chen Song, Martin Wlotzka,
// Kristian Stavaker
//
// Main class
model ElasticitySolver
  // HiFlow^3 component
  ElasticitySolver_me_FMU hfBlock;

  // Source for signals that
  // should be constantly 0
  Modelica.Blocks.Sources.Constant
    zeroSource(k=0.0);

  Real u_center(start=0.0);
  Real force(start=10.0);
equation
 connect(hfBlock.u_center, u_center);
 connect(force,hfBlock.force);
 force = 10.0;
 connect(hfBlock.der_stateVar,
   zeroSource.y);
 connect(hfBlock.stateVar,
   zeroSource.y);
end ElasticitySolver;
```

10.7 Parallelization Concept

The parallelization concept of HiFlow3 is based on a decomposition of the spatial domain into a number of subdomains. For distributed memory systems, the Message Passing Interface (MPI) [63] is used for data transfer. Each MPI process is dedicated to the computation for one of the subdomains. Work is hereby distributed among the processes. After creating the mesh that our finite element discretization is based on, the METIS graph partitioner [39] is used to determine a balanced partitioning of the mesh according to the number of MPI processes. Each process then only stores one part of the

global mesh. Couplings between neighboring parts are taken into account by means of a layer of ghost cells. Figure 10.6 shows an example of a domain decomposition into 8 parts.

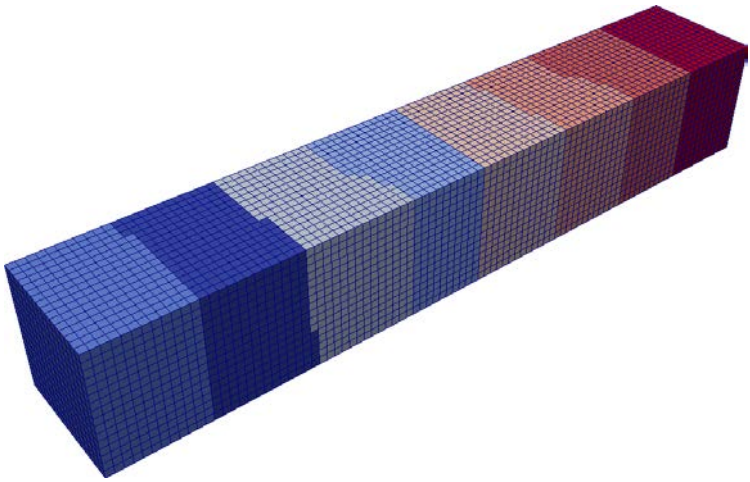


Figure 10.6: *Partitioning of the mesh into 8 subdomains, indicated by different color.*

The matrix and vector data structures in HiFlow3 are distributed data structures that fit the partitioning imposed by the domain decomposition. Each process holds exactly those degrees of freedom of the finite element space that belong to its part of the domain. Couplings between different partitions are achieved by using ghost degrees of freedom. Only these have to be exchanged during parallel matrix-vector-product execution.
Assembly of the system matrix and right-hand-side vector, i.e. the computation of the entries, is performed independently on each process for the corresponding subdomain. The assembling process is hereby designed in two levels: The global assembler iterates concurrently on each subdomain over the cells, while the local assembler computes the contributions for any single cell. Once the matrix and vector are assembled, the Conjugate Gradient linear solver takes advantage of the parallel implementation of the matrix-vector-operations when computing the solution.

10.7.1 Parallel Execution of the Model

The Modelica compiled model code is executed on a number of processes. The Modelica compiled model code is hereby replicated on each process. Whenever the HiFlow3 PDE component is called, it performs distributed parallel computations for solving the elasticity problem, hereby taking advantage of its parallelization concept based on the domain decomposition.

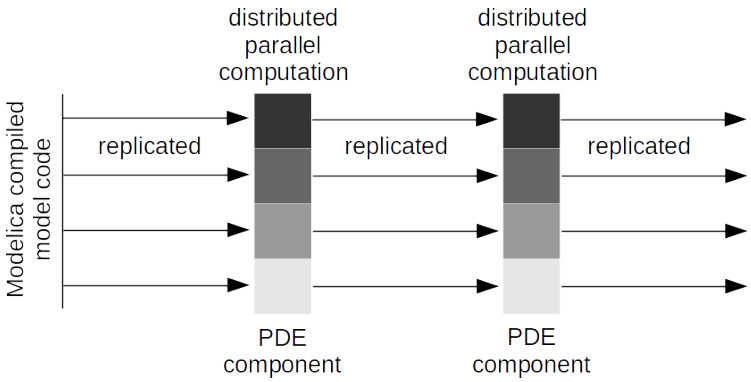


Figure 10.7: *Replicated parallel execution of the Modelica compiled model code and distributed parallel computation in the HiFlow3 PDE component, illustrated for 4 processes.*

10.8 Measurements

A numerical experiment with the following setting was conducted. A simulation time of $T = 1500$ seconds was chosen and a time step of $\delta t = 1$. The initial temperature $u_0 = 0$ was set everywhere in the computational domain Ω, and the desired temperature as $u_{\text{ref}} = 3$ was specified. On the upper part of the boundary Γ_{top} and on the left part of the boundary Γ_{left} the environmental influences was modeled by the functions

$$u_{\text{top}}(t) = \begin{cases} 3.5 & \text{if } t < 150\,, \\ 3.5 + 0.5 \sin\left(\frac{(t-150)\pi}{500}\right) & \text{if } t \geq 150\,, \end{cases}$$

and

$$u_{\text{left}}(t) = \begin{cases} 3 & \text{if } t < 350\,, \\ 3 + 6 \sin\left(\frac{(t-350)\pi}{250}\right) & \text{if } t \geq 350\,, \end{cases}$$

which are shown in Figure 10.8.

For comparison, a simulation run with a constant, uncontrolled heat source $g \equiv 3$ on the lower boundary Γ_{src} was first performed. Figure 10.9 shows that the temperature at the point of measurement deviates from the desired temperature $u_{\text{ref}} = 3$ due to the environmental influences.

The results of our simulation run with a controlled heat source $g = g(t)$ are shown in Figure 10.10. At the beginning, the heat source was fixed at $g = 2.5$ to let the temperature distribution in the copper bar evolve from the initial state to an equilibrium, at which the measured temperature is slightly higher than desired. The PID controller was switched on at $t = 50$ to take control of the heat source. The curves show that the controller first cools the

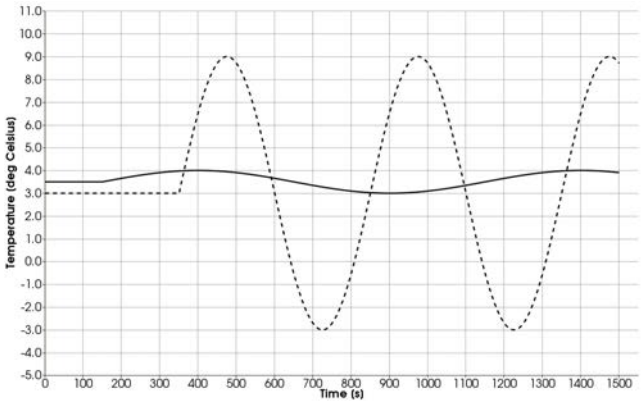


Figure 10.8: *Environmental temperature prescribed on the boundary parts* Γ_{left} *and* Γ_{top}*. Dashed:* $u_{left}(t)$*, solid:* $u_{top}(t)$*.*

bottom to bring the temperature at the point of measurement down to the target value. During the rest of the simulation, the PID controller reacts to the environmental influences and adjusts the heat source dynamically over time, so that the temperature accurately follows the desired state. Figures 10.11-10.13 illustrate the temperature distribution in the copper bar.

A series of test runs were carried out with number of processes $n \in \{1, 2, 4, 8, 16\}$. The runtime T_n for the PDE component when running on n processes was measured. To assess the parallel performance of the solver, the speedup

$$S_n := \frac{T_n}{T_1}$$

and the efficiency

$$E_n := \frac{S_n}{n},$$

were computed, where T_n is the runtime of the solver when executed on n MPI processes, each running on one processor. The results are shown in

n	runtime T_n [sec]	speedup S_n	efficiency E_n
1	8.830	1.0	1.0
2	4.736	1.864	0.932
4	2.948	2.995	0.749
8	1.968	4.487	0.561
16	1.741	5.072	0.317

Table 10.2: *Runtimes for the PDE component with varying number of MPI processes.*

Table 10.2. Figure 10.15 shows a plot of speedup and efficiency.

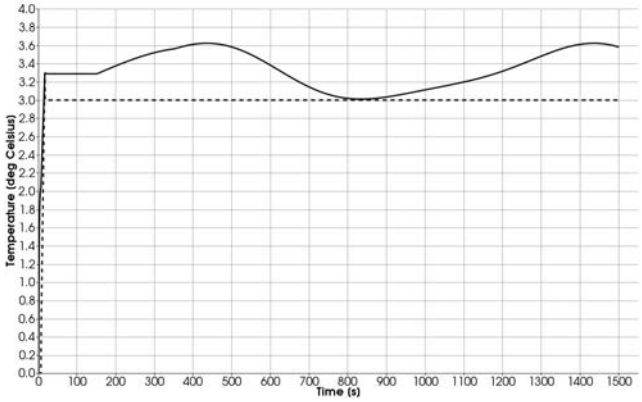


Figure 10.9: *Simulation run with constant heat source $g \equiv 3$. The temperature $u(x_0, t)$ at the point of measurement deviates from the desired value. Dashed: g, solid: $u(x_0, t)$.*

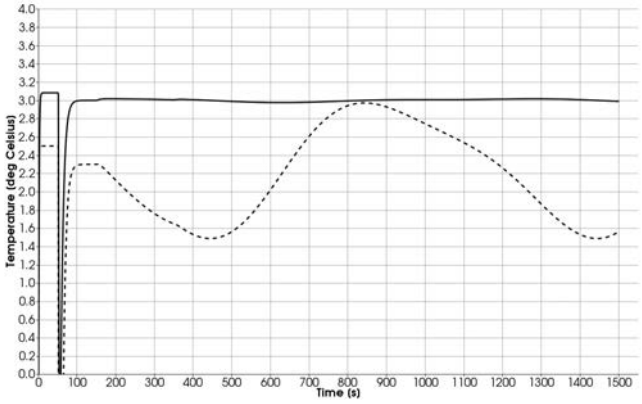


Figure 10.10: *Simulation run with controlled heat source $g = g(t)$. The temperature $u(x_0, t)$ at the point of measurement accurately follows the desired value $u_{ref} = 3$. Dashed: $g(t)$, solid: $u(x_0, t)$.*

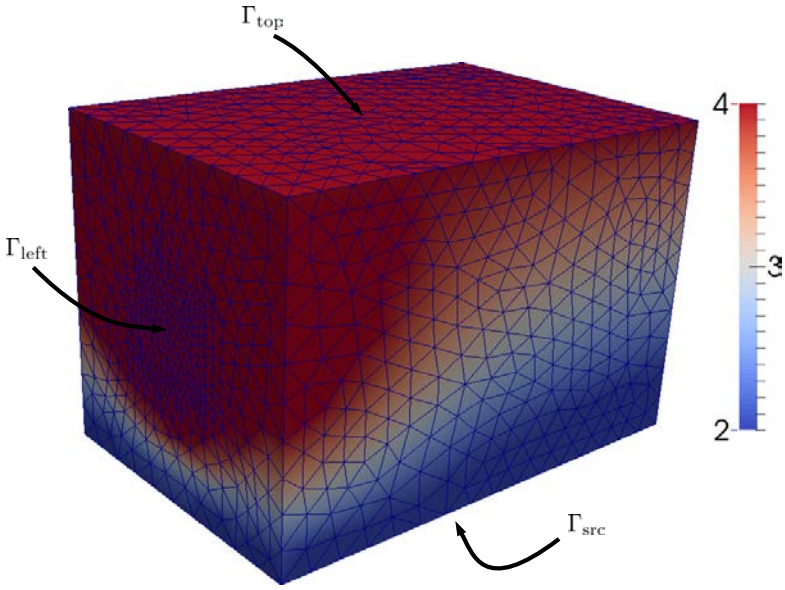


Figure 10.11: *Computational domain of the copper bar with triangulation. The colors indicate the temperature distribution on the surface at time* $t = 440$.

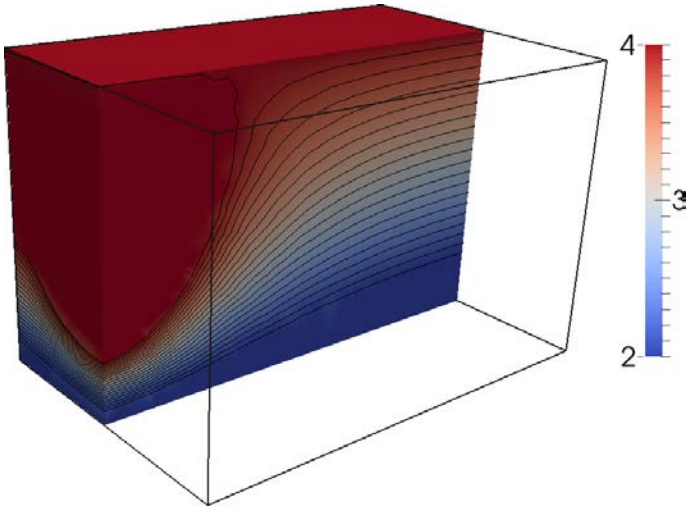


Figure 10.12: *Sectional view with isothermal lines at time* $t = 440$.

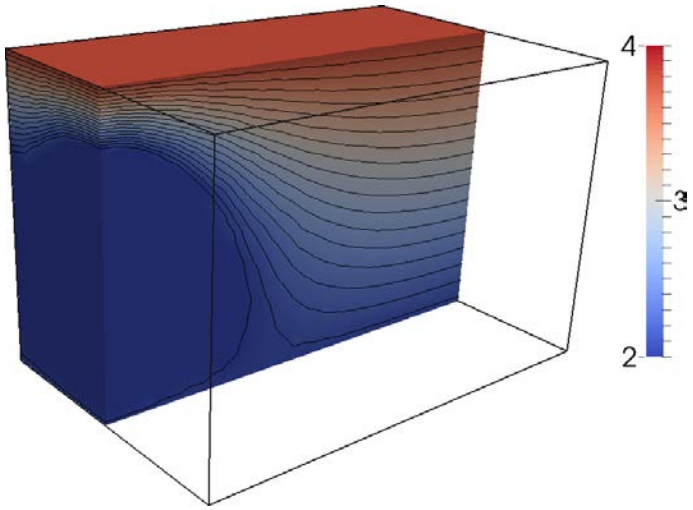


Figure 10.13: *Sectional view with isothermal lines at time $t = 1250$.*

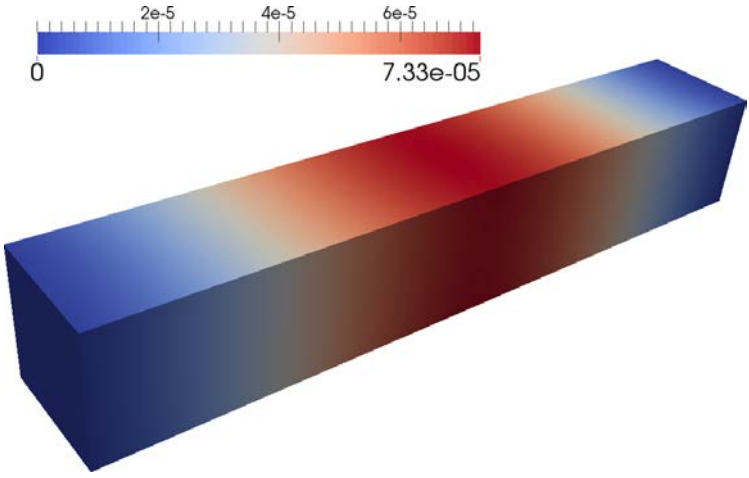


Figure 10.14: *Visualization of the displacement in vertical direction.*

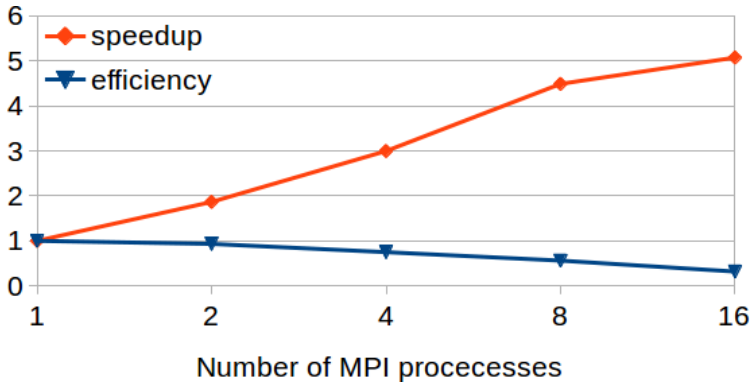


Figure 10.15: *Parallel speedup and efficiency plot for* $n = 1, 2, 4, 8, 16$ *MPI processes.*

The results show parallel performance of the HiFlow3 PDE component within the Modelica context, which is much poorer than the performance obtained for pure HiFlow3 applications. This is due to a technical reason: a custom OpenMPI library had to be installed, where the plugin architecture of the OpenMPI implementation was disabled on the machine while still using shared libraries. This was necessary for compiling the HiFlow3 PDE component into a dynamic shared object that can be loaded by the Modelica compiled model code during runtime. The diminished parallel efficiency is clearly due to the use of such non-optimized MPI installation, since HiFlow3 shows good scalability on other machines with a high-performance MPI installation [88].

Nevertheless, it was possible to leverage the parallel computing capabilities of HiFlow3 in the PDE component to introduce distributed memory parallelization for Modelica simulations. The performance tests show that even if parallel efficiency may not be optimal, our approach allows for solving large scale 3D PDE problems in high resolution on distributed memory machines. This is especially advantageous with respect to the amount of memory available, as the problem data can be split and distributed to several compute nodes, as opposed to shared memory parallelization.

10.9 Discussion

The numerical results for the example presented in the previous section show that our method of integrating the PDE solver from HiFlow3 into a Modelica simulation functions correctly. The realization of this particular scenario serves as an illustration of how one can integrate other, more complicated, PDE models into the complex dynamical simulations for which Modelica is especially suited.

The coding and maintenance effort for importing an existing PDE model implemented in HiFlow3 with the method presented here is minimal; in essence only a set of wrapper functions dealing with input and output of parameters and state variables is all that is required. The fact that HiFlow3 is free and open source software simplifies the process greatly, since it makes it possible to adapt and recompile the code. This is significant, since the FMI model import requires the component to be available either as C source code or as a dynamically shared object, which is loaded at runtime.

Compared to the efforts aiming at extending the Modelica language with support for PDEs, we are working at a different level of abstraction, namely that of software components. The advantage of this is the ability to make use of the large wealth of existing implementations of solvers for various models, in the present case the multi-purpose HiFlow3 library. Extending the Modelica language would also make it considerably more complex, since problems for PDEs generally require descriptions of the geometry and the conditions applied to the different parts of the boundary. Furthermore, using this information to automatically generate a discretization and a solution algorithm would require a sophisticated classification of the problem, since different types of PDEs often require different numerical methods. However, a drawback of working at the software component level is that the mathematical description of the problem is not directly visible, as it would be if it could be expressed directly in the language.

In contrast to the use of ‘‘co-simulation’’, in this work it was decided to import the PDE component into the OpenModelica environment, and to make use of one of its solvers. The main benefit is again simplicity; very few changes were required to the PDE component itself, and it was possible to maximize the reuse of existing software. On the other hand, co-simulation, where each sub-model has its own independent solver, which is executed independently of the others, also has its advantages. In particular, specialized, highly efficient solution algorithms can be applied to each part of the model, and it is possible to execute the various components in parallel. Extending present work to make it usable in a co-simulation setting has been considered. Furthermore, it would be of interest to investigate the parallelization of the simulation both within and between components.

In this work a method of incorporating PDEs in the context of a Modelica model was investigated, by using FMI to import a PDE solver from the finite element library HiFlow3. Numerical results obtained using a simple coupled model controlling the heat equation using a PID controller demonstrate that this method works in practice. The main advantages of this type of coupling include its simplicity and the possibility of reusing existing efficient and already validated software. This approach allows the use of more complex PDE models including high-performance, parallel computations. It has the potential of greatly simplifying the development of large-scale coupled simulations. In that case, however, an extension of the method presented here to support co-simulation might be necessary.

In the results section some runtime measurements of these parallel computations and comparisons to single-core computations were provided. Induced by the needs for compiling the PDE component into a dynamically linked shared object and loading it by the Modelica compiled model code, limitations in the MPI library influenced parallel performance. However, the speedup obtained is considerable in simulation practice, and the use of distributed memory architectures is a clear advantage with respect to memory, especially for large-scale problems. In combination with our previous work reported in [50], this opens opportunities to address even more compute- and memory-intensive applications, such as instationary fluid dynamics problems. The parallelization approach with replication might seem somewhat clumsy. A better method would perhaps be to let the OpenModelica runtime system drive the parallel distribution. This is a subject of future work and it should be noted that the FMI places limits on parallel communication.

In this work a method of incorporating PDEs in the context of a Modelica model was investigated by using FMI to import a PDE solver from the finite element library HiFlow3. Numerical results were obtained from two scenarios: 1.) the distribution of heat in a piece of copper where the heat source was controlled by a PID-controller; and 2.) a simple coupled model that invoke a force and measures the elasticity deformation of a beam demoefficient. The main advantages of this type of coupling include its simplicity and the possibility reusing existing solver technology on multi-core and distributed memory architectures.

Part IV

Using Parallel Skeletons from Modelica

Chapter 11

Using Parallel Skeletons from Modelica

11.1 Introduction

The goal with this work is to make it possible to use skeleton programming in Modelica models. Modelica is mainly a declarative equation-based simulation and modeling language that also includes imperative algorithmic parts i.e. functions and algorithm sections delimited with the algorithm keyword. Skeleton programming is a style of programming that makes use of high-level structures called skeletons that are basically higher order functions that models a complex computational scheme. SkePU [45, 46, 86, 85, 8, 28, 29] is a well-maintained C++-based library supporting skeletons such as Map, MapReduce, MapArray, MapOverlap, Scan, Reduce, and Generate. SkePU allows for execution of the skeletons on various architectures, including multi-core. Parallel skeletons are useful for developing parallel algorithms in a quick and efficient manner and they allow structured composition.

In this work we designed and implemented a Modelica library that call external C++ objects containing SkePU library code. This work also includes some minor compiler extensions here made in the open-source OpenModelica [71] compiler. We provide some measurements of using the Modelica-SkePU library with examples from the SkePU test suite ported into Modelica. We have concluded that this a good way of extending Modelica with parallel skeleton programming capabilities in an efficient and relatively easy way. We also discuss the limitations of our approach and compare our work with related work and provide an outlook for the future. Generation of fast parallel executable code from Modelica, as well as an adaption of the OpenModelica/Modelica runtime system for parallel execution, has been a research topic for several years in our research group.

This chapter begins with a motivation on why skeleton programming is useful followed by a description of the SkePU C++ library (version 1.1). This is followed by a use case section with examples on how the Modelica-SkePU library is to be used in the Modelica environment. Next is a description of the implementation of the Modelica-SkePU library as well as the compiler extensions. After this comes a measurement section followed by discussions.

11.2 Motivation

The PhD thesis [29] addresses issues associated with efficiently programming modern heterogeneous GPU-based systems. The described SkePU library is a skeleton programming library that makes intelligent implementation decisions - at compile time or runtime - while providing high-level abstractions. Algorithmic skeletons were first in introduced in 1989 in [68]. The idea of using skeletons comes from the area of software composition; smaller components can be used to compose larger software applications. When using a skeleton one needs to provide the actual computation logic as a parameter for the skeleton. The skeleton provides the mechanisms for composition into a larger program but the actual computation logic must be provided. Using skeletons has several advantages: re-use, compositionality, maintainability, readability and usability.

11.3 SkePU - Autotunable Multi-Back-end Skeleton Programming Framework for Multi-Core CPU and Multi-GPU Systems

SkePU is an open-source skeleton programming framework for multi-GPU systems and multi-core CPUs. It has been developed as part of a research project at PELAB, Linköping University. Eight different skeletons are supported in version 1.1: Map, Reduce, MapReduce, MapArray, MapOverlap, Overlap, Scan, and Farm. Each skeleton has back-ends (implementations) for sequential C, OpenMP, OpenCL, CUDA, and multi-GPU OpenCL as well as CUDA. For more information on SkePU and skeleton programming see [45, 46, 86, 85, 8, 28, 29]. Some examples of C++ code making use of the SkePU library are given in Section 11.4 and in Section C.3.1.

11.3.1 Containers

SkePU provides its own data structures, containers, that are used together with the skeletons (for instance SkePU Matrix and SkePU Vector).

11.3.2 User Functions

SkePU provides user functions, that are basically C structs containing data and functions, that can be used with the skeletons. The user can define his or her own functions in a macro language. These macros are then automatically expanded into the structs that contains data and functions for different target architectures. This solution with user functions that are expanded to different target architectures at compile time (by the pre-processor) is a good way of writing target-independent code.

11.3.3 Skeletons

The following skeletons are included in the SkePU distribution 1.1.

- **Map**: Takes a user function at instantiation; this function is mapped to each element of a container, resulting in a new container.
- **Reduce**: Takes a binary user function at instantiation; this function is applied repeatedly on all elements of a container until all values have been reduced to a scalar value.
- **MapReduce**: Similar to Reduce but takes an additional user function that is applied to every pair of elements in a container before the reduction with the Reduce function.
- **MapOverlap**: Each element of the result container is a function of several adjacent elements of one input container that reside within a certain constant maximum distance from i in the input container.
- **MapArray**: For two input containers it produces an output container where each element of the resulting container is a function of the corresponding element of one of the input containers and any number of elements from the other input container.
- **Scan**: Takes a binary user function at instantiation, and for a given input container, computes a new container where each element is the result of applying the function on an increasing number of elements from the input container.
- **Generate**: Takes a generate user function that is used for generating the elements of a container.

11.4 Use Cases

In this section a description is provided of how the Modelica-SkePU library is to be used. In other words, from the perspective of the end-user modeler. We provide two Modelica use cases. More Modelica use cases can be found in Section C.3.

11.4.1 Use Case 1: Main1

In Listing 11.1 C++ code is shown that makes use of the SkePU 1.1 library. The corresponding Modelica code, which make use of the Modelica-SkePU library, is shown in Listing 11.2.

In the C++ code in Listing 11.1, needed SkePU libraries are first included. A SkePU macro function is defined, *square_f*, with macro type, name, return type, parameter and function body. In the main function a skeleton-object *square* of *Map* type is declared that takes the *square_f* name as template parameter and a memory allocation of a *square_f* object to the constructor. After this two SkePU containers, *SkePU:Matrix*, are declared. The square skeleton-object is then applied to the two matrices and the result matrix is printed.

The corresponding Modelica code is shown in Listing 11.2. The accompanying C++ function is shown in Listing 11.3. A function, *macro1*, is shown with an external SkePU deceleration. The type of the macro as well as the return type are provided as an annotation. In the main class code a *Map* object and two containers are instantiated. An instantiation call to the macro function is given in the algorithmic section. In the equation section calls are made to an apply function and a printing function.

Listing 11.1: *main1 C++*

```
#include <iostream>
#include "skepu/matrix.h"
#include "skepu/map.h"

UNARY_FUNC(square_f, int, a,
    return a*a;
    )

int main()
{
    skepu::Map<square_f> square(new square_f);
    skepu::Matrix<int> m(5, 5, 3);
    skepu::Matrix<int> r(5, 5);
    square(m,r);

    std::cout << "Result: " << r << "\n";

    return 0;
}
```

Listing 11.2: *main1 Modelica*

```
// Map Example
 function macro1
   external "SkePU" square_f();
```

```
  annotation(MacroType="UNARY_FUNC",Type1="double");
end macro1;

class main1
  import skepu_matrix.*;
  import skepu_modelica.*;

  Map square = Map(funcName="square_f");
  SkePU_Matrix m = SkePU_Matrix(type1="double",dim1=5,dim2=5,
      initValue=3.0);
  SkePU_Matrix r = SkePU_Matrix(type1="double",dim1=5,dim2=5,
      initValue=0.0);
algorithm
  macro1();
equation
  Map_MM(square,m,r);
  displayDataMatrix(r,"double");
end main1;
```

Listing 11.3: *skepu_macro_functions.h*

```
#define FUNC_NAME_UNARY1 square_f

// UNARY AND BINARY FUNCTIONS
/*UNARY_FUNC(square_f, double, a,
       return a*a;
       )*/
int square_f(double a)
{
  return a*a;
}
```

11.4.2 Use Case 2: SPH Fluid Dynamics

For a larger example see Listing 11.4, Smooth Particle Hydrodynamics (SPH). The corresponding C++ user functions can be found in Listing C.27. In this example, 6 user functions of different macro types are used; the interfaces are listed first in 11.4. In the class, the different skeletons are instantiated that make use of the user functions, as well as two vectors that are used with the skeletons.

Listing 11.4: *Smooth Particle Hydrodynamics (SPH), Fluid Dynamics Shocktube simulation*

```
// SPH Fluid Dynamics
function macro13
  external "SkePU" sph_init();
  annotation(MacroType="GENERATE_FUNC",Type1="ParticleSPH",Type2="int
      ");
end macro13;

function macro14
  external "SkePU" sph_assign();
  annotation(MacroType="UNARY_FUNC",Type1="ParticleSPH");
```

```
end macro14;

function macro15
  external "SkePU" sph_updatecell();
  annotation(MacroType="ARRAY_FUNC",Type1="ParticleSPH");
end macro15;

function macro16
  external "SkePU" sph_computedensity();
  annotation(MacroType="ARRAY_FUNC",Type1="ParticleSPH");
end macro16;

function macro17
  external "SkePU" sph_updateforce();
  annotation(MacroType="ARRAY_FUNC",Type1="ParticleSPH");
end macro17;

function macro18
  external "SkePU" sph_updateposition();
  annotation(MacroType="UNARY_FUNC",Type1="ParticleSPH");
end macro18;

class SPH_Fluid_Dynamics
  import skepu_matrix.*;
  import skepu_modelica.*;

  parameter Integer XYLEN = 100;
  parameter Integer NTrials = 3;
  parameter Integer XLEN = XYLEN;
  parameter Integer YLEN = XYLEN;
  parameter Integer ZLEN = 1;
  parameter Integer NPARTICLES = XLEN*YLEN*ZLEN;
  parameter Integer timesteps = 100;
  Generate sph_init =
       Generate(funcName="sph_init");
  Map sph_assign =
       Map(funcName="sph_assign");
  MapArray sph_update_cell =
       MapArray(funcName="sph_updatecell");
  MapArray sph_compute_density =
       MapArray(funcName="sph_computedensity");
  MapArray sph_update_force =
       MapArray(funcName="sph_updateforce");
  Map sph_update_position =
       Map(funcName="sph_updateposition");
  SkePU_Vector fluid1 =
       SkePU_Vector(type1="ParticleSPH",
           dim1=NPARTICLES,initValue=0.0);
  SkePU_Vector fluid2 =
       SkePU_Vector(type1="ParticleSPH",dim1=NPARTICLES,initValue=0.0
           );
algorithm
  macro13();
  macro14();
  macro15();
  macro16();
  macro17();
  macro18();
  for i in 0:NTrials-1 loop
    Generate_V(sph_init,
           NPARTICLES, fluid1,"ParticleSPH");
    Map_VV(sph_assign,
           fluid1,fluid2,"ParticleSPH");
    MapArray_VVV(sph_update_cell,
           fluid2,fluid1,fluid1,"ParticleSPH");

    for i in 0:timesteps-1 loop
      MapArray_VVV(sph_compute_density,
```

```
                fluid1,fluid2,fluid2,"ParticleSPH");
      MapArray_VVV(sph_update_force,
                fluid2,fluid1,fluid1,"ParticleSPH");
      Map_VV(sph_update_position,
                fluid1,fluid1,"ParticleSPH");
    end for;
  end for;
  displayDataVector(fluid1,"ParticleSPH");
  displayDataVector(fluid2,"ParticleSPH");
equation
end SPH_Fluid_Dynamics;
```

11.5 Implementation

C++ code, containing the creation and managing of external C++ objects, is first compiled into a library file. The SkePU-Modelica library (contained in two modelica files) calls upon external C++ functions contained in the compiled library file (there is a C++ interface header file with function decelerations). The user can then include the two SkePU-Modelica library files and use the constructs contained in these files. The simplest option is to then use an OpenModelica script file to launch the simulation and link the compiled library file with the Modelica code. See Figure 11.1 for an overview of the usage process.

11.5.1 Modelica-SkePU Library

Most of the code is in Modelica files and in external C++ files. Minor changes have been made in the OpenModelica compiler; see the next section.

- **skepu_cpp.cpp**: This file contains the actual implementations of the used C++ objects and functions. See Listing C.2.
- **skepu_header.h**: This file contains declarations of all the functions in *skepu_cpp.cpp*. This is an interface needed for the linking process. See Listing C.1
- **skepu_macro_functions.h**: This file contains the SkePU user functions. The user of the Modelica-SkePU library must put his or her SkePU user functions here. See Listing C.27.
- **skepu_modelica.mo**: This file contains the classes and functions that implement SkePU skeletons in Modelica. However, actual SkePU implementations of skeletons and functions implementations are called using external C/C++ and external C/C++ objects. See Listing C.5.
- **skepu_matrix.mo**: This file contains implementations of matrix and vector. See Listing C.4.

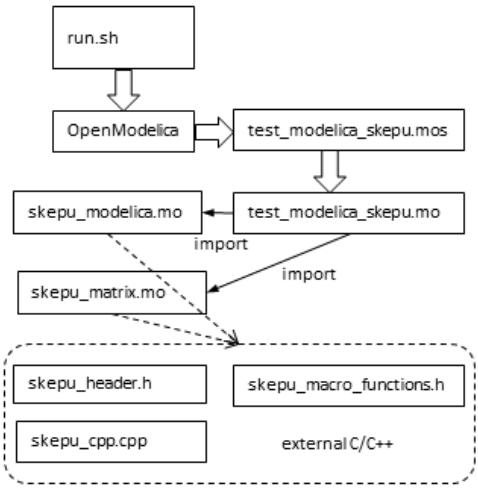


Figure 11.1: *Modelica-SkePU Library and Test Files*

11.5.2 OpenModelica Compiler Extensions

Some minor compiler extensions have been made to handle the external SkePU-macros.

- **SimCode.mo**: In union type *Function* a new data structure *SkePU_MACRO* has been added for handling an external SkePU macro function.
- **SimCodeUtil.mo**: In *elaborateFunctions2* a new case has been added for handling the external SkePU macros.
- **CodegenC.tpl**: In template function *translateModel* a call is made to a new template function *simulationSkePUMacros*. From this template function the new template function *generateSkePUMacro* is called. It is here that the SkePU macros are generated from the *SkePU_MACRO* constructs.

11.5.3 Implementation Status

The current Modelica-SkePU implementation has been verified on six code examples from Chapter 3 of [28]: Map, MapReduce, Scan, Reduce, MapOverlap and MapArray, as well as in eight example programs from the SkePU Version 1.1 test suite: Mandelbrot, LU Factorization, Mean Square Error (MSE), Pearson Product-Moment Correlation Coefficient (PPMCC), Peak Signal to Noise Ratio (PSNR), Taylor Series Calculation, Smooth Particle Hydrodynamics (SPH) and A Runge-Kutta ODE solver. All of the example

code has been manually ported into Modelica. The code can be found in Section C.3. This is the implementation status of the Modelica-SkePU library at the time of printing.

- Support for double and integer type. Weaker support for arbitrary types, such as in Use Case 2 (Listing 11.4).
- There is a limit on the number of skeletons that can be used at the same time. This has to do with the way the external objects are created in C++.
- Not all the API functions in SkePU 1.1 have yet been ported.

11.6 Measurements

In this section certain measurements are provided. The current Modelica-SkePU implementation has been verified the six code examples from Chapter 3 of [28]: Map, MapReduce, Scan, Reduce, MapOverlap, and MapArray, as well as on eight example programs from the SkePU Version 1.1 test suite: Mandelbrot, LU Factorization, Mean Square Error (MSE), Pearson Product-Moment Correlation Coefficient (PPMCC), Peak Signal to Noise Ratio (PSNR), Taylor Series Calculation, Smooth Particle Hydrodynamics (SPH) and A Runge-Kutta ODE solver. All of the example code has been manually ported into Modelica.

11.6.1 System Settings

The following system settings were used for the measurements.

- Linux version 3.10.10-1-ARCH (gcc version 4.8.1 20130725)
- 16 X: Intel Xeon CPU E5520 @ 2.27GHz Cache size 8192 KB
- SkePU Version 1.1
- OpenModelica 1.9.2 Revision 23433

11.6.2 Modelica-SkePU Test Suite Models - Serial

Two series of measurements were performed: serial C++-SkePU and serial Modelica-SkePU. The measurements can be found in Section C.1.1. The built-in C clock was used for both series of measurements. For the Modelica-SkePU code, time measurements were inserted in the generated code that computes the main algorithmic part of the program. The main reason for not using the Linux *time* command for the Modelica-SkePU simulations was that the simulations were run several times. This cannot be changed by modifying the simulation settings since it is hard-coded in the OpenModelica runtime

system that the right-hand side or algorithmic part might be computed several times. Another reason was to avoid the process startup overhead when using the Linux *time* command.

11.6.3 Modelica-SkePU Test Suite Models - Parallel

Two series of measurements were performed (for various sizes of data): serial Modelica-SkePU and parallel Modelica-SkePU with OpenMP. The built in C clock was used. Time measurements were inserted in the generated code that computes the main algorithmic part of the program. Figures 11.2, 11.3, 11.4, 11.5, 11.6, 11.7, 11.8, and 11.9 show the execution time for the main algorithmic part of the various models running Modelica-SkePU serial and parallel with OpenMP. Figures 11.10, 11.11, 11.12, 11.13, 11.14, 11.15, 11.16, and 11.17 show the relative speedup for the main algorithmic part of the various models running Modelica-SkePU serial and parallel with OpenMP. All the time measurements can also be found in Section C.1.2. According to the Frequently Asked Questions (FAQ) section of the official SkePU webpage [8], SkePU will run as many OpenMP threads as possible by default. See Section 11.6.1 for system settings.

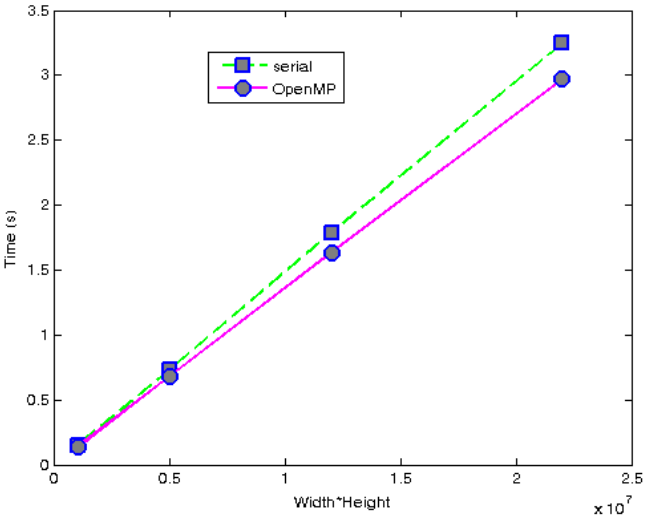


Figure 11.2: *Computation time for various data sizes, serial Modelica-SkePU and Modelica-SkePU with OpenMP, Mandelbrot Fractals*

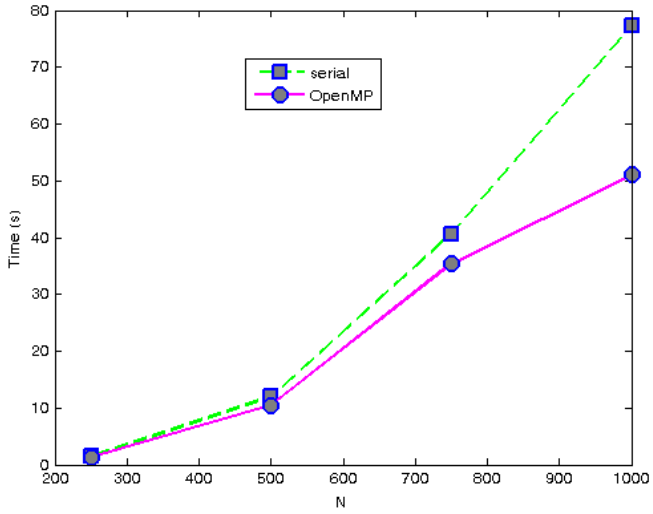


Figure 11.3: *Computation time for various data sizes, serial Modelica-SkePU and Modelica-SkePU with OpenMP, LU Decomposition*

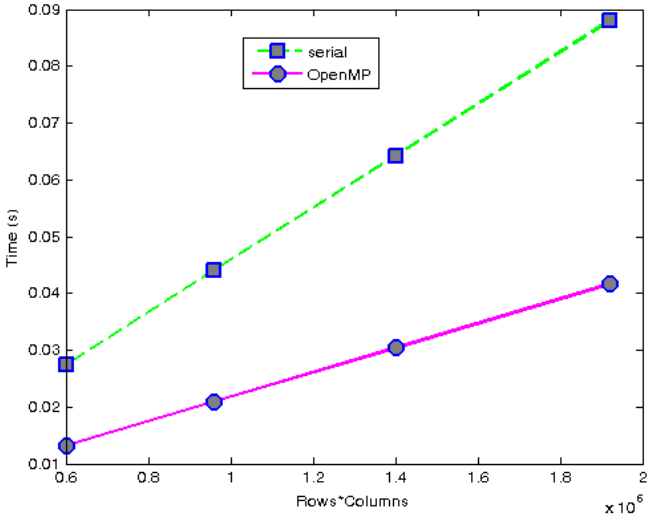


Figure 11.4: *Computation time for various data sizes, serial Modelica-SkePU and Modelica-SkePU with OpenMP, Mean Squared Error (MSE)*

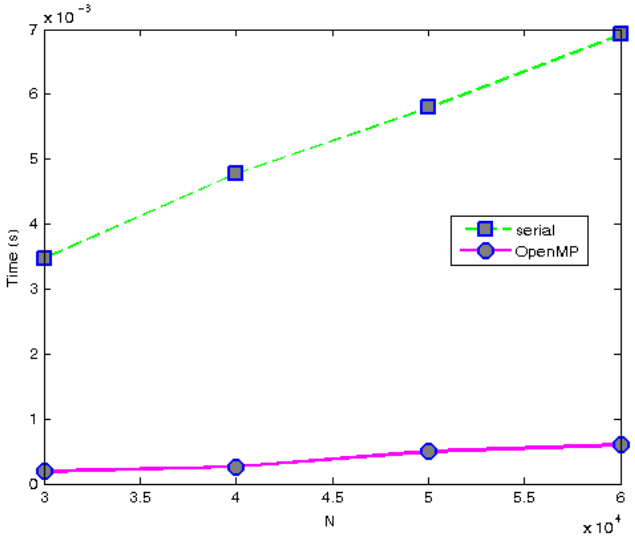


Figure 11.5: *Computation time for various data sizes, serial Modelica-SkePU and Modelica-SkePU with OpenMP, Pearson Product-Moment Correlation Coefficient (PPMCC)*

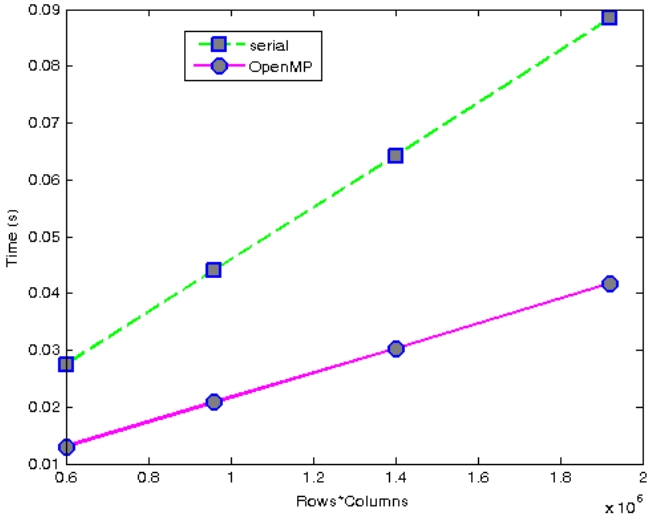


Figure 11.6: *Computation time for various data sizes, serial Modelica-SkePU and Modelica-SkePU with OpenMP, Peak Signal to Noise Ratio (PSNR)*

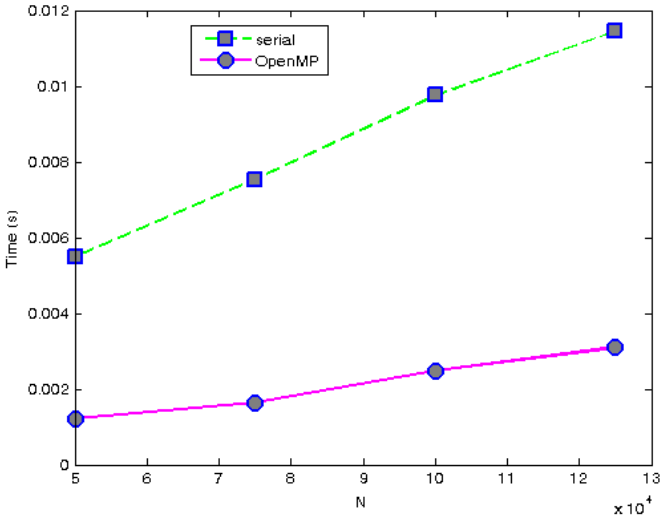


Figure 11.7: *Computation time for various data sizes, serial Modelica-SkePU and Modelica-SkePU with OpenMP, Taylor Series Calculation*

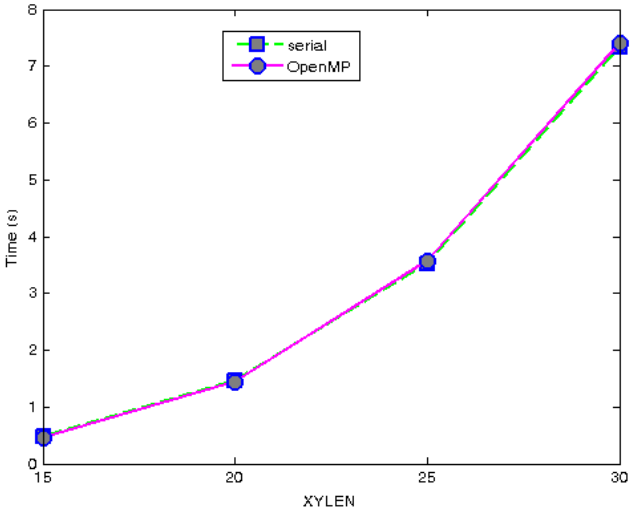


Figure 11.8: *Computation time for various data sizes, serial Modelica-SkePU and Modelica-SkePU with OpenMP, Smooth Particle Hydrodynamics (SPH)*

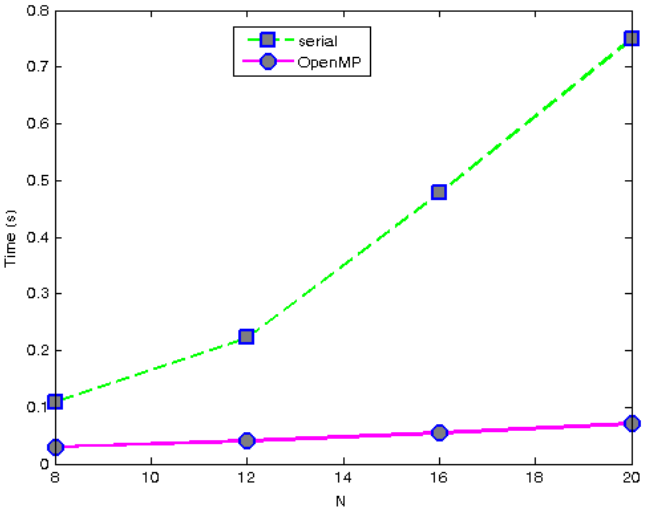


Figure 11.9: *Computation time for various data sizes, serial Modelica-SkePU and Modelica-SkePU with OpenMP, Runge-Kutta ODE Solver*

11.7 Discussion

From the measurements section and Section C.1.1 it can be concluded that the execution times for SkePU with C++ are comparable with using the SkePU-Modelica library. However, some overhead for the Modelica-SkePU code can be noted in Section C.1.1. This could potentially be due to the fact that the generated code from the OpenModelica compiler is different in structure than the manually hand-coded C++-SkePU code. SkePU-Modelica is useful as an addition to the Modelica environment, providing new and strong capabilities for (parallel) algorithm development. In other words in cases when modeling and simulation are needed, SkePU-Modelica offers new and powerful constructs. However, in many cases of computation, when implementation is performed in C or C++, SkePU-C++ is enough. Please also note that only the execution time of computing the main algorithmic part was measured for the SkePU-Modelica code since C++ and Modelica follow very different modes of execution.

From Figures 11.2, 11.3, 11.4, 11.5, 11.6, 11.7, 11.8, and 11.9 and Section it can be concluded that speedup can be gained for at least some of the example codes when comparing Modelica-SkePU serial execution with Modelica-SkePU OpenMP execution. This result was expected since the same skeletons and containers are used. For some of the examples, such as Mandelbrot Fractals and Smooth Particle Hydrodynamics (SPH), there was not any significant improvement in execution time however. All the

example models can be seen in Section C.3. According to the Frequently Asked Questions (FAQ) section of the official SkePU webpage [8], SkePU will run as many OpenMP threads as possible by default. See Section 11.6.1 for system settings. Please also note that only the execution time of computing the main algorithmic part was measured for the SkePU-Modelica code.

In this chapter we have investigated using SkePU skeleton programming with the equation-based modeling and simulation language Modelica using a method with external C++ objects. Skeleton programming is advantageous as parallelism and synchronization come almost for free for the skeleton based expression. Furthermore we can leverage the target architectural features. We described implementation and provided measurements of examples from the SkePU 1.1 test suite ported into Modelica. To the best of our knowledge this is the first attempt at merging skeleton programming with Modelica. We believe that this holds promise for the future. More work is needed with the Modelica-SkePU library for full coverage of the SkePU 1.1 library and to remedy the short-comings described in Section 11.5.3.

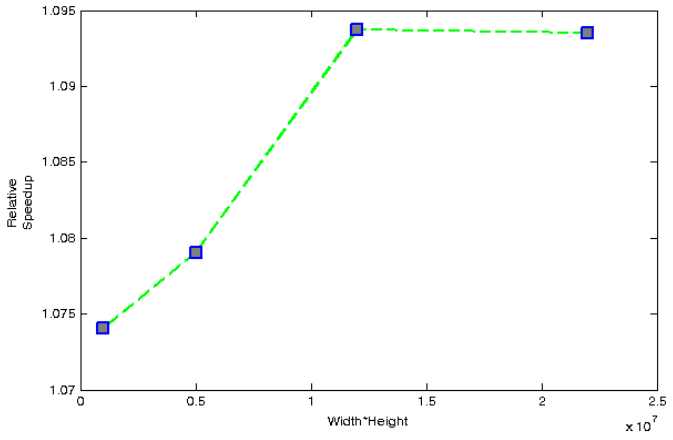


Figure 11.10: *Relative speedup, Mandelbrot Fractals*

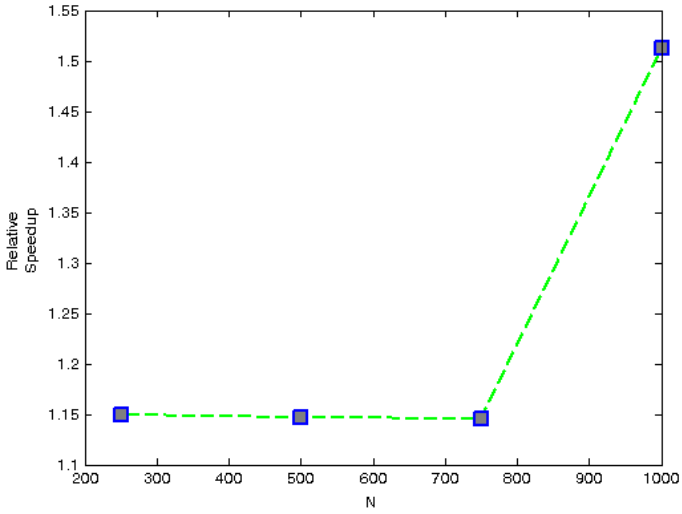


Figure 11.11: *Relative speedup, LU Decomposition*

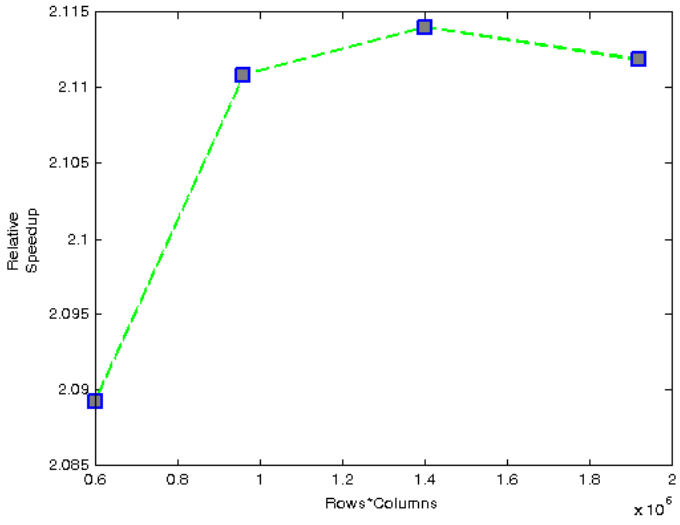


Figure 11.12: *Relative speedup, Mean Squared Error (MSE)*

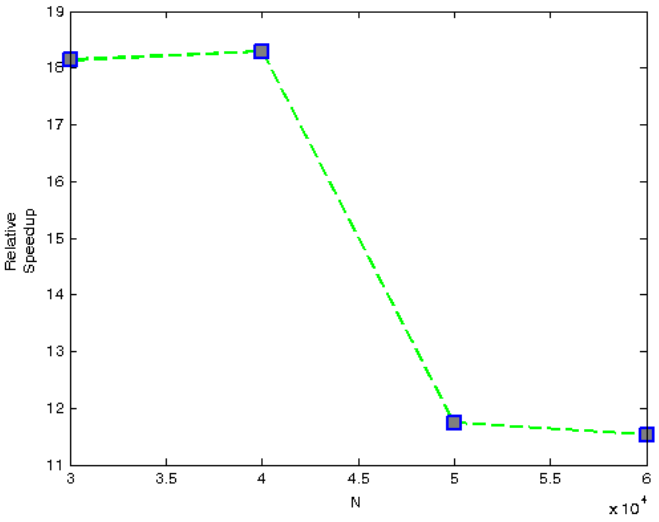


Figure 11.13: *Relative speedup, Pearson Product-Moment Correlation Coefficient (PPMCC)*

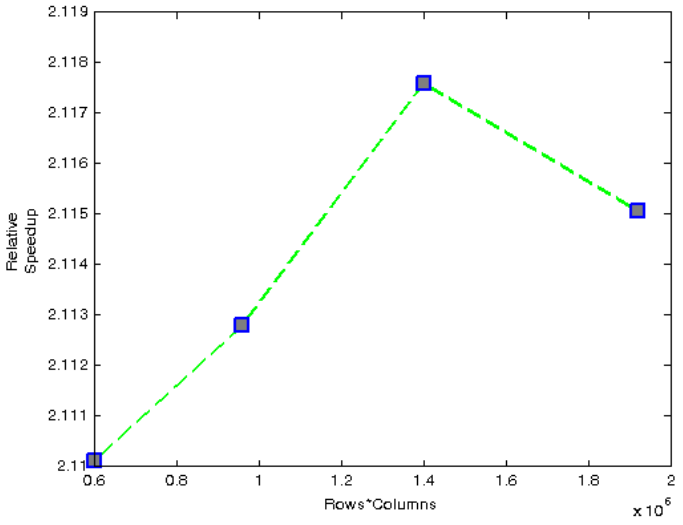


Figure 11.14: *Relative speedup, Peak Signal to Noise Ratio (PSNR)*

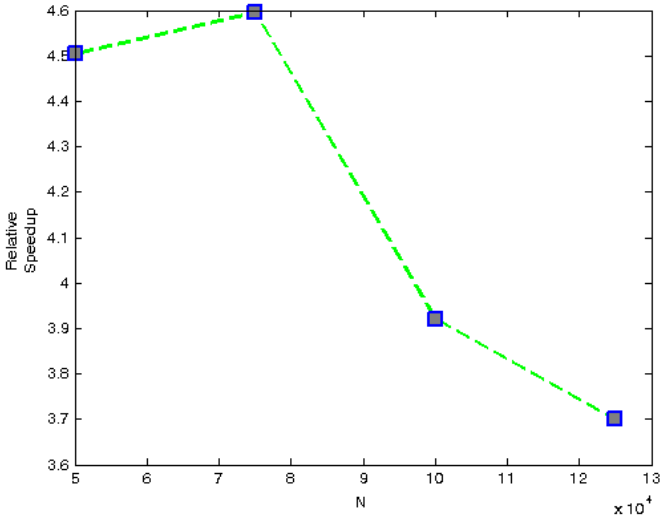


Figure 11.15: *Relative speedup, Taylor Series Calculation*

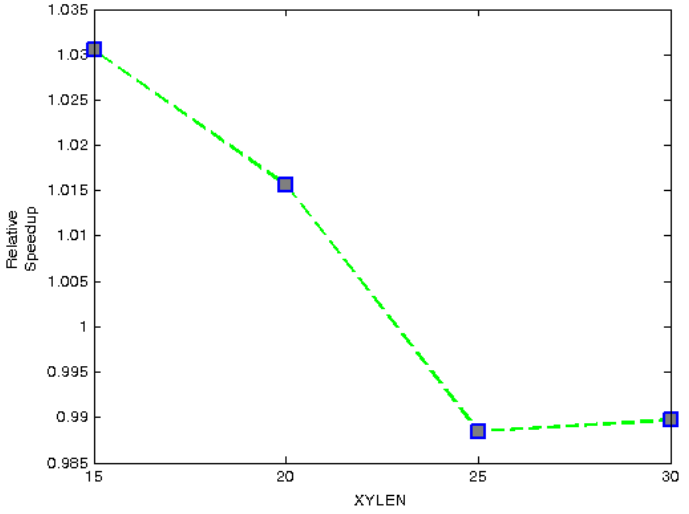


Figure 11.16: *Relative speedup, Smooth Particle Hydrodynamics (SPH)*

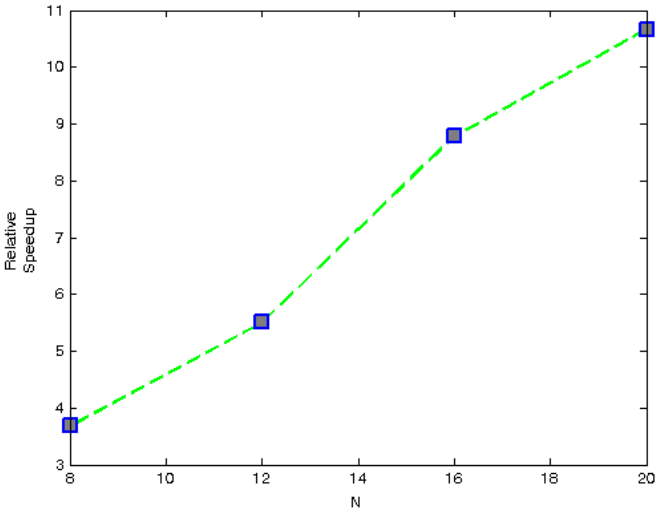


Figure 11.17: *Relative speedup, Runge-Kutta ODE Solver*

Part V

Epilogue

Chapter 12

Discussion

In this chapter the work described in the previous parts (I, II, III and IV) of this thesis are summarized and discussed. Let us first consider the research questions from Chapter 1:

- Is it possible to simulate Modelica models with GPU architectures? Will such simulations run at sufficient speed compared to simulation on other architectures, for instance single- and multi-core CPUs? Are GPUs beneficial for performance? What challenges are there in terms of hardware limitations, memory limitations, etc.?
- What is the current state of PDE modeling in the context of Modelica? What previous research has been done in this area and what are the strengths and weaknesses of this previous research? What are the strengths and weaknesses of the approach of connecting an external (finite element) solver to the Modelica environment via the functional mockup interface?
- What is the current state of skeleton programming in the context of Modelica? What previous research has been done in this area and what are the strengths and weaknesses of this previous research? What are the strengths and weaknesses of the approach of implementing a Modelica-SkePU library that calls external C++ (with some additional minor changes in the OpenModelica compilation)?

GPU architectures in general are discussed in Section 12.1. Section 12.2 provides an attempt to answer the research question from Part 2, Chapter 1 (whether GPU architectures are suitable for parallel simulation of equation-based models). In Section 12.3 the current state of PDE modeling in Modelica and the implementation described in Part 3 of this thesis are discussed. In Section 12.4 the current state of Skeleton programming with Modelica and the implementation described in Part 4 of this thesis are discussed.

12.1 What Kind of Problems are Graphics Processing Units Suitable for?

It is important to note that GPUs have traditionally been used for handling computer graphics computations. They are suitable for algorithms where processing of large blocks of data is performed in a data-parallel fashion. Initially GPUs were used for texture-mapping and polygon rendering, geometric calculations (such as the rotation and translation of vertices), oversampling and interpolation techniques for reducing aliasing. Other examples of applications that can be accelerated in video decoding include: motion compensation, inverse discrete cosine transform, intra-frame prediction, bit stream processing, inverse quantization (IQ), etc.. Applications that are ideally suitable for GPUs have large data sets, high parallelism, and minimal dependency between data elements. [6]

The advent of GPGPU has expanded the usage of GPUs to not just include computer graphics calculations. This move into other fields of computations has come from the observation that most computer graphics calculations include computations of matrices and vectors. GPGPUs have been made possible by the addition of stream processing on non-graphics data features, which in turn have been made possible by the addition of programmable stages and higher precision arithmetic to the rendering pipelines. Although GPGPU and architectures/software platforms such as CUDA and OpenCL have expanded the set of problems that GPUs can solve it is important to note that GPUs are still mainly suitable for algorithms where processing of large blocks of data is performed in parallel, i.e. data-parallelism.

12.2 Are Graphics Processing Units Suitable for Simulating Equation-Based Modelica Models?

The problem of simulating equation-based Modelica models is essentially one of solving a system of differential equations, either in ODE or DAE form, given initial values of the state variables, values of constants and parameters, start time, stop time, time step, etc. (there is also the algorithmic part of Modelica, see below for further information). The front-end and middle part of a Modelica compiler removes all high-level structure of the model and we essentially end up with an equation system that is to be solved at runtime. When solving such an equation system on a computer a numerical method for solution of ODEs is applied (solving it analytically in the general case with a computer is not possible in practice). There are two categories of such numerical methods that have been described in this thesis: time-stepping based methods (e.g., Euler, Runge-Kutta, DASSL, etc.) and quantization of

state methods (QSS).

With the classical time-stepping approach there is essentially a central loop that usually is run on one computational node[1]. When using a system containing GPUs this central loop is most suitable to run on the host CPU. At each time step data must be gathered and distributed from the host memory to the device memory to perform the time step computations, which is time consuming. Another problem is simply that we need to distribute the equation system over the GPU device architecture for computation at each time step. However this is not an easy task since some parts of the equation system might potentially depend on other parts of the equation system, in fact in general this must be assumed. We must therefore communicate between different parts of the distributed equation system and it is often not clear as to how this can be accomplished in a cost-efficient manner. In the approach of Chapter 6 this was done (when data between different streaming multiprocessors needed to be sent) by synchronizing with the global memory, which was very slow. The problem of solving a system of differential equations is not data-parallel in nature in the general case. Models that exhibit a large degree of data-parallelism are suitable though, which is discussed below.

Yet another problem is that the equation system may contain algebraic loops (simultaneous equation systems), in other words an additional solver step must be applied to each such simultaneous equation system in each time iteration. This could potentially be done on one streaming multiprocessor for each simultaneous equation system. Even with a distributed solver approach, such as the approach described in [55], we have the same problem of communication of data between different computations; the same notion still holds, we are trying to solve a problem that is linear in nature on a highly data-parallel architecture. The QSS-based method is somewhat more suitable for parallel computations and its suitability for GPUs is discussed below.

It is also important to note that in our work some of the more complex features of Modelica have not been involved. Work has been limited to models that result in purely continuous time equation systems. Hybrid models have not been dealt with, in other words models that can contain both continuous-time and discrete-time variables and language constructs. For dealing with such models one approach could be to run the solver on the host CPU, calculate the events there, and at each time step, compute the equation system on the GPU device(s). Another approach is to use a parallel QSS solver, which is more suitable for the parallel solution of hybrid models [36].

There are also purely algorithmic models that consist of imperative code and

[1] An approach of inlining the solver and replicating code has also been tried, which was described briefly in Chapter 2 and is described in more detail in [55]

data-parallel models. These kinds of models are different from the above and the previous discussion does not concern these models. This is instead this is discussed below.

12.2.1 Discussion on the Various Approaches of Simulating Modelica Models on GPUs

In this section the various approaches from the different chapters of the thesis are discussed. It is important to note that although the general problem of solving an ODE or DAE equation system may not be very data-parallel in nature, there are important subsets of models that contain data-parallel features where the use of GPUs is suitable.

- *Simulation of Equation-Based Models with Task Graph Creation on Graphics Processing Units*: This approach might work for equation systems where there is little dependency between different parts of the equation system, where little communication is needed between different streaming multiprocessors. However in the general case a task graph arising from an equation-based models is not necessarily data-parallel in nature, thus it is not easy to map a task graph for execution with a GPU since communication between different parts of the equation system is needed. Moreover, the volume of memory transfers taking place between the CPU and the GPU might be time consuming.

- *Simulation of Equation-Based Models with Quantized State Systems on Graphics Processing Units*: The approach presented in this thesis was an attempt to compile Modelica to QSS-based code on a GPU, with a small and simplified model. Simulation times were poor because of the volume of memory transfers taking place. Although QSS-based simulations might be suitable for parallel executions we still have the same problem with GPUs as with time-stepping methods: in general the computation of the equation system is not a data-parallel task, which GPUs are good at. Even with the QSS-based method, when updating the states, the same equation system must be solved, even though *when* to solve this equation system is different from a time-stepping method. But the QSS methods have the advantage that each state variable is computed more independently.

- *TLM Component-Based Partitioning*: The approach with TLM component-based partitioning has not yet been tried with GPUs by our research group PELAB. TLM-based component partitioning was described briefly in Chapter 3. One approach could be to put each partitioned sub-model on a streaming multiprocessor. The computations on the streaming multiprocessors could run fairly independently given that the partition of the original model results in sub-models that are rather independent.

- *Compilation of Unexpanded Modelica Array Equations for Efficient Simulation on Graphics Processing Units*: In the work described in Chapter 9 and Chapter 7 a restricted subset of Modelica models was investigated. The main focus was on models that are data-parallel in nature, or have features that are data-parallel in nature. GPUs are suitable for these kind of models.
- *Extending the Algorithmic Subset of Modelica with Explicit Parallel Programming Constructs for Multi-core Simulation*: In Chapter 8 purely algorithmic models that were data-parallel in nature were investigated. GPUs are suitable for these kind of models.

12.3 Discussion on Modeling of Partial Differential Equations Modeling with Modelica

Rather extensive research has been carried out on the topic of PDEs in the context of Modelica. The most notable work is [76]. This work was carried out in the PELAB research group and the goal was to extend the core Modelica language with constructs for PDE modeling as well as connecting external PDE solvers. In Part 3 of this thesis an external C++ PDE solver was connected with a Modelica environment via FMI thus opening up for ODE/DAE and PDE systems to be modeled and simulated in the same context. The approach in [76], however was a more extensive. Our approach in this thesis has certain advantages.

- HiFlow3 is well maintained and has strong support and capabilities for PDE modeling and solving,
- HiFlow3 and OpenModelica are free to download and use,
- The PDE structure is not lost but is maintained throughout the actual runtime simulation process. This allows for mesh refinement, solver runtime adjustments, etc.,
- PDE and DAE systems can be mixed in the same simulation setup. This is also possible in [76].

The approach in [76] has many of the same advantages but the FEM package in that work was not as extensive and well-maintained as HiFlow3. Moreover, the FMI interface puts several limitations on communication with DAE solving code and PDE solving code, limitations that are difficult to remedy.

12.4 Discussion on Skeletons and Parallel Patterns in the context of Modelica

Using skeletons and parallel patterns in the context of Modelica is novel research. In this work an approach with building a Modelica-library and calling on C++-skeleton code via external C++ was used. A few language extensions have also been approached, in other words changes in the OpenModelica compiler. In the measurements section it was shown that 1.) the execution times of running C++ with SkePU are comparable with Modelica with the Modelica-SkePU library; and 2.) speedup can be gained by running Modelica code with Modelica-SkePU and with OpenMP compared to without OpenMP.

In this chapter we have investigated using SkePU skeleton programming with the equation-based modeling and simulation language Modelica, using a method with external C++ objects. Skeleton programming is advantageous as parallelism and synchronization comes almost for free for the skeleton-based expression. Furthermore, we leverage the target architectural features. We have described the implementation and provided measurements of examples from the SkePU 1.1 ported into Modelica. To the best of our knowledge, this is the first attempt at merging skeleton programming with Modelica. We believe that this holds promise for the future.

More work is needed with the Modelica-SkePU library for full coverage of the SkePU 1.1 library. SkePU-Modelica is useful as an addition to the Modelica environment: new and strong capabilities for algorithm development. In other words in cases when modeling and simulation are needed, SkePU-Modelica offers new and powerful constructs. However in many cases of computation, SkePU-C++ is enough.

Chapter 13

Future Work

In this chapter future work from the previous parts (I, II, III and IV) of this thesis is discussed. In Section 13.1 future work with simulation of Modelica models on GPUs is discussed. In Section 13.2 future work with PDE modeling in Modelica is discussed. In Section 13.3 future work of Skeleton programming with Modelica is discussed.

13.1 Simulation of Modelica Models on GPUs

As discussed earlier, the general problem of solving an ODE or DAE equation system may not be very data-parallel in nature. There are however important subsets of models that contain data-parallel features where the use of GPUs are suitable. These are mainly models that contain operations over large arrays of state variables (or other variables). For such models it is suitable to compile the array operations directly into GPU-based code. A related approach is to search for data-parallelism in the resulting compiled equation system, i.e. to reconstruct array operations from sets of similar operations on scalar variables (array elements). However, it is more efficient for the compiler to directly compile array operations than to later reconstruct them from scalars.

One kind of Modelica model that could be of interest for simulation on GPUs are models containing Partial Differential Equations (PDEs). The Modelica language standard does not currently include constructs for modeling PDEs. However, such constructs were proposed in [76] where PDEs in the context of Modelica were extensively discussed. Currently PDEs can be modeled in Modelica via a discretization approach using for-equations; the WaveEquationSample model in Listing 6.1 is an example of such a model. These kind of models are almost always highly data-parallel in nature.

A possible area of interest in future work could be trying to classify the

Modelica models that are suitable for simulation with GPUs, and those models where a different simulation architecture would be more suitable. A question then is whether this classification should be done by the front-end part of the compiler, before all the structure is removed, or later in the compilation process when the whole equation system is available as one system. Applying machine-learning techniques could be an area of interest in future work. With machine learning techniques computers use empirical data to learn various behaviors. The goal would be to run OpenModelica with many different models and the compiler could then learn what kind of architecture is most suitable for a particular kind of model. [65]

It is important to note that GPGPU is rapidly evolving. CUDA for instance, now supports function pointers, recursion, C++ templates, virtual methods, etc. as noted in Chapter 2. But as David Black-Schaffer one of the developers of OpenCL notes [30], the underlying hardware architecture is still one optimized for data-parallel problems. He proposes the following check list for determining whether an application is suitable for implementation on GPUs.

- Is the application data-parallel?
- Is the application computationally intensive?
- Do you want to avoid global synchronization?
- Does the application require considerable bandwidth?
- Is the use of small caches acceptable for the application?
- Does the application utilize single precision?[1]

Regarding the ongoing discussion of whether to use CPUs or GPUs, in [87] it is claimed that the gap in performance between CPUs and GPUs is overestimated and it is suggested that the performance gap can be decreased, which is demonstrated for a set of example applications, provided the right optimization techniques are applied to the CPU implementation. Perhaps future generations of CPUs and GPUs will converge towards each other.

13.2 PDE Modeling with Modelica

Regarding future work on PDE modeling with Modelica.

- Discussions are ongoing regarding activating the PDE language extensions that were presented in [76],

[1] For GPGPUs with good double precision support, which is becoming more and more common, this check is not needed.

- PDE language extensions have not yet been included in the Modelica language specification, which hampers implementation efforts,
- More future effort should be put into work with connecting an external C++ PDE solver (e.g. HiFlow3) to a Modelica environment, in particular when it comes to multi-core computing.

In Part 4 of this thesis a method was investigated of incorporating PDEs in the context of a Modelica model was investigated, by using FMI to import a PDE solver from the finite element library HiFlow3. Numerical results were obtained from two scenarios:

1. The distribution of heat in a piece of copper where the heat source was controlled by a PID-controller.
2. A simple coupled model that invokes a force and measures the elasticity deformation of a beam demoefficient.

The main advantages of this type of coupling include its simplicity and the possibility of using existing solver technology on multi-core and distributed memory architectures.

The results section contains certain runtime measurements of these parallel computations, and comparisons to single-core computations were provided. Induced by the need for compiling the PDE component into a dynamically loaded shared object and loading it using the Modelica compiled model code, limitations in the MPI library influenced parallel performance. However, the speedup obtained is considerable in practical simulation, and the use of distributed memory architectures is a clear advantage with respect to memory, especially for large scale problems.

13.3 Skeleton and Parallel Pattern Programming in the Context of Modelica

Regarding future work with skeleton and parallel-pattern programming in the context of Modelica:

- It will be difficult to convince the Modelica language design group to add language extensions with skeleton and parallel patterns to the Modelica language specification. This because it has low priority compared to many other things, PDE language extensions.
- Skeletons and parallel patterns are very suitable for use in the context of Modelica when the modeler wishes to perform computationally-heavy matrix and vector computations. The SkePU library for instance has strong support for the use of multi-core architectures.
- In Part 4 measurements were provided showing the method suitable.

13.3.1 Overview of the FastFlow Parallel Pattern Framework

Fastflow is a C++ class library. The main idea of FastFlow is to provide application designers with suitable high-level, parallel programming abstractions. A powerful runtime system comes with the implementation. FastFlow has several advantages for the programmer, e.g. shorter time-to-market, portability, efficiency and performance portability. FastFlow targets streaming and data parallelism. FastFlow can be used both by application programmers and system programmers. The application programmer can select appropriate patterns from those provided. The task of the application programmer is then to connect them to obtain a suitable streaming network. The system programmer on the other hand, can use the low-level building blocks to create new patterns. The system programmer can also use and optimize the composition of existing patterns. In this way, new skeletons can be built for a specific target platform. Designing FastFlow in different layers has two main purposes: 1) to achieve flexibility and efficiency when programming multi- and many-core platforms; 2) to promote high-level parallel programming.

FastFlow is made up of three different layers.

- *Building Blocks:* This is the lowest level layer. It contains the wrapper nodes derived from ffnode. These nodes allow existing code to be embedded into parallel applications. This level also contains the one-to-many, many-to-one and feedback combinators for connecting nodes and routing data in different ways.
- *Core Parallel Patterns:* This is the next level after the building block level. This level contains the pipeline and several forms of the task-farm skeleton. The code from the building block level have been used to implement these skeletons. The pipeline and task-farm skeleton can then be nested and composed in different ways.
- *High-Level Parallel Patterns:* This is the top level layer and highest level of abstraction. Example of patterns from this level are ParallelFor Map, ParallelReduce, Stencil, ParallelSearch, MacroDataflow, DC and Pool Evolution. The application programmer uses the pipeline and task-farm core patterns for composition of the patterns available at this level.

Application Examples

A brief example of the use of FastFlow is shown in the following two listings.

Listing 13.1: *Run function of the HeatSolver class.*

```
#include <ff/pipeline.hpp>
using namespace ff;

int main() {
  ff_pipeline pipe;

    for (int i=0;i<nStages;++i) {
    pipe.add_stage(new Stage());
    if (pipe.run() and wait_end()<0)
    return -1;
    }

    return 0;
}

class Stage: public ff_node {
  int svc_init() {
    printf("Stage \%dnn",get_my_id());
    return 0;
  }

  void* svc(void* task) {
    if (ff_node::get_my_id()==0)
            for (long i=0;i<ntasks;++i)
                ff_send_out(i);
        else printf("Task=\%dnn",(long)task);

        return task;
  }
}
```

Listing 13.2: *Run function of the HeatSolver class.*

```
#include <ff/farm.hpp>
using namespace ff;

int main() {
    farm<>* farm;
    std::vector<ff_node*> workers;

    for(int i=0;i<nWorkers;++i)
        workers.push_back(new Worker);

    farm.add_workers(workers);
    farm.add_emitter(new Emitter(nTasks));
    farm.add_collector(new Collector);

    if (farm.run and wait_end()<0)
        return -1;

    return 0;
}

struct Emitter : public *node {
Emitter(int ntask):ntask(ntask)fg
  int svc_init() f
```

```
  printf ("Work Start\n");
}

void* svc(void *) {
  long task = new task_t(ntask*);
  return (void*)task;
 }

 long ntask;
};

struct Worker : public ff_node {
    void* svc(void * task) {
    // do something useful with the task

    return task;
    }
};

struct Collector : public ff_node {

    void* svc(void* task) {
        printf("Task=%d\n",(long)task);
        delete task;
        return GO_ON;
    }

    void svc_end() { printf("Done!\n"); }
};
```

Part VI

Appendix

Appendix A

QSS Generated CUDA Code

The following code examples contain the model-dependent part of the code from the experiment presented in Chapter 5 with 8 state variables, where three output variables are defined. The original Modelica model is in Listing A.1.

Listing A.1: *Test Model*

```
model Test_Model
    parameter Integer N = 8;
    input Real inputVars[1](start = 0.0);
    Real stateVars[N](start = 0.0);
    output Real outputVars[3];
equation
    der(stateVars[1]) = N*N * (-2.0*stateVars[1] +
        stateVars[2] + inputVars[1]);
    for i in 2:(N-1) loop
        der(stateVars[i]) = N*N * (-2.0*stateVars[i] +
        stateVars[i-1] + stateVars[i+1]);
    end for;
    der(stateVars[N]) = N * (stateVars[N-1] -
        1000 * ((N+1)/N) * stateVars[N]);
    outputVars[1] = stateVars[1];
    outputVars[2] = stateVars[4];
    outputVars[3] = stateVars[N];
end Test_Model;
```

Two output files are produced: model.h and model.cu. The first one is a C-CUDA header file and contains the function prototypes of the routine contained in the second one.

Listing A.2: *Generated CUDA QSS Code*

```
/*********************************
* MODEL.H
*********************************/
#ifdef _MODEL_H
#define _MODEL_H
#define NUMBER_STATES 8
#define NUMBER_INPUTS 1
#define NUMBER_OUTPUT 3
#define NUMBER_EVENTS 10
#define SIMULATION_TIME 10
#define SIMULATION_STEP 0.001
/* Initializations */
void initializeSystem(float* x, float* u);
void initializeEvents(float* t, unsigned* i, float* v);

/* Derivative calculation */
__global__ void derivative
(float* dx, float* x, float* u, float* t, unsigned* c);
__device__ void dx7
(float* dx, float* x, float* u, float* t, unsigned* c);
__device__ void dx6
(float* dx, float* x, float* u, float* t, unsigned* c);
__device__ void dx5
(float* dx, float* x, float* u, float* t, unsigned* c);
__device__ void dx4
(float* dx, float* x, float* u, float* t, unsigned* c);
__device__ void dx3
(float* dx, float* x, float* u, float* t, unsigned* c);
__device__ void dx2
(float* dx, float* x, float* u, float* t, unsigned* c);
__device__ void dx1
(float* dx, float* x, float* u, float* t, unsigned* c);
__device__ void dx0
(float* dx, float* x, float* u, float* t, unsigned* c);
/* Output calculation */
__global__ void output
(float* y, float* x, float* u, float* t, unsigned* c);
__device__ void y2
(float* y, float* x, float* u, float* t, unsigned* c);
__device__ void y1
(float* y, float* x, float* u, float* t, unsigned* c);
__device__ void y0
(float* y, float* x, float* u, float* t, unsigned* c);
#endif

/*********************************
* MODEL.CU
*********************************/
#include "inclusion.h"
#include "model.h"
/* Initializations */
void initializeSystem(float *x, float* u) {
  int i;
  u[0]=0.0;
  for(i=0;i<NUMBER_STATES;i++) x[i]=0.0;
}
void initializeEvents(float* t, unsigned* i, float* v) {
  t[0] = 1; i[0] = 0; v[0] = 1;
  t[1] = 2; i[1] = 0; v[1] = 0;
  t[2] = 3; i[2] = 0; v[2] = 1;
  t[3] = 4; i[3] = 0; v[3] = 0;
  t[4] = 5; i[4] = 0; v[4] = 1;
  t[5] = 6; i[5] = 0; v[5] = 0;
  t[6] = 7; i[6] = 0; v[6] = 1;
  t[7] = 8; i[7] = 0; v[7] = 0;
  t[8] = 9; i[8] = 0; v[8] = 1;
  t[9] = 10;i[9] = 0; v[9] = 0;
}
```

```
/* Derivative calculation */
__global__ void derivative
(float* dx, float* x, float* u, float* t, unsigned* c) {
  int i = threadIdx.x;
  switch(i) {
  case 7: dx7(dx, x, u, t, c); break;
  case 6: dx6(dx, x, u, t, c); break;
  case 5: dx5(dx, x, u, t, c); break;
  case 4: dx4(dx, x, u, t, c); break;
  case 3: dx3(dx, x, u, t, c); break;
  case 2: dx2(dx, x, u, t, c); break;
  case 1: dx1(dx, x, u, t, c); break;
  case 0: dx0(dx, x, u, t, c); break;
  }
}
__device__ void dx7
(float* dx, float* x, float* u, float* t, unsigned* c) {
  dx[7] = 8.0 * (x[6] - 1000 * 1.0625 * x[7]);
}
__device__ void dx6
(float* dx, float* x, float* u, float* t, unsigned* c) {
  dx[6] = 16384.0 * (-2.0 * x[6] + x[5] + x[7]);
}
__device__ void dx5
(float* dx, float* x, float* u, float* t, unsigned* c) {
  dx[5] = 16384.0 * (-2.0 * x[5] + x[4] + x[6]);
}
__device__ void dx4
(float* dx, float* x, float* u, float* t, unsigned* c) {
  dx[4] = 16384.0 * (-2.0 * x[4] + x[3] + x[5]);
}
__device__ void dx3
(float* dx, float* x, float* u, float* t, unsigned* c) {
  dx[3] = 16384.0 * (-2.0 * x[3] + x[2] + x[4]);
}
__device__ void dx2
(float* dx, float* x, float* u, float* t, unsigned* c) {
  dx[2] = 16384.0 * (-2.0 * x[2] + x[1] + x[3]);
}
__device__ void dx1
(float* dx, float* x, float* u, float* t, unsigned* c) {
  dx[1] = 16384.0 * (-2.0 * x[1] + x[0] + x[2]);
}
__device__ void dx0
(float* dx, float* x, float* u, float* t, unsigned* c) {
  dx[0] = 16384.0 * (-2.0 * x[0] + x[1] + u[0]);
}
/* Output calculation */
__global__ void output
(float* y, float* x, float* u, float* t, unsigned* c) {
  int i = threadIdx.x;
  switch(i) {
    case 2: y2(y, x, u, t, c); break;
    case 1: y1(y, x, u, t, c); break;
    case 0: y0(y, x, u, t, c); break;
  }
}
__device__ void y2
(float* y, float* x, float* u, float* t, unsigned* c) {
  y[2] = x[7];
}
__device__ void y1
(float* y, float* x, float* u, float* t, unsigned* c) {
  y[1] = x[3];
}
__device__ void y0
(float* y, float* x, float* u, float* t, unsigned* c) {
  y[0] = x[0];
```

```
}
```

Appendix B

Partial Differential Equation Modeling with Modelica via FMI Import of HiFlow3 C++ Components

The following code examples contain the Modelica-dependent part of the code from the experiment presented in Part 4. The C/C++ code of the Elasticity Solver (the HiFlow3 code) is omitted. We only show the Elasticity Solver code, since the Heat Solver code follows a very similar approach.

Listing B.1: *Code Structure*

```
krsta@mina7:src> pwd
/home/krsta/SIMS2014paper/hiflow_modelica/
   SIMS_14_stat_lin_elast_par/src
krsta@mina7:src> ls
elasticity_solver  Make.inc.template  run.sh
fmu                modelica           sim
krsta@mina7:src> cd modelica/
krsta@mina7:modelica> ls
ElasticitySolver.mo   ElasticitySolver.mos
krsta@mina7:modelica> cd ..
krsta@mina7:src> cd elasticity_solver/
krsta@mina7:elasticity_solver> ls
BEAM.inp               elasticity_test.cc
membrane_modelica.inp  elasticity_solver.cc
elasticity.xml         test1.inp
elasticity_solver.h    Makefile
test2.inp
krsta@mina7:elasticity_solver> cd ..
krsta@mina7:src> cd fmu/
```

```
krsta@mina7:fmu> ls
ElasticitySolver.c    Makefile
include               modelDescription.xml
```

Listing B.2: *run.sh*

```
#!/bin/bash

cd fmu; ./run.sh
cd ..
mkdir -p sim; cd sim
rm -rf *
cp ../elasticity_solver/elasticity.xml .
cp ../fmu/ElasticitySolver.fmu .
cp ../modelica/ElasticitySolver* .

/home/krsta/openmodelica/build/bin/omc ElasticitySolver.mos +s
```

Listing B.3: *Make.inc.template*

```
# use the MPI compiler wrapper to link MPI libraries.
CXX=mpicxx

# base directory of includes, libs etc, adjust to your machine
BASE_DIR=/home/krsta/local2

# HiFlow3 includes
HIFLOW_INC=-I$(BASE_DIR)/include/hiflow3 -I$(BASE_DIR)/include/
    hiflow3/boost/tr1

# metis graph partitioner include dir
METIS_INC=-I$(BASE_DIR)/include

# compiler flags
CXXFLAGS=$(HIFLOW_INC) $(METIS_INC) -O3 -fPIC

# linker flags
LDFLAGS=-L$(BASE_DIR)/lib -lhiflow -lmetis -fopenmp
```

Listing B.4: *modelica/ElasticitySolver.mo*

```
// ------------
// ElasticitySolver
// FMI application with HiFlow^3 block for PDE solving.
//
// Authors: Chen Song, Martin Wlotzka, Kristian Stavaker
// Main class
model ElasticitySolver
  // HiFlow^3 component
  ElasticitySolver_me_FMU hfBlock;

  // Source for signals that should be constantly 0
  Modelica.Blocks.Sources.Constant zeroSource(k=0.0);
```

```
  Real u_center(start=0.0);
  Real force(start=10.0);
equation
 connect(hfBlock.u_center, u_center);
 connect(force,hfBlock.force);
 force = 10.0;
 connect(hfBlock.der_stateVar,zeroSource.y);
 connect(hfBlock.stateVar,zeroSource.y);
end ElasticitySolver;
```

Listing B.5: *modelica/ElasticitySolver.mos*

```
loadModel(Modelica); getErrorString();
importFMU("ElasticitySolver.fmu"); getErrorString();
loadFile("ElasticitySolver_me_FMU.mo"); getErrorString();
loadFile("ElasticitySolver.mo"); getErrorString();
instantiateModel(ElasticitySolver); getErrorString();
checkModel(ElasticitySolver); getErrorString();
simulate(ElasticitySolver,startTime=0.0,stopTime=1.0,
 numberOfIntervals=10,outputFormat="plt",method="euler");
getErrorString();
//plot({hfBlock.u,hfBlock.g,PI.u_s}); getErrorString();
```

Listing B.6: *elasticitysolver/Makefile*

```
include ../Make.inc

all: elasticity_test elasticity_solver.o test.so

elasticity_test: elasticity_test.o
    $(CXX) -o elasticity_test elasticity_test.o -ldl

test.so: elasticity_solver.o
    $(CXX) -shared -o test.so elasticity_solver.o $(LDFLAGS)

%.o: %.cc
    $(CXX) $(CXXFLAGS) -o $@ -c $<

clean:
    rm -f *.o *.so *vtu *log *~ elasticity_test
```

Listing B.7: *elasticitysolver/elasticity.xml*

```
<Param>
   <OutputPathAndPrefix>elasticity_test</OutputPathAndPrefix>
   <Mesh>
     <Filename>BEAM.inp</Filename>
     <InitialRefLevel>4</InitialRefLevel>
   </Mesh>
   <LinearAlgebra>
     <Platform>CPU</Platform>
     <Implementation>Naive</Implementation>
```

```
    <MatrixFormat>CSR</MatrixFormat>
    <MatrixFreePrecond>NOPRECOND</MatrixFreePrecond>
    <Omega>2.5</Omega>
    <ILU_P>2.5</ILU_P>
  </LinearAlgebra>
  <ElasticityModel>
    <Density>2400</Density>
    <young>17e9</young>
    <poisson>0.2</poisson>
    <gravity>9.81</gravity>
  </ElasticityModel>
  <QuadratureOrder>2</QuadratureOrder>
  <FiniteElements>
    <DisplacementDegree>1</DisplacementDegree>
  </FiniteElements>
  <Instationary>
    <SolveInstationary>0</SolveInstationary>
    <Method>CrankNicolson</Method>
    <Timestep>0.05</Timestep>
    <Endtime>3.0</Endtime>
  </Instationary>
  <Boundary>
    <DirichletMaterial1>14</DirichletMaterial1>
    <DirichletMaterial2>0</DirichletMaterial2>
    <NeumannMaterial1>12</NeumannMaterial1>
  </Boundary>
  <LinearSolver>
    <MaximumIterations>1000</MaximumIterations>
    <AbsoluteTolerance>1.e-12</AbsoluteTolerance>
    <RelativeTolerance>1.e-8</RelativeTolerance>
    <DivergenceLimit>1.e6</DivergenceLimit>
    <BasisSize>1000</BasisSize>
    <Preconditioning>1</Preconditioning>
    <UseILUPP>0</UseILUPP>
  </LinearSolver>
  <ILUPP>
    <PreprocessingType>0</PreprocessingType>
    <PreconditionerNumber>11</PreconditionerNumber>
    <MaxMultilevels>20</MaxMultilevels>
    <MemFactor>0.8</MemFactor>
    <PivotThreshold>2.75</PivotThreshold>
    <MinPivot>0.05</MinPivot>
  </ILUPP>

  <Backup>
    <Restore>0</Restore>
    <LastTimeStep>160</LastTimeStep>
    <Filename>backup.h5</Filename>
  </Backup>
</Param>
```

Listing B.8: *fmu/Makefile*

```
include ../Make.inc

INC=-I./include -I../elasticity_solver

all: ElasticitySolver.fmu

ElasticitySolver.fmu: ElasticitySolver.so
    rm -rf fmu
    mkdir -p fmu/binaries/linux64
    mkdir fmu/sources
```

```
    cp ElasticitySolver.so fmu/binaries/linux64/
    cp elasticity_solver.cc fmu/sources/
    cp elasticity_solver.h fmu/sources/
    cp ElasticitySolver.c fmu/sources/
    cp modelDescription.xml fmu/
    cp ../elasticity_solver/membrane_modelica.inp fmu/
    cp ../elasticity_solver/elasticity.xml fmu/
    (cd fmu; zip -r ../ElasticitySolver.fmu *)

ElasticitySolver.so: ElasticitySolver.o elasticity_solver.o
    $(CXX) -shared -Wl,-soname,ElasticitySolver.so -o $@ $+ $(LDFLAGS
        )

ElasticitySolver.o: ElasticitySolver.c
    $(CXX) $(CXXFLAGS) $(INC) -o $@ -c $<

elasticity_solver.o: elasticity_solver.cc
    $(CXX) $(CXXFLAGS) -o $@ -c $<

clean:
    rm -f *~ *.fmu *.so *.o
```

Listing B.9: *fmu/modelDescription.xml*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<fmiModelDescription
fmiVersion="1.0"
modelName="ElasticitySolver"
modelIdentifier="ElasticitySolver"
guid="{8c4e810f-3df3-4a00-8276-176fa3c9f003}"
numberOfContinuousStates="1"
numberOfEventIndicators="0">
<ModelVariables>
<ScalarVariable name="u_center" valueReference="0"
 description="output value of HiFlow3 block" causality="output">
<Real/>
</ScalarVariable>
<ScalarVariable name="force" valueReference="1"
 description="input value of HiFlow3 block" causality="input">
<Real/>
</ScalarVariable>
<ScalarVariable name="stateVar" valueReference="2"
 description="state variable" causality="input">
<Real/>
</ScalarVariable>
<ScalarVariable name="der_stateVar" valueReference="3"
 description="state variable derivative" causality="input">
<Real/>
</ScalarVariable>
</ModelVariables>
</fmiModelDescription>
```

Listing B.10: *fmu/ElasticitySolver.c*

```
* ----------------------------------------------------------------*
  * Sample implementation of an FMU - a bouncing ball.
  * This demonstrates the use of state events and reinit of states.
  * Equations:
```

```
 *
 * (c) 2010 QTronic GmbH
 * ----------------------------------------------------------------*/

// define class name and unique id
#define MODEL_IDENTIFIER ElasticitySolver
#define MODEL_GUID "{8c4e810f-3df3-4a00-8276-176fa3c9f003}"

// define model size
#define NUMBER_OF_REALS 4
#define NUMBER_OF_INTEGERS 0
#define NUMBER_OF_BOOLEANS 0
#define NUMBER_OF_STRINGS 0
#define NUMBER_OF_STATES 1
#define NUMBER_OF_EVENT_INDICATORS 0

// include fmu header files, typedefs and macros
#include "elasticity_solver.h"

extern "C" {
#include "fmuTemplate.h"
}

// define all model variables and their value references
// conventions used here:
// - if x is a variable,
// then macro x_ is its variable reference
// - the vr of a variable is its index in array
// r, i, b or s
// - if k is the vr of a real state, then k+1 is
// the vr of its derivative
#define u_center_          0
#define force_             1
#define stateVar_          2
#define der_stateVar_      3

// define initial state vector as vector
// of value references
#define STATES { stateVar_ }

// called by fmiInstantiateModel
// Set values for all variables that define
// a start value
// Settings used unless changed by
// fmiSetX before fmiInitialize
void setStartValues(ModelInstance *comp) {
  r(u_center_) = 0.0;
  r(force_) = 0.0;
  r(stateVar_) = 0.0;
  r(der_stateVar_)=0.0;
}

// called by fmiGetReal, fmiGetContinuousStates
// and fmiGetDerivatives
fmiReal getReal(ModelInstance* comp, fmiValueReference vr){
  switch(vr) {
  case u_center_:
       return (fmiReal)HiFlow3_PDE_COMPONENT(0,NULL,r(force_));
  case force_: return r(force_);
  case stateVar_: return r(stateVar_);
  case der_stateVar_: return r(der_stateVar_);
  default: return 0.0;
  }
}

// called by fmiGetReal, fmiGetContinuousStates and
// fmiGetDerivatives
fmiReal getInteger(ModelInstance* comp, fmiValueReference vr){
```

```
  return 0;
}

// called by fmiInitialize() after setting eventInfo to defaults
// Used to set the first time event, if any.
void initialize(ModelInstance* comp, fmiEventInfo* eventInfo) {
  /*    eventInfo->upcomingTimeEvent   = fmiTrue;
    eventInfo->nextEventTime       = 0.1 + comp->time;*/
}

// Used to set the next time event, if any.
void eventUpdate(ModelInstance* comp, fmiEventInfo* eventInfo) {
  /*i(counter_) += 1;
  r(u_)=(fmiReal)preRun(0,NULL,r(g_),r(t_));
  if (i(counter_) == 1000)
    eventInfo->terminateSimulation = fmiTrue;
  else {
    eventInfo->upcomingTimeEvent   = fmiTrue;
    eventInfo->nextEventTime       = 0.1 + r(t_);
  }*/
}

// include code that implements the FMI based
// on the above definitions
extern "C" {
#include "fmuTemplate.c"
}
```

Appendix C

Modelica-SkePU Library Code

C.1 Modelica-SkePU Library Code

C.1.1 Modelica-SkePU Test Suite Models - Serial

Mandelbrot Fractals

Parameters width 4000 and height 3000.

C++-SkePU	0.927197s	0.93655s	0.936864s
Modelica-SkePU	1.738816s	1.690458s	1.744684s

LU Factorization

Parameter N 1000.

C++-SkePU	1m 36.18017s	1m 36.17789s	1m 36.14924s
Modelica-SkePU	1m 36.07740s	1m 36.11069s	1m 36.11236s

Mean Square Error (MSE)

Parameters rows 1600 and cols 1200.

C++-SkePU	0.1193780s	0.1138280s	0.113955s
Modelica-SkePU	0.08788500s	0.08790700s	0.08786200s

Pearson Product-Moment Correlation Coefficient (PPMCC)

Parameter N 50000.

C++-SkePU	0.010036s	0.012182s	0.010055s
Modelica-SkePU	0.00656400s	0.00590200s	0.0058400s

Peak Signal to Noise Ratio (PSNR)

Parameters rows 1600 and cols 1200.

C++-SkePU	0.1088490s	0.110301s	0.108632s
Modelica-SkePU	0.08850200s	0.08794100s	0.08845300s

Taylor Series Calculation

Parameter N 100000.

C++-SkePU	0.0138850s	0.012422s	0.013897s
Modelica-SkePU	0.00950500s	0.00940400s	0.00939900s

Smooth Particle Hydrodynamics (SPH), Fluid Dynamics Shock-tube simulation

Parameter XYLEN 30, timesteps = 100, NTRIALS = 3.

C++-SkePU	2.423772s	2.423721s	2.423851s
Modelica-SkePU	7.29832s	7.291270s	7.296242s

A Runge-Kutta ODE solver

Parameters N 16, H 4, DOPRI5, BRUSS2D-MIX.

C++	0.001951s	0.001957s	0.001944s
Modelica-SkePU	0.504022s	0.512004s	0.517614s

C.1.2 Modelica-SkePU Test Suite Models - Parallel

Mandelbrot Fractals

Parameters width 1000 and height 1000.

OPENMP:	1.379900e-01	1.353170e-01	1.372530e-01
SERIAL:	1.471710e-01	1.471210e-01	1.466670e-01

Parameters width 2500 and height 2000.

OPENMP:	6.810950e-01	6.806280e-01	6.855220e-01
SERIAL:	7.368220e-01	7.346270e-01	7.375890e-01

Parameters width 4000 and height 3000.

OPENMP:	1.647204e+00	1.602092e+00	1.655988e+00
SERIAL:	1.754251e+00	1.812344e+00	1.798421e+00

Parameters width 5500 and height 4000.

OPENMP:	2.969644e+00	2.950355e+00	2.983768e+00
SERIAL:	3.220907e+00	3.253921e+00	3.261657e+00

LU Factorization

Parameter N 250.

OPENMP:	1.305080e+00	1.305720e+00	1.305917e+00
SERIAL:	1.501272e+00	1.501086e+00	1.500983e+00

Parameter N 500.

OPENMP:	1.044757e+01	1.045716e+01	1.045651e+01
SERIAL:	1.198756e+01	1.198806e+01	1.198762e+01

Parameter N 750.

OPENMP:	3.539186e+01	3.541675e+01	3.542214e+01
SERIAL:	4.058260e+01	4.058430e+01	4.056709e+01

Parameter N 1000.

OPENMP:	5.103503e+01	5.104907e+01	5.104988e+01
SERIAL:	7.722086e+01	7.721975e+01	7.722822e+01

Mean Square Error (MSE)

Parameters rows 1000 and cols 600.

OPENMP:	1.303000e-02	1.299300e-02	1.336500e-02
SERIAL:	2.744800e-02	2.743200e-02	2.741200e-02

Parameters rows 1200 and cols 800.

OPENMP:	2.097000e-02	2.084800e-02	2.082600e-02
SERIAL:	4.407000e-02	4.407200e-02	4.408900e-02

Parameters rows 1400 and cols 1000.

OPENMP:	3.049500e-02	3.037600e-02	3.027100e-02
SERIAL:	6.426000e-02	6.420300e-02	6.421400e-02

Parameters rows 1600 and cols 1200.

OPENMP:	4.183100e-02	4.155200e-02	4.158500e-02
SERIAL:	8.791300e-02	8.796500e-02	8.803600e-02

Pearson Product-Moment Correlation Coefficient (PPMCC)

Parameter N 30000.

OPENMP:	1.920000e-04	1.920000e-04	1.910000e-04
SERIAL:	3.503000e-03	3.466000e-03	3.460000e-03

Parameter N 40000.

OPENMP:	2.790000e-04	2.530000e-04	2.520000e-04
SERIAL:	5.094000e-03	4.616000e-03	4.627000e-03

Parameter N 50000.

OPENMP:	4.950000e-04	4.930000e-04	4.930000e-04
SERIAL:	5.728000e-03	5.889000e-03	5.769000e-03

Parameter N 60000.

OPENMP:	6.230000e-04	5.850000e-04	5.920000e-04
SERIAL:	6.971000e-03	6.954000e-03	6.834000e-03

Peak Signal to Noise Ratio (PSNR)

Parameters rows 1000 and cols 600.

OPENMP:	1.306200e-02	1.302500e-02	1.301000e-02
SERIAL:	2.745600e-02	2.751400e-02	2.752900e-02

Parameters rows 1200 and cols 800.

OPENMP:	2.087800e-02	2.084400e-02	2.081600e-02
SERIAL:	4.402400e-02	4.404000e-02	4.406600e-02

Parameters rows 1400 and cols 1000.

OPENMP:	3.035200e-02	3.027400e-02	3.028400e-02
SERIAL:	6.421400e-02	6.414200e-02	6.415400e-02

Parameters rows 1600 and cols 1200.

OPENMP:	4.190200e-02	4.171100e-02	4.176000e-02
SERIAL:	8.801100e-02	8.912000e-02	8.804000e-02

Taylor Series Calculation

Parameter N 50000.

OPENMP:	1.219000e-03	1.228000e-03	1.218000e-03
SERIAL:	6.297000e-03	5.282000e-03	4.927000e-03

Parameter N 75000.

OPENMP:	1.601000e-03	1.638000e-03	1.683000e-03
SERIAL:	8.578000e-03	6.982000e-03	7.054000e-03

Parameter N 100000.

OPENMP:	2.570000e-03	2.478000e-03	2.421000e-03
SERIAL:	9.687000e-03	9.744000e-03	9.849000e-03

Parameter N 125000.

OPENMP:	3.190000e-03	3.090000e-03	3.007000e-03
SERIAL:	1.130200e-02	1.189900e-02	1.116600e-02

Smooth Particle Hydrodynamics (SPH), Fluid Dynamics Shock-tube simulation

Parameter XYLEN 15, timesteps = 100, NTRIALS = 3.

OPENMP:	4.638850e-01	4.611760e-01	4.611330e-01
SERIAL:	4.778220e-01	4.765230e-01	4.741000e-01

Parameter XYLEN 20, timesteps = 100, NTRIALS = 3.

OPENMP:	1.450278e+00	1.445756e+00	1.446441e+00
SERIAL:	1.441323e+00	1.485951e+00	1.482929e+00

Parameter XYLEN 25, timesteps = 100, NTRIALS = 3.

OPENMP:	3.574688e+00	3.562569e+00	3.561548e+00
SERIAL:	3.529829e+00	3.522551e+00	3.523077e+00

Parameter XYLEN 30, timesteps = 100, NTRIALS = 3.

OPENMP:	7.415329e+00	7.409102e+00	7.407670e+00
SERIAL:	7.334901e+00	7.330488e+00	7.338122e+00

A Runge-Kutta ODE solver

Parameters N 8, H 4, DOPRI5, BRUSS2D-MIX.

OPENMP:	3.132000e-02	2.804500e-02	2.957600e-02
SERIAL:	1.085340e-01	1.092920e-01	1.096190e-01

Parameters N 12, H 4, DOPRI5, BRUSS2D-MIX.

OPENMP:	3.893900e-02	3.933800e-02	4.287900e-02
SERIAL:	2.254880e-01	2.217240e-01	2.197470e-01

Parameters N 16, H 4, DOPRI5, BRUSS2D-MIX.

OPENMP:	5.408900e-02	5.603100e-02	5.290500e-02
SERIAL:	4.788900e-01	4.818230e-01	4.716120e-01

Parameters N 20, H 4, DOPRI5, BRUSS2D-MIX.

OPENMP:	7.059900e-02	7.559600e-02	6.442000e-02
SERIAL:	7.420770e-01	7.336770e-01	7.719100e-01

C.2 Modelica-SkePU Library Code

Listing C.1: *skepu$_h$eader.h C++*

```
//*****************************************************************
// skepu_header.h
// Author: Kristian Stavaker, kristian.stavaker@liu.se
```

```
// Description: Header file that can be read by both C and C++.
//*****************************************************************
//#define SKEPU_OPENMP

#ifndef EXTERNAL_CODE_H
#define EXTERNAL_CODE_H

#ifdef __cplusplus
extern "C" {
#endif

#if defined(__STDC__) || defined(__cplusplus)
  extern void* initMyMatrix(char* type1,int dim1, int dim2, double
      initValue);
  extern void closeMyMatrix(void* object);
  extern void* initMyVector(char* type1,int dim1, double initValue);
  extern void closeMyVector(void* object);
  extern double getMatrixElement(void* object, int a1, int a2);
  extern double getVectorElement(void* object, int a1);
  extern void assignMatrixElement(void* object, int a1, int a2, double
       elem);
  extern void assignVectorElement(void* object, int a1, double elem);
  extern void vectorUpdatehost(void *object);
  extern void displayDataVector(void* object,char* type1);
  extern void displayDataMatrix(void* object,char* type1);
  extern void* initMyReduce(char* funcName);
  extern void closeMyReduce(void* object);
  extern void* initMyMapReduce(char* funcName1, char* funcName2);
  extern void closeMyMapReduce(void* object);
  extern void* initMyScan(char* funcName);
  extern void closeMyScan(void* object);
  extern void* initMyMap(char* funcName);
  extern void closeMyMap(void* object);
  extern void* initMyMapArray(char* funcName);
  extern void closeMyMapArray(void* object);
  extern void* initMyMapOverlap(char* funcName);
  extern void closeMyMapOverlap(void* object);
  extern void* initMyGenerate(char* funcName);
  extern void closeMyGenerate(void* object);
  extern void SkePU_Map_builtin_externalMdouble(void* object1, void*
      object2, void* object3);
  extern void SkePU_Map_builtin_externalV1double(void* object1, void*
      object2);
  extern void SkePU_Map_builtin_externalVdouble(void* object1, void*
      object2, void* object3, char* type1);
  extern void SkePU_Map_builtin_externalV2double(void* object1, void*
      object2, void* object3, void* object4, char* type1);
  extern void SkePU_Map_builtin_setConstant(void* object1, double val1
      );
  extern void SkePU_MapArray_builtin_externalVdouble(void* object1,
      void* object2, void* object3, void* object4, char* type1);
  extern void SkePU_MapArray_builtin_externalVMMdouble(void* object1,
      void* object2, void* object3, void* object4);
  extern void SkePU_MapOverlap_builtin_externalVdouble(void* object1,
      void* object2, double object3, void* object4);
  extern void SkePU_Scan_builtin_externalVdouble(void* object1, void*
      object2, void* object3);
  extern double SkePU_MapReduce_builtin_externalVdouble(void* object1,
       void* object2, void* object3);
  extern double SkePU_MapReduce_builtin_externalV1double(void* object1
      , double inVal, void* object3);
  extern double SkePU_MapReduce_builtin_externalMdouble(void* object1,
       void* object2, void* object3);
  extern double SkePU_Reduce_builtin_externalMdouble(void* object1,
      void* object2);
  extern double SkePU_Reduce_builtin_externalVdouble(void* object1,
      void* object2);
```

```
  extern void SkePU_Generate_builtin_externalVdouble(int numElem, void
      * object1, void* object2, char* type1);
  extern void SkePU_Generate_builtin_setConstant(void* object1,int
      elem);
  extern double getRandNum();
#else
  extern void* initMyMatrix(char* type1,int dim1, int dim2, double
      initValue);
  extern void closeMyMatrix(void* object);
  extern void* initMyVector(char* type1,int dim1, double initValue);
  extern void closeMyVector(void* object);
  extern double getMatrixElement(void* object, int a1, int a2);
  extern double getVectorElement(void* object, int a1);
  extern void assignMatrixElement(void* object, int a1, int a2, double
       elem);
  extern void assignVectorElement(void* object, int a1, double elem);
  extern void vectorUpdatehost(void *object);
  extern void displayDataVector(void* object,char* type1);
  extern void displayDataMatrix(void* object,char* type1);
  extern void* initMyReduce(char* funcName);
  extern void closeMyReduce(void* object);
  extern void* initMyMapReduce(char* funcName1, char* funcName2);
  extern void closeMyMapReduce(void* object);
  extern void* initMyScan(char* funcName);
  extern void closeMyScan(void* object);
  extern void* initMyMap(char* funcName);
  extern void closeMyMap(void* object);
  extern void* initMyMapArray(char* funcName);
  extern void closeMyMapArray(void* object);
  extern void* initMyMapOverlap(char* funcName);
  extern void closeMyMapOverlap(void* object);
  extern void* initMyGenerate(char* funcName);
  extern void closeMyGenerate(void* object);
  extern void SkePU_Map_builtin_externalMdouble(void* object1, void*
      object2, void* object3);
  extern void SkePU_Map_builtin_externalV1double(void* object1, void*
      object2);
  extern void SkePU_Map_builtin_externalVdouble(void* object1, void*
      object2, void* object3, char* type1);
  extern void SkePU_Map_builtin_externalV2double(void* object1, void*
      object2, void* object3, void* object4, char* type1);
  extern void SkePU_Map_builtin_setConstant(void* object1, double val1
      );
  extern void SkePU_MapArray_builtin_externalVdouble(void* object1,
      void* object2, void* object3, void* object4, char* type1);
  extern void SkePU_MapArray_builtin_externalVMMdouble(void* object1,
      void* object2, void* object3, void* object4);
  extern void SkePU_MapOverlap_builtin_externalVdouble(void* object1,
      void* object2, double object3, void* object4);
  extern void SkePU_Scan_builtin_externalVdouble(void* object1, void*
      object2, void* object3);
  extern double SkePU_MapReduce_builtin_externalVdouble(void* object1,
       void* object2, void* object3);
  extern double SkePU_MapReduce_builtin_externalV1double(void* object1
      , double inVal, void* object3);
  extern double SkePU_MapReduce_builtin_externalMdouble(void* object1,
       void* object2, void* object3);
  extern double SkePU_Reduce_builtin_externalMdouble(void* object1,
      void* object2);
  extern double SkePU_Reduce_builtin_externalVdouble(void* object1,
      void* object2);
  extern void SkePU_Generate_builtin_externalVdouble(int numElem, void
      * object1, void* object2, char* type1);
  extern void SkePU_Generate_builtin_setConstant(void* object1,int
      elem);
  extern double getRandNum();
#endif
```

```
#ifdef __cplusplus
}
#endif

#endif /*EXTERNAL_CODE_H*/
```

Listing C.2: *skepu$_c$pp.cpp C++*

```
//*****************************************************************
// skepu_cpp.cpp
// Author: Kristian Stavaker, kristian.stavaker@liu.se
// Description: C++ file that uses the C++ SkePU library
//*****************************************************************
#include <iostream>
//#define SKEPU_OPENMP
#include "skepu/matrix.h"
#include "skepu/vector.h"
#include "skepu/reduce.h"
#include "skepu/mapreduce.h"
#include "skepu/scan.h"
#include "skepu/map.h"
#include "skepu/maparray.h"
#include "skepu/mapoverlap.h"
#include "skepu/generate.h"
#include "skepu_header.h"
#include "skepu_macro_functions.h"
#include "math.h"

#define QUOTE(abc) \
  QUOTE2(abc)

#define QUOTE2(abc) \
  #abc


class myStruct {
public:
  void *skel;
  int funcNum1;
  int funcNum2;
};

// MATRIX AND VECTOR FUNCTIONS
void* initMyMatrix(char* type1, int dim1, int dim2, double initValue)
    {
  void *m;

  if (!strcmp(type1,"double")) m = new skepu::Matrix<double>(dim1,dim2
      ,initValue);
  if (!strcmp(type1,"int")) m = new skepu::Matrix<int>(dim1,dim2,0);
  if (!strcmp(type1,QUOTE(TYPE_NAME1))) m = new skepu::Matrix<
      TYPE_NAME1>(dim1,dim2);
  if (!strcmp(type1,QUOTE(TYPE_NAME2))) m = new skepu::Matrix<
      TYPE_NAME2>(dim1,dim2);

  if ( m == NULL ) std::cout << "Not enough memory or wrong type (
      initMyMatrix)!" << std::endl;
  // read table from file and store all data in *table
  return (void*) m;
}

void closeMyMatrix(void* object) { /* Release table storage */
  free(object);
```

```
}

void* initMyVector(char* type1, int dim1, double initValue) {
  void *v;

  if (!strcmp(type1,"double")) v = new skepu::Vector<double>(dim1,
       initValue);
  if (!strcmp(type1,"int")) v = new skepu::Vector<int>(dim1,0);
  if (!strcmp(type1,QUOTE(TYPE_NAME1))) v = new skepu::Vector<
       TYPE_NAME1>(dim1);
  if (!strcmp(type1,QUOTE(TYPE_NAME2))) v = new skepu::Vector<
       TYPE_NAME2>(dim1);

  if ( v == NULL ) std::cout << "Not enough memory or wrong type (
       initMyVector)!" << std::endl;
  // read table from file and store all data in *table
  return (void*) v;
}

void closeMyVector(void* object) { /* Release table storage */
  free(object);
}

double getMatrixElement(void* object, int a1, int a2) {
  skepu::Matrix<double>* mat = (skepu::Matrix<double>*) object;

  return (*mat)[a1,a2];
}

double getVectorElement(void* object, int a1) {
  skepu::Vector<double>* vec = (skepu::Vector<double>*) object;

  return (*vec)[a1];
}

void assignMatrixElement(void* object, int a1, int a2, double elem) {
  skepu::Matrix<double> *mat = (skepu::Matrix<double>*) object;
  (*mat)[a1,a2] = elem;
}

void assignVectorElement(void* object, int a1, double elem) {
  skepu::Vector<double> *vec = (skepu::Vector<double>*) object;
  (*vec)[a1] = elem;
}

void displayDataVector(void* object,char* type1) {

  if (!strcmp(type1,"double"))
    std::cout << "Result: " << *((skepu::Vector<double>*)object) << "\
         n";
  if (!strcmp(type1,"int"))
    std::cout << "Result: " << *((skepu::Vector<int>*)object) << "\n";
  if (!strcmp(type1,QUOTE(TYPE_NAME1)))
    {
      skepu::Vector<TYPE_NAME1>* m;
      m = (skepu::Vector<TYPE_NAME1>*)object;

      for(int j=0; j<m->size(); j++)
    {
      TYPE_NAME1 p=(*m)[j];

      std::cout<<std::setw(15)<<p.id<<std::setw(15)<<p.x<<std::setw
           (15)<<p.y<<std::setw(15)<<p.z<<std::setw(15)<<p.ax<<std::
           setw(15)<<p.ay<<std::setw(15)<<p.az<<std::setw(15)<<p.vx<<
           std::setw(15)<<p.vy<<std::setw(15)<<p.vz<<"\n";
    }
    }
  if (!strcmp(type1,QUOTE(TYPE_NAME2)))
```

```
    {
      skepu::Vector<TYPE_NAME2>* m;
      m = (skepu::Vector<TYPE_NAME2>*)object;

      for(int j=0; j<m->size(); j++)
    {
      TYPE_NAME2 p=(*m)[j];

      std::cout<<std::fixed<<std::setprecision(6)<<p.x<<"\t"
           <<std::fixed<<std::setprecision(6)<<p.y<<"\t"
           <<std::fixed<<std::setprecision(6)<<p.z<<"\t";
      std::cout << "\n";
    }
      std::cout << "\n";

    }
}

void displayDataMatrix(void* object,char* type1) {
  if (!strcmp(type1,"double"))
    std::cout << "Result: " << *((skepu::Matrix<double>*)object) << "\
        n";
  if (!strcmp(type1,"int"))
    std::cout << "Result: " << *((skepu::Matrix<int>*)object) << "\n";
  if (!strcmp(type1,QUOTE(TYPE_NAME1)));
  //std::cout << "Result: " << *((skepu::Vector<TYPE_NAME1>*)object)
       << "\n";
  if (!strcmp(type1,QUOTE(TYPE_NAME2)));
  //std::cout << "Result: " << *((skepu::Vector<TYPE_NAME2>*)object)
       << "\n";
}


double getRandNum()
{
  return rand();
}

void vectorUpdatehost(void *object)
{
  skepu::Vector<TYPE_NAME2> *vec = (skepu::Vector<TYPE_NAME2>*) object
      ;
  vec->updateHost();
}

int getFuncNum(char* funcName)
{
  if (!strcmp(funcName,QUOTE(FUNC_NAME_GENERATE1))) return 1;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_GENERATE2))) return 2;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_GENERATE3))) return 3;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_GENERATE4))) return 4;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_GENERATE5))) return 5;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_GENERATE6))) return 6;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_GENERATE7))) return 7;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_GENERATE8))) return 8;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_GENERATE9))) return 9;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_GENERATE10))) return 10;

  if (!strcmp(funcName,QUOTE(FUNC_NAME_OVERLAP1))) return 11;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_OVERLAP2))) return 12;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_OVERLAP3))) return 13;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_OVERLAP4))) return 14;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_OVERLAP5))) return 15;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_OVERLAP6))) return 16;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_OVERLAP7))) return 17;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_OVERLAP8))) return 18;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_OVERLAP9))) return 19;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_OVERLAP10))) return 20;
```

```
  if (!strcmp(funcName,QUOTE(FUNC_NAME_ARRAY1))) return 21;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_ARRAY2))) return 22;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_ARRAY3))) return 23;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_ARRAY4))) return 24;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_ARRAY5))) return 25;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_ARRAY6))) return 26;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_ARRAY7))) return 27;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_ARRAY8))) return 28;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_ARRAY9))) return 29;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_ARRAY10))) return 30;

  if (!strcmp(funcName,QUOTE(FUNC_NAME_BINARY1))) return 31;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_BINARY2))) return 32;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_BINARY3))) return 33;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_BINARY4))) return 34;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_BINARY5))) return 35;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_BINARY6))) return 36;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_BINARY7))) return 37;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_BINARY8))) return 38;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_BINARY9))) return 39;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_BINARY10))) return 40;

  if (!strcmp(funcName,QUOTE(FUNC_NAME_UNARY1))) return 41;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_UNARY2))) return 42;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_UNARY3))) return 43;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_UNARY4))) return 44;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_UNARY5))) return 45;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_UNARY6))) return 46;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_UNARY7))) return 47;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_UNARY8))) return 48;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_UNARY9))) return 49;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_UNARY10))) return 50;

  if (!strcmp(funcName,QUOTE(FUNC_NAME_BINARY_CONST1))) return 51;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_BINARY_CONST2))) return 52;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_BINARY_CONST3))) return 53;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_BINARY_CONST4))) return 54;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_BINARY_CONST5))) return 55;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_BINARY_CONST6))) return 56;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_BINARY_CONST7))) return 57;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_BINARY_CONST8))) return 58;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_BINARY_CONST9))) return 59;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_BINARY_CONST10))) return 60;

  if (!strcmp(funcName,QUOTE(FUNC_NAME_UNARY_CONST1))) return 61;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_UNARY_CONST2))) return 62;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_UNARY_CONST3))) return 63;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_UNARY_CONST4))) return 64;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_UNARY_CONST5))) return 65;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_UNARY_CONST6))) return 66;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_UNARY_CONST7))) return 67;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_UNARY_CONST8))) return 68;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_UNARY_CONST9))) return 69;
  if (!strcmp(funcName,QUOTE(FUNC_NAME_UNARY_CONST10))) return 70;

  return 1;
}


// SKELETON FUNTIONS
void* initMyReduce(char* funcName) {
  myStruct *m = new myStruct();
  int funcNum = getFuncNum(funcName);

  m->funcNum1 = funcNum;

  switch (funcNum)
```

```
    {
    case 31: m->skel = new skepu::Reduce<FUNC_NAME_BINARY1>(new
        FUNC_NAME_BINARY1); break;
    case 32: m->skel = new skepu::Reduce<FUNC_NAME_BINARY2>(new
        FUNC_NAME_BINARY2); break;
    case 33: m->skel = new skepu::Reduce<FUNC_NAME_BINARY3>(new
        FUNC_NAME_BINARY3); break;
    case 34: m->skel = new skepu::Reduce<FUNC_NAME_BINARY4>(new
        FUNC_NAME_BINARY4); break;
    case 35: m->skel = new skepu::Reduce<FUNC_NAME_BINARY5>(new
        FUNC_NAME_BINARY5); break;
    case 36: m->skel = new skepu::Reduce<FUNC_NAME_BINARY6>(new
        FUNC_NAME_BINARY6); break;
    case 37: m->skel = new skepu::Reduce<FUNC_NAME_BINARY7>(new
        FUNC_NAME_BINARY7); break;
    case 38: m->skel = new skepu::Reduce<FUNC_NAME_BINARY8>(new
        FUNC_NAME_BINARY8); break;
    case 39: m->skel = new skepu::Reduce<FUNC_NAME_BINARY9>(new
        FUNC_NAME_BINARY9); break;
    case 40: m->skel = new skepu::Reduce<FUNC_NAME_BINARY10>(new
        FUNC_NAME_BINARY10); break;

    case 41: m->skel = new skepu::Reduce<FUNC_NAME_UNARY1>(new
        FUNC_NAME_UNARY1); break;
    case 42: m->skel = new skepu::Reduce<FUNC_NAME_UNARY2>(new
        FUNC_NAME_UNARY2); break;
    case 43: m->skel = new skepu::Reduce<FUNC_NAME_UNARY3>(new
        FUNC_NAME_UNARY3); break;
    case 44: m->skel = new skepu::Reduce<FUNC_NAME_UNARY4>(new
        FUNC_NAME_UNARY4); break;
    case 45: m->skel = new skepu::Reduce<FUNC_NAME_UNARY5>(new
        FUNC_NAME_UNARY5); break;
    case 46: m->skel = new skepu::Reduce<FUNC_NAME_UNARY6>(new
        FUNC_NAME_UNARY6); break;
    case 47: m->skel = new skepu::Reduce<FUNC_NAME_UNARY7>(new
        FUNC_NAME_UNARY7); break;
    case 48: m->skel = new skepu::Reduce<FUNC_NAME_UNARY8>(new
        FUNC_NAME_UNARY8); break;
    case 49: m->skel = new skepu::Reduce<FUNC_NAME_UNARY9>(new
        FUNC_NAME_UNARY9); break;
    case 50: m->skel = new skepu::Reduce<FUNC_NAME_UNARY10>(new
        FUNC_NAME_UNARY10); break;

    case 51: m->skel = new skepu::Reduce<FUNC_NAME_BINARY_CONST1>(new
        FUNC_NAME_BINARY_CONST1); break;
    case 52: m->skel = new skepu::Reduce<FUNC_NAME_BINARY_CONST2>(new
        FUNC_NAME_BINARY_CONST2); break;
    case 53: m->skel = new skepu::Reduce<FUNC_NAME_BINARY_CONST3>(new
        FUNC_NAME_BINARY_CONST3); break;
    case 54: m->skel = new skepu::Reduce<FUNC_NAME_BINARY_CONST4>(new
        FUNC_NAME_BINARY_CONST4); break;
    case 55: m->skel = new skepu::Reduce<FUNC_NAME_BINARY_CONST5>(new
        FUNC_NAME_BINARY_CONST5); break;
    case 56: m->skel = new skepu::Reduce<FUNC_NAME_BINARY_CONST6>(new
        FUNC_NAME_BINARY_CONST6); break;
    case 57: m->skel = new skepu::Reduce<FUNC_NAME_BINARY_CONST7>(new
        FUNC_NAME_BINARY_CONST7); break;
    case 58: m->skel = new skepu::Reduce<FUNC_NAME_BINARY_CONST8>(new
        FUNC_NAME_BINARY_CONST8); break;
    case 59: m->skel = new skepu::Reduce<FUNC_NAME_BINARY_CONST9>(new
        FUNC_NAME_BINARY_CONST9); break;
    case 60: m->skel = new skepu::Reduce<FUNC_NAME_BINARY_CONST10>(new
         FUNC_NAME_BINARY_CONST10); break;

    case 61: m->skel = new skepu::Reduce<FUNC_NAME_UNARY_CONST1>(new
        FUNC_NAME_UNARY_CONST1); break;
    case 62: m->skel = new skepu::Reduce<FUNC_NAME_UNARY_CONST2>(new
        FUNC_NAME_UNARY_CONST2); break;
```

```
    case 63: m->skel = new skepu::Reduce<FUNC_NAME_UNARY_CONST3>(new
        FUNC_NAME_UNARY_CONST3); break;
    case 64: m->skel = new skepu::Reduce<FUNC_NAME_UNARY_CONST4>(new
        FUNC_NAME_UNARY_CONST4); break;
    case 65: m->skel = new skepu::Reduce<FUNC_NAME_UNARY_CONST5>(new
        FUNC_NAME_UNARY_CONST5); break;
    case 66: m->skel = new skepu::Reduce<FUNC_NAME_UNARY_CONST6>(new
        FUNC_NAME_UNARY_CONST6); break;
    case 67: m->skel = new skepu::Reduce<FUNC_NAME_UNARY_CONST7>(new
        FUNC_NAME_UNARY_CONST7); break;
    case 68: m->skel = new skepu::Reduce<FUNC_NAME_UNARY_CONST8>(new
        FUNC_NAME_UNARY_CONST8); break;
    case 69: m->skel = new skepu::Reduce<FUNC_NAME_UNARY_CONST9>(new
        FUNC_NAME_UNARY_CONST9); break;
    case 70: m->skel = new skepu::Reduce<FUNC_NAME_UNARY_CONST10>(new
        FUNC_NAME_UNARY_CONST10); break;

    default: m->skel = NULL; break;
    }

  if ( m->skel == NULL ) std::cout << "Not enough memory or wrong type
      " << std::endl;
  // read table from file and store all data in *table
  return (void*)m;
}

void closeMyReduce(void* object) {
  myStruct* t = (myStruct*)object;
  free(t->skel);
  delete t;
}

void* initMyMapReduce(char* funcName1, char* funcName2) {
  myStruct *m = new myStruct();
  int funcNum1 = getFuncNum(funcName1);
  int funcNum2 = getFuncNum(funcName2);

  m->funcNum1 = funcNum1;
  m->funcNum2 = funcNum2;

  switch (funcNum1)
    {
    case 31:
      {
    switch (funcNum2) {
    case 32: m->skel=new skepu::MapReduce<FUNC_NAME_BINARY1,
        FUNC_NAME_BINARY2>(new FUNC_NAME_BINARY1, new
        FUNC_NAME_BINARY2);break;
    default: m->skel = NULL; break;
    }
    break;}

    case 34:
      {
    switch (funcNum2) {
    case 35: m->skel=new skepu::MapReduce<FUNC_NAME_BINARY4,
        FUNC_NAME_BINARY5>(new FUNC_NAME_BINARY4, new
        FUNC_NAME_BINARY5); break;
    default: m->skel = NULL; break;
    }
    break;}

    case 36:
      {
    switch (funcNum2) {
    case 40:m->skel=new skepu::MapReduce<FUNC_NAME_BINARY6,
        FUNC_NAME_BINARY10>(new FUNC_NAME_BINARY6,new
        FUNC_NAME_BINARY10);break;
```

```
    default: m->skel = NULL; break;
    }
    break;}

    case 41:
      {switch (funcNum2) {
    case 32: m->skel = new skepu::MapReduce<FUNC_NAME_UNARY1,
        FUNC_NAME_BINARY2>(new FUNC_NAME_UNARY1, new
        FUNC_NAME_BINARY2); break;
    default: m->skel = NULL; break;
    }
    break;}

    case 61:
      {switch (funcNum2) {
    case 32: m->skel = new skepu::MapReduce<FUNC_NAME_UNARY_CONST1,
        FUNC_NAME_BINARY2>(new FUNC_NAME_UNARY_CONST1, new
        FUNC_NAME_BINARY2); break;
    default: m->skel = NULL; break;
    }
    break;}
    }

  if ( m->skel == NULL ) std::cout << "Not enough memory or wrong type
      " << std::endl;
  // read table from file and store all data in *table
  return (void*) m;
}

void closeMyMapReduce(void* object) {
  myStruct* t = (myStruct*)object;
  free(t->skel);
  delete t;
}

void* initMyScan(char* funcName) {
  myStruct *m = new myStruct();
  int funcNum = getFuncNum(funcName);
  m->funcNum1 = funcNum;

   switch (funcNum)
    {
    case 31: m->skel = new skepu::Scan<FUNC_NAME_BINARY1>(new
        FUNC_NAME_BINARY1); break;
    case 32: m->skel = new skepu::Scan<FUNC_NAME_BINARY2>(new
        FUNC_NAME_BINARY2); break;
    case 33: m->skel = new skepu::Scan<FUNC_NAME_BINARY3>(new
        FUNC_NAME_BINARY3); break;
    case 34: m->skel = new skepu::Scan<FUNC_NAME_BINARY4>(new
        FUNC_NAME_BINARY4); break;
    case 35: m->skel = new skepu::Scan<FUNC_NAME_BINARY5>(new
        FUNC_NAME_BINARY5); break;
    case 36: m->skel = new skepu::Scan<FUNC_NAME_BINARY6>(new
        FUNC_NAME_BINARY6); break;
    case 37: m->skel = new skepu::Scan<FUNC_NAME_BINARY7>(new
        FUNC_NAME_BINARY7); break;
    case 38: m->skel = new skepu::Scan<FUNC_NAME_BINARY8>(new
        FUNC_NAME_BINARY8); break;
    case 39: m->skel = new skepu::Scan<FUNC_NAME_BINARY9>(new
        FUNC_NAME_BINARY9); break;
    case 40: m->skel = new skepu::Scan<FUNC_NAME_BINARY10>(new
        FUNC_NAME_BINARY10); break;

    case 41: m->skel = new skepu::Scan<FUNC_NAME_UNARY1>(new
        FUNC_NAME_UNARY1); break;
    case 42: m->skel = new skepu::Scan<FUNC_NAME_UNARY2>(new
        FUNC_NAME_UNARY2); break;
    case 43: m->skel = new skepu::Scan<FUNC_NAME_UNARY3>(new
```

```
        FUNC_NAME_UNARY3); break;
    case 44: m->skel = new skepu::Scan<FUNC_NAME_UNARY4>(new
        FUNC_NAME_UNARY4); break;
    case 45: m->skel = new skepu::Scan<FUNC_NAME_UNARY5>(new
        FUNC_NAME_UNARY5); break;
    case 46: m->skel = new skepu::Scan<FUNC_NAME_UNARY6>(new
        FUNC_NAME_UNARY6); break;
    case 47: m->skel = new skepu::Scan<FUNC_NAME_UNARY7>(new
        FUNC_NAME_UNARY7); break;
    case 48: m->skel = new skepu::Scan<FUNC_NAME_UNARY8>(new
        FUNC_NAME_UNARY8); break;
    case 49: m->skel = new skepu::Scan<FUNC_NAME_UNARY9>(new
        FUNC_NAME_UNARY9); break;
    case 50: m->skel = new skepu::Scan<FUNC_NAME_UNARY10>(new
        FUNC_NAME_UNARY10); break;

    case 51: m->skel = new skepu::Scan<FUNC_NAME_BINARY_CONST1>(new
        FUNC_NAME_BINARY_CONST1); break;
    case 52: m->skel = new skepu::Scan<FUNC_NAME_BINARY_CONST2>(new
        FUNC_NAME_BINARY_CONST2); break;
    case 53: m->skel = new skepu::Scan<FUNC_NAME_BINARY_CONST3>(new
        FUNC_NAME_BINARY_CONST3); break;
    case 54: m->skel = new skepu::Scan<FUNC_NAME_BINARY_CONST4>(new
        FUNC_NAME_BINARY_CONST4); break;
    case 55: m->skel = new skepu::Scan<FUNC_NAME_BINARY_CONST5>(new
        FUNC_NAME_BINARY_CONST5); break;
    case 56: m->skel = new skepu::Scan<FUNC_NAME_BINARY_CONST6>(new
        FUNC_NAME_BINARY_CONST6); break;
    case 57: m->skel = new skepu::Scan<FUNC_NAME_BINARY_CONST7>(new
        FUNC_NAME_BINARY_CONST7); break;
    case 58: m->skel = new skepu::Scan<FUNC_NAME_BINARY_CONST8>(new
        FUNC_NAME_BINARY_CONST8); break;
    case 59: m->skel = new skepu::Scan<FUNC_NAME_BINARY_CONST9>(new
        FUNC_NAME_BINARY_CONST9); break;
    case 60: m->skel = new skepu::Scan<FUNC_NAME_BINARY_CONST10>(new
        FUNC_NAME_BINARY_CONST10); break;

    case 61: m->skel = new skepu::Scan<FUNC_NAME_UNARY_CONST1>(new
        FUNC_NAME_UNARY_CONST1); break;
    case 62: m->skel = new skepu::Scan<FUNC_NAME_UNARY_CONST2>(new
        FUNC_NAME_UNARY_CONST2); break;
    case 63: m->skel = new skepu::Scan<FUNC_NAME_UNARY_CONST3>(new
        FUNC_NAME_UNARY_CONST3); break;
    case 64: m->skel = new skepu::Scan<FUNC_NAME_UNARY_CONST4>(new
        FUNC_NAME_UNARY_CONST4); break;
    case 65: m->skel = new skepu::Scan<FUNC_NAME_UNARY_CONST5>(new
        FUNC_NAME_UNARY_CONST5); break;
    case 66: m->skel = new skepu::Scan<FUNC_NAME_UNARY_CONST6>(new
        FUNC_NAME_UNARY_CONST6); break;
    case 67: m->skel = new skepu::Scan<FUNC_NAME_UNARY_CONST7>(new
        FUNC_NAME_UNARY_CONST7); break;
    case 68: m->skel = new skepu::Scan<FUNC_NAME_UNARY_CONST8>(new
        FUNC_NAME_UNARY_CONST8); break;
    case 69: m->skel = new skepu::Scan<FUNC_NAME_UNARY_CONST9>(new
        FUNC_NAME_UNARY_CONST9); break;
    case 70: m->skel = new skepu::Scan<FUNC_NAME_UNARY_CONST10>(new
        FUNC_NAME_UNARY_CONST10); break;

    default: m->skel = NULL; break;
    }

  if ( m->skel == NULL ) std::cout << "Not enough memory or wrong type
      " << std::endl;
  // read table from file and store all data in *table
  return (void*)m;
}

void closeMyScan(void* object) {
```

```
 myStruct* t = (myStruct*)object;
 free(t->skel);
  delete t;
}

void* initMyMap(char* funcName) {
  myStruct *m = new myStruct();
  int funcNum = getFuncNum(funcName);
  m->funcNum1 = funcNum;

  switch (funcNum)
    {
    case 31: m->skel = new skepu::Map<FUNC_NAME_BINARY1>(new
         FUNC_NAME_BINARY1); break;
    case 32: m->skel = new skepu::Map<FUNC_NAME_BINARY2>(new
         FUNC_NAME_BINARY2); break;
    case 33: m->skel = new skepu::Map<FUNC_NAME_BINARY3>(new
         FUNC_NAME_BINARY3); break;
    case 34: m->skel = new skepu::Map<FUNC_NAME_BINARY4>(new
         FUNC_NAME_BINARY4); break;
    case 35: m->skel = new skepu::Map<FUNC_NAME_BINARY5>(new
         FUNC_NAME_BINARY5); break;
    case 36: m->skel = new skepu::Map<FUNC_NAME_BINARY6>(new
         FUNC_NAME_BINARY6); break;
    case 37: m->skel = new skepu::Map<FUNC_NAME_BINARY7>(new
         FUNC_NAME_BINARY7); break;
    case 38: m->skel = new skepu::Map<FUNC_NAME_BINARY8>(new
         FUNC_NAME_BINARY8); break;
    case 39: m->skel = new skepu::Map<FUNC_NAME_BINARY9>(new
         FUNC_NAME_BINARY9); break;
    case 40: m->skel = new skepu::Map<FUNC_NAME_BINARY10>(new
         FUNC_NAME_BINARY10); break;

    case 41: m->skel = new skepu::Map<FUNC_NAME_UNARY1>(new
         FUNC_NAME_UNARY1); break;
    case 42: m->skel = new skepu::Map<FUNC_NAME_UNARY2>(new
         FUNC_NAME_UNARY2); break;
    case 43: m->skel = new skepu::Map<FUNC_NAME_UNARY3>(new
         FUNC_NAME_UNARY3); break;
    case 44: m->skel = new skepu::Map<FUNC_NAME_UNARY4>(new
         FUNC_NAME_UNARY4); break;
    case 45: m->skel = new skepu::Map<FUNC_NAME_UNARY5>(new
         FUNC_NAME_UNARY5); break;
    case 46: m->skel = new skepu::Map<FUNC_NAME_UNARY6>(new
         FUNC_NAME_UNARY6); break;
    case 47: m->skel = new skepu::Map<FUNC_NAME_UNARY7>(new
         FUNC_NAME_UNARY7); break;
    case 48: m->skel = new skepu::Map<FUNC_NAME_UNARY8>(new
         FUNC_NAME_UNARY8); break;
    case 49: m->skel = new skepu::Map<FUNC_NAME_UNARY9>(new
         FUNC_NAME_UNARY9); break;
    case 50: m->skel = new skepu::Map<FUNC_NAME_UNARY10>(new
         FUNC_NAME_UNARY10); break;

    case 51: m->skel = new skepu::Map<FUNC_NAME_BINARY_CONST1>(new
         FUNC_NAME_BINARY_CONST1); break;
    case 52: m->skel = new skepu::Map<FUNC_NAME_BINARY_CONST2>(new
         FUNC_NAME_BINARY_CONST2); break;
    case 53: m->skel = new skepu::Map<FUNC_NAME_BINARY_CONST3>(new
         FUNC_NAME_BINARY_CONST3); break;
    case 54: m->skel = new skepu::Map<FUNC_NAME_BINARY_CONST4>(new
         FUNC_NAME_BINARY_CONST4); break;
    case 55: m->skel = new skepu::Map<FUNC_NAME_BINARY_CONST5>(new
         FUNC_NAME_BINARY_CONST5); break;
    case 56: m->skel = new skepu::Map<FUNC_NAME_BINARY_CONST6>(new
         FUNC_NAME_BINARY_CONST6); break;
    case 57: m->skel = new skepu::Map<FUNC_NAME_BINARY_CONST7>(new
         FUNC_NAME_BINARY_CONST7); break;
```

```
    case 58: m->skel = new skepu::Map<FUNC_NAME_BINARY_CONST8>(new
        FUNC_NAME_BINARY_CONST8); break;
    case 59: m->skel = new skepu::Map<FUNC_NAME_BINARY_CONST9>(new
        FUNC_NAME_BINARY_CONST9); break;
    case 60: m->skel = new skepu::Map<FUNC_NAME_BINARY_CONST10>(new
        FUNC_NAME_BINARY_CONST10); break;

    case 61: m->skel = new skepu::Map<FUNC_NAME_UNARY_CONST1>(new
        FUNC_NAME_UNARY_CONST1); break;
    case 62: m->skel = new skepu::Map<FUNC_NAME_UNARY_CONST2>(new
        FUNC_NAME_UNARY_CONST2); break;
    case 63: m->skel = new skepu::Map<FUNC_NAME_UNARY_CONST3>(new
        FUNC_NAME_UNARY_CONST3); break;
    case 64: m->skel = new skepu::Map<FUNC_NAME_UNARY_CONST4>(new
        FUNC_NAME_UNARY_CONST4); break;
    case 65: m->skel = new skepu::Map<FUNC_NAME_UNARY_CONST5>(new
        FUNC_NAME_UNARY_CONST5); break;
    case 66: m->skel = new skepu::Map<FUNC_NAME_UNARY_CONST6>(new
        FUNC_NAME_UNARY_CONST6); break;
    case 67: m->skel = new skepu::Map<FUNC_NAME_UNARY_CONST7>(new
        FUNC_NAME_UNARY_CONST7); break;
    case 68: m->skel = new skepu::Map<FUNC_NAME_UNARY_CONST8>(new
        FUNC_NAME_UNARY_CONST8); break;
    case 69: m->skel = new skepu::Map<FUNC_NAME_UNARY_CONST9>(new
        FUNC_NAME_UNARY_CONST9); break;
    case 70: m->skel = new skepu::Map<FUNC_NAME_UNARY_CONST10>(new
        FUNC_NAME_UNARY_CONST10); break;

    default: m->skel = NULL; break;
    }


  if ( m->skel == NULL ) std::cout << "Not enough memory or wrong type
      " << std::endl;
  // read table from file and store all data in *table
  return (void*)m;
}

void closeMyMap(void* object) {
  myStruct* t = (myStruct*)object;
  free(t->skel);
  delete t;
}

void* initMyMapArray(char* funcName) {
  myStruct *m = new myStruct();
  int funcNum = getFuncNum(funcName);
  m->funcNum1 = funcNum;

  switch (funcNum)
    {
    case 21: m->skel = new skepu::MapArray<FUNC_NAME_ARRAY1>(new
        FUNC_NAME_ARRAY1); break;
    case 22: m->skel = new skepu::MapArray<FUNC_NAME_ARRAY2>(new
        FUNC_NAME_ARRAY2); break;
    case 23: m->skel = new skepu::MapArray<FUNC_NAME_ARRAY3>(new
        FUNC_NAME_ARRAY3); break;
    case 24: m->skel = new skepu::MapArray<FUNC_NAME_ARRAY4>(new
        FUNC_NAME_ARRAY4); break;
    case 25: m->skel = new skepu::MapArray<FUNC_NAME_ARRAY5>(new
        FUNC_NAME_ARRAY5); break;
    case 26: m->skel = new skepu::MapArray<FUNC_NAME_ARRAY6>(new
        FUNC_NAME_ARRAY6); break;
    case 27: m->skel = new skepu::MapArray<FUNC_NAME_ARRAY7>(new
        FUNC_NAME_ARRAY7); break;
    case 28: m->skel = new skepu::MapArray<FUNC_NAME_ARRAY8>(new
        FUNC_NAME_ARRAY8); break;
    case 29: m->skel = new skepu::MapArray<FUNC_NAME_ARRAY9>(new
```

```
        FUNC_NAME_ARRAY9); break;
    case 30: m->skel = new skepu::MapArray<FUNC_NAME_ARRAY10>(new
        FUNC_NAME_ARRAY10); break;
    default: m->skel = NULL; break;
    }

 if ( m->skel == NULL ) std::cout << "Not enough memory or wrong type"
      << std::endl;
  // read table from file and store all data in *table
 return (void*)m;
}

void closeMyMapArray(void* object) {
 myStruct* t = (myStruct*)object;
 free(t->skel);
  delete t;
}

void* initMyMapOverlap(char* funcName) {
  myStruct *m = new myStruct();
  int funcNum = getFuncNum(funcName);
  m->funcNum1 = funcNum;

  switch (funcNum)
    {
    case 11: m->skel = new skepu::MapArray<FUNC_NAME_OVERLAP1>(new
        FUNC_NAME_OVERLAP1); break;
    case 12: m->skel = new skepu::MapArray<FUNC_NAME_OVERLAP2>(new
        FUNC_NAME_OVERLAP2); break;
    case 13: m->skel = new skepu::MapArray<FUNC_NAME_OVERLAP3>(new
        FUNC_NAME_OVERLAP3); break;
    case 14: m->skel = new skepu::MapArray<FUNC_NAME_OVERLAP4>(new
        FUNC_NAME_OVERLAP4); break;
    case 15: m->skel = new skepu::MapArray<FUNC_NAME_OVERLAP5>(new
        FUNC_NAME_OVERLAP5); break;
    case 16: m->skel = new skepu::MapArray<FUNC_NAME_OVERLAP6>(new
        FUNC_NAME_OVERLAP6); break;
    case 17: m->skel = new skepu::MapArray<FUNC_NAME_OVERLAP7>(new
        FUNC_NAME_OVERLAP7); break;
    case 18: m->skel = new skepu::MapArray<FUNC_NAME_OVERLAP8>(new
        FUNC_NAME_OVERLAP8); break;
    case 19: m->skel = new skepu::MapArray<FUNC_NAME_OVERLAP9>(new
        FUNC_NAME_OVERLAP9); break;
    case 20: m->skel = new skepu::MapArray<FUNC_NAME_OVERLAP10>(new
        FUNC_NAME_OVERLAP10); break;
    default: m->skel = NULL; break;
    }

  if ( m->skel == NULL ) std::cout << "Not enough memory or wrong type
      " << std::endl;
  // read table from file and store all data in *table
  return (void*)m;
}

void closeMyMapOverlap(void* object) {
 myStruct* t = (myStruct*)object;
 free(t->skel);
  delete t;
}

void* initMyGenerate(char* funcName) {
  myStruct *m = new myStruct();
  int funcNum = getFuncNum(funcName);
  m->funcNum1 = funcNum;

  switch (funcNum)
    {
    case 1: m->skel = new skepu::Generate<FUNC_NAME_GENERATE1>(new
```

```
          FUNC_NAME_GENERATE1); break;
     case 2: m->skel = new skepu::Generate<FUNC_NAME_GENERATE2>(new
          FUNC_NAME_GENERATE2); break;
     case 3: m->skel = new skepu::Generate<FUNC_NAME_GENERATE3>(new
          FUNC_NAME_GENERATE3); break;
     case 4: m->skel = new skepu::Generate<FUNC_NAME_GENERATE4>(new
          FUNC_NAME_GENERATE4); break;
     case 5: m->skel = new skepu::Generate<FUNC_NAME_GENERATE5>(new
          FUNC_NAME_GENERATE5); break;
     case 6: m->skel = new skepu::Generate<FUNC_NAME_GENERATE6>(new
          FUNC_NAME_GENERATE6); break;
     case 7: m->skel = new skepu::Generate<FUNC_NAME_GENERATE7>(new
          FUNC_NAME_GENERATE7); break;
     case 8: m->skel = new skepu::Generate<FUNC_NAME_GENERATE8>(new
          FUNC_NAME_GENERATE8); break;
     case 9: m->skel = new skepu::Generate<FUNC_NAME_GENERATE9>(new
          FUNC_NAME_GENERATE9); break;
     case 10: m->skel = new skepu::Generate<FUNC_NAME_GENERATE10>(new
          FUNC_NAME_GENERATE10); break;

     default: m->skel = NULL; break;
     }

  if (m->skel == NULL ) std::cout << "Not enough memory or wrong
       type333" << std::endl;
  // read table from file and store all data in *table
  return (void*)m;
}

void closeMyGenerate(void* object) {
  myStruct* t = (myStruct*)object;
  free(t->skel);
  delete t;
}

void generateType1(int numElem,void* object1,void* object2)
{
  skepu::Vector<TYPE_NAME1>* vec1 = (skepu::Vector<TYPE_NAME1>*)
       object1;
  myStruct *m = (myStruct*)object2;
  int funcNum = m->funcNum1;
  void * m2 = m->skel;

  switch (funcNum)
    {
    case 3:
      (*((skepu::Generate<FUNC_NAME_GENERATE3>*)m2)).setConstant((int)
           0);
      (*((skepu::Generate<FUNC_NAME_GENERATE3>*)m2))(numElem,*vec1);
           break;
    default: break;
    }
}

void generateType2(int numElem,void* object1,void* object2)
{
  skepu::Vector<TYPE_NAME2>* vec1 = (skepu::Vector<TYPE_NAME2>*)
       object1;
  myStruct *m = (myStruct*)object2;
  int funcNum = m->funcNum1;
  void * m2 = m->skel;

  switch (funcNum)
    {
    case 4:
      (*((skepu::Generate<FUNC_NAME_GENERATE4>*)m2))(numElem,*vec1);
           break;
    default:
```

```
      break;
    }
}


void SkePU_Generate_builtin_externalVdouble(int numElem, void* object1
    , void* object2, char* type1)
{
  if (!strcmp(type1,QUOTE(TYPE_NAME1)))
    generateType1(numElem,object1,object2);
  else if (!strcmp(type1,QUOTE(TYPE_NAME2)))
    generateType2(numElem,object1,object2);
  else { //double

    skepu::Vector<double>* vec1 = (skepu::Vector<double>*) object1;
    myStruct *m = (myStruct*)object2;
    int funcNum = m->funcNum1;
    void * m2 = m->skel;

    switch (funcNum)
      {
      case 1: (*((skepu::Generate<FUNC_NAME_GENERATE1>*)m2))(numElem,*
          vec1); break;
      case 2: (*((skepu::Generate<FUNC_NAME_GENERATE2>*)m2))(numElem,*
          vec1); break;

      default: break;
      }
  }
}

void SkePU_Generate_builtin_setConstant(void* object1,int elem)
{
  myStruct *m = (myStruct*)object1;
  int funcNum = m->funcNum1;
  void * m2 = m->skel;
  (*((skepu::Generate<FUNC_NAME_GENERATE3>*)m2)).setConstant(elem);
}

void SkePU_Map_builtin_externalMdouble(void* object1, void* object2,
    void* object3)
{
  skepu::Matrix<double>* mat1 = (skepu::Matrix<double>*) object1;
  skepu::Matrix<double>* mat2 = (skepu::Matrix<double>*) object2;
  myStruct *m = (myStruct*)object3;
  int funcNum = m->funcNum1;
  void * m2 = m->skel;

switch (funcNum)
    {
    case 31: (*((skepu::Map<FUNC_NAME_BINARY1>*)m2))(*mat1,*mat2);
        break;
    case 32: (*((skepu::Map<FUNC_NAME_BINARY2>*)m2))(*mat1,*mat2);
        break;
    case 33: (*((skepu::Map<FUNC_NAME_BINARY3>*)m2))(*mat1,*mat2);
        break;
    case 34: (*((skepu::Map<FUNC_NAME_BINARY4>*)m2))(*mat1,*mat2);
        break;
    case 35: (*((skepu::Map<FUNC_NAME_BINARY5>*)m2))(*mat1,*mat2);
        break;
    case 36: (*((skepu::Map<FUNC_NAME_BINARY6>*)m2))(*mat1,*mat2);
        break;
    case 37: (*((skepu::Map<FUNC_NAME_BINARY7>*)m2))(*mat1,*mat2);
        break;
    case 38: (*((skepu::Map<FUNC_NAME_BINARY8>*)m2))(*mat1,*mat2);
        break;
    case 39: (*((skepu::Map<FUNC_NAME_BINARY9>*)m2))(*mat1,*mat2);
        break;
```

```
    case 40: (*((skepu::Map<FUNC_NAME_BINARY10>*)m2))(*mat1,*mat2);
        break;

    case 41: (*((skepu::Map<FUNC_NAME_UNARY1>*)m2))(*mat1,*mat2);
        break;
    case 42: (*((skepu::Map<FUNC_NAME_UNARY2>*)m2))(*mat1,*mat2);
        break;
    case 43: (*((skepu::Map<FUNC_NAME_UNARY3>*)m2))(*mat1,*mat2);
        break;
    case 44: (*((skepu::Map<FUNC_NAME_UNARY4>*)m2))(*mat1,*mat2);
        break;
    case 45: (*((skepu::Map<FUNC_NAME_UNARY5>*)m2))(*mat1,*mat2);
        break;
    case 46: (*((skepu::Map<FUNC_NAME_UNARY6>*)m2))(*mat1,*mat2);
        break;
    case 47: (*((skepu::Map<FUNC_NAME_UNARY7>*)m2))(*mat1,*mat2);
        break;
    case 48: (*((skepu::Map<FUNC_NAME_UNARY8>*)m2))(*mat1,*mat2);
        break;
      //case 49: (*((skepu::Map<FUNC_NAME_UNARY9>*)m2))(*mat1,*mat2);
          break;
      //case 50: (*((skepu::Map<FUNC_NAME_UNARY10>*)m2))(*mat1,*mat2);
          break;

    case 51: (*((skepu::Map<FUNC_NAME_BINARY_CONST1>*)m2))(*mat1,*mat2
        ); break;
    case 52: (*((skepu::Map<FUNC_NAME_BINARY_CONST2>*)m2))(*mat1,*mat2
        ); break;
    case 53: (*((skepu::Map<FUNC_NAME_BINARY_CONST3>*)m2))(*mat1,*mat2
        ); break;
    case 54: (*((skepu::Map<FUNC_NAME_BINARY_CONST4>*)m2))(*mat1,*mat2
        ); break;
    case 55: (*((skepu::Map<FUNC_NAME_BINARY_CONST5>*)m2))(*mat1,*mat2
        ); break;
    case 56: (*((skepu::Map<FUNC_NAME_BINARY_CONST6>*)m2))(*mat1,*mat2
        ); break;
    case 57: (*((skepu::Map<FUNC_NAME_BINARY_CONST7>*)m2))(*mat1,*mat2
        ); break;
    case 58: (*((skepu::Map<FUNC_NAME_BINARY_CONST8>*)m2))(*mat1,*mat2
        ); break;
    case 59: (*((skepu::Map<FUNC_NAME_BINARY_CONST9>*)m2))(*mat1,*mat2
        ); break;
    case 60: (*((skepu::Map<FUNC_NAME_BINARY_CONST10>*)m2))(*mat1,*
        mat2); break;

    case 61: (*((skepu::Map<FUNC_NAME_UNARY_CONST1>*)m2))(*mat1,*mat2)
        ; break;
    case 62: (*((skepu::Map<FUNC_NAME_UNARY_CONST2>*)m2))(*mat1,*mat2)
        ; break;
    case 63: (*((skepu::Map<FUNC_NAME_UNARY_CONST3>*)m2))(*mat1,*mat2)
        ; break;
    case 64: (*((skepu::Map<FUNC_NAME_UNARY_CONST4>*)m2))(*mat1,*mat2)
        ; break;
    case 65: (*((skepu::Map<FUNC_NAME_UNARY_CONST5>*)m2))(*mat1,*mat2)
        ; break;
    case 66: (*((skepu::Map<FUNC_NAME_UNARY_CONST6>*)m2))(*mat1,*mat2)
        ; break;
    case 67: (*((skepu::Map<FUNC_NAME_UNARY_CONST7>*)m2))(*mat1,*mat2)
        ; break;
    case 68: (*((skepu::Map<FUNC_NAME_UNARY_CONST8>*)m2))(*mat1,*mat2)
        ; break;
    case 69: (*((skepu::Map<FUNC_NAME_UNARY_CONST9>*)m2))(*mat1,*mat2)
        ; break;
    case 70: (*((skepu::Map<FUNC_NAME_UNARY_CONST10>*)m2))(*mat1,*mat2
        ); break;

    default: m->skel = NULL; break;
    }
```

```
}

void mapType2(void* object1, void* object2, void* object3)
{
  skepu::Vector<TYPE_NAME2>* vec1 = (skepu::Vector<TYPE_NAME2>*)
      object1;
  skepu::Vector<TYPE_NAME2>* vec2 = (skepu::Vector<TYPE_NAME2>*)
      object2;
  myStruct *m = (myStruct*)object3;
  int funcNum = m->funcNum1;
  void * m2 = m->skel;

  switch (funcNum)
    {
    case 49: (*((skepu::Map<FUNC_NAME_UNARY9>*)m2))(*vec1,*vec2);
         break;
    case 50: (*((skepu::Map<FUNC_NAME_UNARY10>*)m2))(*vec1,*vec2);
         break;
    default: break;
    }
}

void SkePU_Map_builtin_externalVdouble(void* object1, void* object2,
     void* object3,char* type1)
{
  if (!strcmp(type1,QUOTE(TYPE_NAME2)))
    mapType2(object1,object2,object3);
  else
    {
      skepu::Vector<double>* vec1 = (skepu::Vector<double>*) object1;
      skepu::Vector<double>* vec2 = (skepu::Vector<double>*) object2;
      myStruct *m = (myStruct*)object3;
      int funcNum = m->funcNum1;
      void * m2 = m->skel;

      switch (funcNum)
    {
    case 31: (*((skepu::Map<FUNC_NAME_BINARY1>*)m2))(*vec1,*vec2);
         break;
    case 32: (*((skepu::Map<FUNC_NAME_BINARY2>*)m2))(*vec1,*vec2);
         break;
    case 33: (*((skepu::Map<FUNC_NAME_BINARY3>*)m2))(*vec1,*vec2);
         break;
    case 34: (*((skepu::Map<FUNC_NAME_BINARY4>*)m2))(*vec1,*vec2);
         break;
    case 35: (*((skepu::Map<FUNC_NAME_BINARY5>*)m2))(*vec1,*vec2);
         break;
    case 36: (*((skepu::Map<FUNC_NAME_BINARY6>*)m2))(*vec1,*vec2);
         break;
    case 37: (*((skepu::Map<FUNC_NAME_BINARY7>*)m2))(*vec1,*vec2);
         break;
    case 38: (*((skepu::Map<FUNC_NAME_BINARY8>*)m2))(*vec1,*vec2);
         break;
    case 39: (*((skepu::Map<FUNC_NAME_BINARY9>*)m2))(*vec1,*vec2);
         break;
    case 40: (*((skepu::Map<FUNC_NAME_BINARY10>*)m2))(*vec1,*vec2);
         break;

    case 41: (*((skepu::Map<FUNC_NAME_UNARY1>*)m2))(*vec1,*vec2);
         break;
    case 42: (*((skepu::Map<FUNC_NAME_UNARY2>*)m2))(*vec1,*vec2);
         break;
    case 43: (*((skepu::Map<FUNC_NAME_UNARY3>*)m2))(*vec1,*vec2);
         break;
    case 44: (*((skepu::Map<FUNC_NAME_UNARY4>*)m2))(*vec1,*vec2);
         break;
    case 45: (*((skepu::Map<FUNC_NAME_UNARY5>*)m2))(*vec1,*vec2);
         break;
```

```
    case 46: (*((skepu::Map<FUNC_NAME_UNARY6>*)m2))(*vec1,*vec2);
        break;
    case 47: (*((skepu::Map<FUNC_NAME_UNARY7>*)m2))(*vec1,*vec2);
        break;
    case 48: (*((skepu::Map<FUNC_NAME_UNARY8>*)m2))(*vec1,*vec2);
        break;
      //case 49: (*((skepu::Map<FUNC_NAME_UNARY9>*)m2))(*vec1,*vec2);
          break;
      //case 50: (*((skepu::Map<FUNC_NAME_UNARY10>*)m2))(*vec1,*vec2);
          break;

    case 51: (*((skepu::Map<FUNC_NAME_BINARY_CONST1>*)m2))(*vec1,*vec2
        ); break;
    case 52: (*((skepu::Map<FUNC_NAME_BINARY_CONST2>*)m2))(*vec1,*vec2
        ); break;
    case 53: (*((skepu::Map<FUNC_NAME_BINARY_CONST3>*)m2))(*vec1,*vec2
        ); break;
    case 54: (*((skepu::Map<FUNC_NAME_BINARY_CONST4>*)m2))(*vec1,*vec2
        ); break;
    case 55: (*((skepu::Map<FUNC_NAME_BINARY_CONST5>*)m2))(*vec1,*vec2
        ); break;
    case 56: (*((skepu::Map<FUNC_NAME_BINARY_CONST6>*)m2))(*vec1,*vec2
        ); break;
    case 57: (*((skepu::Map<FUNC_NAME_BINARY_CONST7>*)m2))(*vec1,*vec2
        ); break;
    case 58: (*((skepu::Map<FUNC_NAME_BINARY_CONST8>*)m2))(*vec1,*vec2
        ); break;
    case 59: (*((skepu::Map<FUNC_NAME_BINARY_CONST9>*)m2))(*vec1,*vec2
        ); break;
    case 60: (*((skepu::Map<FUNC_NAME_BINARY_CONST10>*)m2))(*vec1,*
        vec2); break;

    case 61: (*((skepu::Map<FUNC_NAME_UNARY_CONST1>*)m2))(*vec1,*vec2)
        ; break;
    case 62: (*((skepu::Map<FUNC_NAME_UNARY_CONST2>*)m2))(*vec1,*vec2)
        ; break;
    case 63: (*((skepu::Map<FUNC_NAME_UNARY_CONST3>*)m2))(*vec1,*vec2)
        ; break;
    case 64: (*((skepu::Map<FUNC_NAME_UNARY_CONST4>*)m2))(*vec1,*vec2)
        ; break;
    case 65: (*((skepu::Map<FUNC_NAME_UNARY_CONST5>*)m2))(*vec1,*vec2)
        ; break;
    case 66: (*((skepu::Map<FUNC_NAME_UNARY_CONST6>*)m2))(*vec1,*vec2)
        ; break;
    case 67: (*((skepu::Map<FUNC_NAME_UNARY_CONST7>*)m2))(*vec1,*vec2)
        ; break;
    case 68: (*((skepu::Map<FUNC_NAME_UNARY_CONST8>*)m2))(*vec1,*vec2)
        ; break;
    case 69: (*((skepu::Map<FUNC_NAME_UNARY_CONST9>*)m2))(*vec1,*vec2)
        ; break;
    case 70: (*((skepu::Map<FUNC_NAME_UNARY_CONST10>*)m2))(*vec1,*vec2
        ); break;

    default: m->skel = NULL; break;
    }

    }
}

void SkePU_Map_builtin_externalV1double(void* object1, void* object2)
{
  //if (!strcmp(type1,QUOTE(TYPE_NAME1)))
  //  mapType1(object1,object2,object3);
  //else if (!strcmp(type1,QUOTE(TYPE_NAME2)))
  //  mapType2(object1,object2,object3);
  //else
  //{
      skepu::Vector<double>* vec1 = (skepu::Vector<double>*) object1;
```

```
    myStruct *m = (myStruct*)object2;
    int funcNum = m->funcNum1;
    void * m2 = m->skel;
    switch (funcNum)
  {
  case 31: (*((skepu::Map<FUNC_NAME_BINARY1>*)m2))(*vec1); break;
  case 32: (*((skepu::Map<FUNC_NAME_BINARY2>*)m2))(*vec1); break;
  case 33: (*((skepu::Map<FUNC_NAME_BINARY3>*)m2))(*vec1); break;
  case 34: (*((skepu::Map<FUNC_NAME_BINARY4>*)m2))(*vec1); break;
  case 35: (*((skepu::Map<FUNC_NAME_BINARY5>*)m2))(*vec1); break;
  case 36: (*((skepu::Map<FUNC_NAME_BINARY6>*)m2))(*vec1); break;
  case 37: (*((skepu::Map<FUNC_NAME_BINARY7>*)m2))(*vec1); break;
  case 38: (*((skepu::Map<FUNC_NAME_BINARY8>*)m2))(*vec1); break;
  case 39: (*((skepu::Map<FUNC_NAME_BINARY9>*)m2))(*vec1); break;
  case 40: (*((skepu::Map<FUNC_NAME_BINARY10>*)m2))(*vec1); break;

  case 41: (*((skepu::Map<FUNC_NAME_UNARY1>*)m2))(*vec1); break;
  case 42: (*((skepu::Map<FUNC_NAME_UNARY2>*)m2))(*vec1); break;
  case 43: (*((skepu::Map<FUNC_NAME_UNARY3>*)m2))(*vec1); break;
  case 44: (*((skepu::Map<FUNC_NAME_UNARY4>*)m2))(*vec1); break;
  case 45: (*((skepu::Map<FUNC_NAME_UNARY5>*)m2))(*vec1); break;
  case 46: (*((skepu::Map<FUNC_NAME_UNARY6>*)m2))(*vec1); break;
  case 47: (*((skepu::Map<FUNC_NAME_UNARY7>*)m2))(*vec1); break;
  case 48: (*((skepu::Map<FUNC_NAME_UNARY8>*)m2))(*vec1); break;
    //case 49: (*((skepu::Map<FUNC_NAME_UNARY9>*)m2))(*vec1); break;
    //case 50: (*((skepu::Map<FUNC_NAME_UNARY10>*)m2))(*vec1); break
        ;

  case 51: (*((skepu::Map<FUNC_NAME_BINARY_CONST1>*)m2))(*vec1);
      break;
  case 52: (*((skepu::Map<FUNC_NAME_BINARY_CONST2>*)m2))(*vec1);
      break;
  case 53: (*((skepu::Map<FUNC_NAME_BINARY_CONST3>*)m2))(*vec1);
      break;
  case 54: (*((skepu::Map<FUNC_NAME_BINARY_CONST4>*)m2))(*vec1);
      break;
  case 55: (*((skepu::Map<FUNC_NAME_BINARY_CONST5>*)m2))(*vec1);
      break;
  case 56: (*((skepu::Map<FUNC_NAME_BINARY_CONST6>*)m2))(*vec1);
      break;
  case 57: (*((skepu::Map<FUNC_NAME_BINARY_CONST7>*)m2))(*vec1);
      break;
  case 58: (*((skepu::Map<FUNC_NAME_BINARY_CONST8>*)m2))(*vec1);
      break;
  case 59: (*((skepu::Map<FUNC_NAME_BINARY_CONST9>*)m2))(*vec1);
      break;
  case 60: (*((skepu::Map<FUNC_NAME_BINARY_CONST10>*)m2))(*vec1);
      break;

  case 61: (*((skepu::Map<FUNC_NAME_UNARY_CONST1>*)m2))(*vec1);
      break;
  case 62: (*((skepu::Map<FUNC_NAME_UNARY_CONST2>*)m2))(*vec1);
      break;
  case 63: (*((skepu::Map<FUNC_NAME_UNARY_CONST3>*)m2))(*vec1);
      break;
  case 64: (*((skepu::Map<FUNC_NAME_UNARY_CONST4>*)m2))(*vec1);
      break;
  case 65: (*((skepu::Map<FUNC_NAME_UNARY_CONST5>*)m2))(*vec1);
      break;
  case 66: (*((skepu::Map<FUNC_NAME_UNARY_CONST6>*)m2))(*vec1);
      break;
  case 67: (*((skepu::Map<FUNC_NAME_UNARY_CONST7>*)m2))(*vec1);
      break;
  case 68: (*((skepu::Map<FUNC_NAME_UNARY_CONST8>*)m2))(*vec1);
      break;
  case 69: (*((skepu::Map<FUNC_NAME_UNARY_CONST9>*)m2))(*vec1);
      break;
  case 70: (*((skepu::Map<FUNC_NAME_UNARY_CONST10>*)m2))(*vec1);
```

```
        break;

    default: m->skel = NULL; break;
    }
}

void SkePU_Map_builtin_externalV2double(void* object1, void* object2,
     void* object3, void* object4, char* type1)
{
  skepu::Vector<double>* vec1 = (skepu::Vector<double>*) object1;
  skepu::Vector<double>* vec2 = (skepu::Vector<double>*) object2;
  skepu::Vector<double>* vec3 = (skepu::Vector<double>*) object3;
  myStruct *m = (myStruct*)object4;
  int funcNum = m->funcNum1;
  void * m2 = m->skel;

  switch (funcNum)
    {
    case 31: (*((skepu::Map<FUNC_NAME_BINARY1>*)m2))(*vec1,*vec2,*vec3
        ); break;
    case 32: (*((skepu::Map<FUNC_NAME_BINARY2>*)m2))(*vec1,*vec2,*vec3
        ); break;
    case 33: (*((skepu::Map<FUNC_NAME_BINARY3>*)m2))(*vec1,*vec2,*vec3
        ); break;
    case 34: (*((skepu::Map<FUNC_NAME_BINARY4>*)m2))(*vec1,*vec2,*vec3
        ); break;
    case 35: (*((skepu::Map<FUNC_NAME_BINARY5>*)m2))(*vec1,*vec2,*vec3
        ); break;
    case 36: (*((skepu::Map<FUNC_NAME_BINARY6>*)m2))(*vec1,*vec2,*vec3
        ); break;
    case 37: (*((skepu::Map<FUNC_NAME_BINARY7>*)m2))(*vec1,*vec2,*vec3
        ); break;
    case 38: (*((skepu::Map<FUNC_NAME_BINARY8>*)m2))(*vec1,*vec2,*vec3
        ); break;
    case 39: (*((skepu::Map<FUNC_NAME_BINARY9>*)m2))(*vec1,*vec2,*vec3
        ); break;
    case 40: (*((skepu::Map<FUNC_NAME_BINARY10>*)m2))(*vec1,*vec2,*
        vec3); break;

    case 41: (*((skepu::Map<FUNC_NAME_UNARY1>*)m2))(*vec1,*vec2,*vec3)
        ; break;
    case 42: (*((skepu::Map<FUNC_NAME_UNARY2>*)m2))(*vec1,*vec2,*vec3)
        ; break;
    case 43: (*((skepu::Map<FUNC_NAME_UNARY3>*)m2))(*vec1,*vec2,*vec3)
        ; break;
    case 44: (*((skepu::Map<FUNC_NAME_UNARY4>*)m2))(*vec1,*vec2,*vec3)
        ; break;
    case 45: (*((skepu::Map<FUNC_NAME_UNARY5>*)m2))(*vec1,*vec2,*vec3)
        ; break;
    case 46: (*((skepu::Map<FUNC_NAME_UNARY6>*)m2))(*vec1,*vec2,*vec3)
        ; break;
    case 47: (*((skepu::Map<FUNC_NAME_UNARY7>*)m2))(*vec1,*vec2,*vec3)
        ; break;
    case 48: (*((skepu::Map<FUNC_NAME_UNARY8>*)m2))(*vec1,*vec2,*vec3)
        ; break;
      //case 49: (*((skepu::Map<FUNC_NAME_UNARY9>*)m2))(*vec1,*vec2,*
          vec3); break;
      //case 50: (*((skepu::Map<FUNC_NAME_UNARY10>*)m2))(*vec1,*vec2,*
          vec3); break;

    case 51: (*((skepu::Map<FUNC_NAME_BINARY_CONST1>*)m2))(*vec1,*vec2
        ,*vec3); break;
    case 52: (*((skepu::Map<FUNC_NAME_BINARY_CONST2>*)m2))(*vec1,*vec2
        ,*vec3); break;
    case 53: (*((skepu::Map<FUNC_NAME_BINARY_CONST3>*)m2))(*vec1,*vec2
        ,*vec3); break;
    case 54: (*((skepu::Map<FUNC_NAME_BINARY_CONST4>*)m2))(*vec1,*vec2
        ,*vec3); break;
```

```
    case 55: (*((skepu::Map<FUNC_NAME_BINARY_CONST5>*)m2))(*vec1,*vec2
        ,*vec3); break;
    case 56: (*((skepu::Map<FUNC_NAME_BINARY_CONST6>*)m2))(*vec1,*vec2
        ,*vec3); break;
    case 57: (*((skepu::Map<FUNC_NAME_BINARY_CONST7>*)m2))(*vec1,*vec2
        ,*vec3); break;
    case 58: (*((skepu::Map<FUNC_NAME_BINARY_CONST8>*)m2))(*vec1,*vec2
        ,*vec3); break;
    case 59: (*((skepu::Map<FUNC_NAME_BINARY_CONST9>*)m2))(*vec1,*vec2
        ,*vec3); break;
    case 60: (*((skepu::Map<FUNC_NAME_BINARY_CONST10>*)m2))(*vec1,*
        vec2,*vec3); break;

    case 61: (*((skepu::Map<FUNC_NAME_UNARY_CONST1>*)m2))(*vec1,*vec2
        ,*vec3); break;
    case 62: (*((skepu::Map<FUNC_NAME_UNARY_CONST2>*)m2))(*vec1,*vec2
        ,*vec3); break;
    case 63: (*((skepu::Map<FUNC_NAME_UNARY_CONST3>*)m2))(*vec1,*vec2
        ,*vec3); break;
    case 64: (*((skepu::Map<FUNC_NAME_UNARY_CONST4>*)m2))(*vec1,*vec2
        ,*vec3); break;
    case 65: (*((skepu::Map<FUNC_NAME_UNARY_CONST5>*)m2))(*vec1,*vec2
        ,*vec3); break;
    case 66: (*((skepu::Map<FUNC_NAME_UNARY_CONST6>*)m2))(*vec1,*vec2
        ,*vec3); break;
    case 67: (*((skepu::Map<FUNC_NAME_UNARY_CONST7>*)m2))(*vec1,*vec2
        ,*vec3); break;
    case 68: (*((skepu::Map<FUNC_NAME_UNARY_CONST8>*)m2))(*vec1,*vec2
        ,*vec3); break;
    case 69: (*((skepu::Map<FUNC_NAME_UNARY_CONST9>*)m2))(*vec1,*vec2
        ,*vec3); break;
    case 70: (*((skepu::Map<FUNC_NAME_UNARY_CONST10>*)m2))(*vec1,*vec2
        ,*vec3); break;

    default: m->skel = NULL; break;
    }
}

void SkePU_Map_builtin_setConstant(void* object1, double val1)
{
 myStruct *m = (myStruct*)object1;
 int funcNum = m->funcNum1;
 void * m2 = m->skel;

 switch (funcNum)
    {
    case 51:
      ((skepu::Map<FUNC_NAME_BINARY_CONST1>*)m2)->setConstant(val1);
          break;
    case 52:
      ((skepu::Map<FUNC_NAME_BINARY_CONST2>*)m2)->setConstant(val1);
          break;
    case 53:
      ((skepu::Map<FUNC_NAME_BINARY_CONST3>*)m2)->setConstant(val1);
          break;
    case 54:
      ((skepu::Map<FUNC_NAME_BINARY_CONST4>*)m2)->setConstant(val1);
          break;
    case 55:
      ((skepu::Map<FUNC_NAME_BINARY_CONST5>*)m2)->setConstant(val1);
          break;
    case 56:
      ((skepu::Map<FUNC_NAME_BINARY_CONST6>*)m2)->setConstant(val1);
          break;
    case 57:
      ((skepu::Map<FUNC_NAME_BINARY_CONST7>*)m2)->setConstant(val1);
          break;
    case 58:
```

```
      ((skepu::Map<FUNC_NAME_BINARY_CONST8>*)m2)->setConstant(val1);
          break;
    case 59:
      ((skepu::Map<FUNC_NAME_BINARY_CONST9>*)m2)->setConstant(val1);
          break;
    case 60:
      ((skepu::Map<FUNC_NAME_BINARY_CONST10>*)m2)->setConstant(val1);
          break;

   case 61:
      ((skepu::Map<FUNC_NAME_UNARY_CONST1>*)m2)->setConstant(val1);
          break;
    case 62:
      ((skepu::Map<FUNC_NAME_UNARY_CONST2>*)m2)->setConstant(val1);
          break;
    case 63:
      ((skepu::Map<FUNC_NAME_UNARY_CONST3>*)m2)->setConstant(val1);
          break;
    case 64:
      ((skepu::Map<FUNC_NAME_UNARY_CONST4>*)m2)->setConstant(val1);
          break;
    case 65:
      ((skepu::Map<FUNC_NAME_UNARY_CONST5>*)m2)->setConstant(val1);
          break;
    case 66:
      ((skepu::Map<FUNC_NAME_UNARY_CONST6>*)m2)->setConstant(val1);
          break;
    case 67:
      ((skepu::Map<FUNC_NAME_UNARY_CONST7>*)m2)->setConstant(val1);
          break;
    case 68:
      ((skepu::Map<FUNC_NAME_UNARY_CONST8>*)m2)->setConstant(val1);
          break;
    case 69:
      ((skepu::Map<FUNC_NAME_UNARY_CONST9>*)m2)->setConstant(val1);
          break;
    case 70:
      ((skepu::Map<FUNC_NAME_UNARY_CONST10>*)m2)->setConstant(val1);
          break;
    }
}

void maparrayType1(void* object1, void* object2, void* object3, void*
    object4)
{
  skepu::Vector<TYPE_NAME1>* vec1 = (skepu::Vector<TYPE_NAME1>*)
      object1;
  skepu::Vector<TYPE_NAME1>* vec2 = (skepu::Vector<TYPE_NAME1>*)
      object2;
  skepu::Vector<TYPE_NAME1>* vec3 = (skepu::Vector<TYPE_NAME1>*)
      object3;
  myStruct *m = (myStruct*)object4;
  int funcNum = m->funcNum1;
  void * m2 = m->skel;

  switch (funcNum)
    {
    case 30:
      (*((skepu::MapArray<FUNC_NAME_ARRAY10>*)m2))(*vec1,*vec2,*vec3);
          break;
    }
}

void maparrayType2(void* object1, void* object2, void* object3, void*
    object4)
{
  skepu::Vector<TYPE_NAME2>* vec1 = (skepu::Vector<TYPE_NAME2>*)
      object1;
```

```
  skepu::Vector<TYPE_NAME2>* vec2 = (skepu::Vector<TYPE_NAME2>*)
      object2;
  skepu::Vector<TYPE_NAME2>* vec3 = (skepu::Vector<TYPE_NAME2>*)
      object3;
  myStruct *m = (myStruct*)object4;
  int funcNum = m->funcNum1;
  void * m2 = m->skel;

  switch (funcNum)
    {
    case 27:
      (*((skepu::MapArray<FUNC_NAME_ARRAY7>*)m2))(*vec1,*vec2,*vec3);
          break;
    case 28:
      (*((skepu::MapArray<FUNC_NAME_ARRAY8>*)m2))(*vec1,*vec2,*vec3);
          break;
    case 29:
      (*((skepu::MapArray<FUNC_NAME_ARRAY9>*)m2))(*vec1,*vec2,*vec3);
          break;
    }
}

void SkePU_MapArray_builtin_externalVdouble(void* object1, void*
    object2, void* object3, void* object4, char* type1)
{
  if (!strcmp(type1,QUOTE(TYPE_NAME1)))
    maparrayType1(object1,object2,object3,object4);
  else if (!strcmp(type1,QUOTE(TYPE_NAME2)))
    maparrayType2(object1,object2,object3,object4);
  else {
  skepu::Vector<double>* vec1 = (skepu::Vector<double>*) object1;
  skepu::Vector<double>* vec2 = (skepu::Vector<double>*) object2;
  skepu::Vector<double>* vec3 = (skepu::Vector<double>*) object3;
  myStruct *m = (myStruct*)object4;
  int funcNum = m->funcNum1;
  void * m2 = m->skel;

  switch (funcNum)
    {
    case 21:
      (*((skepu::MapArray<FUNC_NAME_ARRAY1>*)m2))(*vec1,*vec2,*vec3);
          break;
    case 22:
      (*((skepu::MapArray<FUNC_NAME_ARRAY2>*)m2))(*vec1,*vec2,*vec3);
          break;
    case 23:
      (*((skepu::MapArray<FUNC_NAME_ARRAY3>*)m2))(*vec1,*vec2,*vec3);
          break;
    case 24:
      (*((skepu::MapArray<FUNC_NAME_ARRAY4>*)m2))(*vec1,*vec2,*vec3);
          break;
    case 25:
      (*((skepu::MapArray<FUNC_NAME_ARRAY5>*)m2))(*vec1,*vec2,*vec3);
          break;
    case 26:
      (*((skepu::MapArray<FUNC_NAME_ARRAY6>*)m2))(*vec1,*vec2,*vec3);
          break;
    }
  }
}

void SkePU_MapOverlap_builtin_externalVdouble(void* object1, void*
    object2, double object3, void* object4)
{
  skepu::Vector<double>* vec1 = (skepu::Vector<double>*) object1;
  skepu::Vector<double>* vec2 = (skepu::Vector<double>*) object2;
  myStruct* m= (myStruct*)object4;
  int funcNum = m->funcNum1;
```

```
  void * m2 = m->skel;

  switch (funcNum)
    {
    case 11:
      (*((skepu::MapOverlap<FUNC_NAME_OVERLAP1>*)m2))(*vec1,*vec2,
          skepu::CONSTANT,object3); break;
    case 12:
      (*((skepu::MapOverlap<FUNC_NAME_OVERLAP2>*)m2))(*vec1,*vec2,
          skepu::CONSTANT,object3); break;
    case 13:
      (*((skepu::MapOverlap<FUNC_NAME_OVERLAP3>*)m2))(*vec1,*vec2,
          skepu::CONSTANT,object3); break;
    case 14:
      (*((skepu::MapOverlap<FUNC_NAME_OVERLAP4>*)m2))(*vec1,*vec2,
          skepu::CONSTANT,object3); break;
    case 15:
      (*((skepu::MapOverlap<FUNC_NAME_OVERLAP5>*)m2))(*vec1,*vec2,
          skepu::CONSTANT,object3); break;
    case 16:
      (*((skepu::MapOverlap<FUNC_NAME_OVERLAP6>*)m2))(*vec1,*vec2,
          skepu::CONSTANT,object3); break;
    case 17:
      (*((skepu::MapOverlap<FUNC_NAME_OVERLAP7>*)m2))(*vec1,*vec2,
          skepu::CONSTANT,object3); break;
    case 18:
      (*((skepu::MapOverlap<FUNC_NAME_OVERLAP8>*)m2))(*vec1,*vec2,
          skepu::CONSTANT,object3); break;
    case 19:
      (*((skepu::MapOverlap<FUNC_NAME_OVERLAP9>*)m2))(*vec1,*vec2,
          skepu::CONSTANT,object3); break;
    case 20:
      (*((skepu::MapOverlap<FUNC_NAME_OVERLAP10>*)m2))(*vec1,*vec2,
          skepu::CONSTANT,object3); break;
    default:
      (*((skepu::MapOverlap<FUNC_NAME_OVERLAP1>*)m2))(*vec1,*vec2,
          skepu::CONSTANT,object3); break;
    }
}

void SkePU_Scan_builtin_externalVdouble(void* object1, void* object2,
    void* object3)
{
  skepu::Vector<double>* vec1 = (skepu::Vector<double>*) object1;
  skepu::Vector<double>* vec2 = (skepu::Vector<double>*) object2;
  myStruct *m = (myStruct*)object3;
  int funcNum = m->funcNum1;
  void * m2 = m->skel;

  switch (funcNum)
    {
    case 31: (*((skepu::Scan<FUNC_NAME_BINARY1>*)m2))(*vec1,*vec2,
        skepu::INCLUSIVE); break;
    case 32: (*((skepu::Scan<FUNC_NAME_BINARY2>*)m2))(*vec1,*vec2,
        skepu::INCLUSIVE); break;
    case 33: (*((skepu::Scan<FUNC_NAME_BINARY3>*)m2))(*vec1,*vec2,
        skepu::INCLUSIVE); break;
    case 34: (*((skepu::Scan<FUNC_NAME_BINARY4>*)m2))(*vec1,*vec2,
        skepu::INCLUSIVE); break;
    case 35: (*((skepu::Scan<FUNC_NAME_BINARY5>*)m2))(*vec1,*vec2,
        skepu::INCLUSIVE); break;
    case 36: (*((skepu::Scan<FUNC_NAME_BINARY6>*)m2))(*vec1,*vec2,
        skepu::INCLUSIVE); break;
    case 37: (*((skepu::Scan<FUNC_NAME_BINARY7>*)m2))(*vec1,*vec2,
        skepu::INCLUSIVE); break;
    case 38: (*((skepu::Scan<FUNC_NAME_BINARY8>*)m2))(*vec1,*vec2,
        skepu::INCLUSIVE); break;
    case 39: (*((skepu::Scan<FUNC_NAME_BINARY9>*)m2))(*vec1,*vec2,
```

```
        skepu::INCLUSIVE); break;
    case 40: (*((skepu::Scan<FUNC_NAME_BINARY10>*)m2))(*vec1,*vec2,
        skepu::INCLUSIVE); break;

    case 41: (*((skepu::Scan<FUNC_NAME_UNARY1>*)m2))(*vec1,*vec2,
        skepu::INCLUSIVE); break;
    case 42: (*((skepu::Scan<FUNC_NAME_UNARY2>*)m2))(*vec1,*vec2,
        skepu::INCLUSIVE); break;
    case 43: (*((skepu::Scan<FUNC_NAME_UNARY3>*)m2))(*vec1,*vec2,
        skepu::INCLUSIVE); break;
    case 44: (*((skepu::Scan<FUNC_NAME_UNARY4>*)m2))(*vec1,*vec2,
        skepu::INCLUSIVE); break;
    case 45: (*((skepu::Scan<FUNC_NAME_UNARY5>*)m2))(*vec1,*vec2,
        skepu::INCLUSIVE); break;
    case 46: (*((skepu::Scan<FUNC_NAME_UNARY6>*)m2))(*vec1,*vec2,
        skepu::INCLUSIVE); break;
    case 47: (*((skepu::Scan<FUNC_NAME_UNARY7>*)m2))(*vec1,*vec2,
        skepu::INCLUSIVE); break;
    case 48: (*((skepu::Scan<FUNC_NAME_UNARY8>*)m2))(*vec1,*vec2,
        skepu::INCLUSIVE); break;
      //case 49: (*((skepu::Scan<FUNC_NAME_UNARY9>*)m2))(*vec1,*vec2,
          skepu::INCLUSIVE); break;
      //case 50: (*((skepu::Scan<FUNC_NAME_UNARY10>*)m2))(*vec1,*vec2,
          skepu::INCLUSIVE); break;

    case 51: (*((skepu::Scan<FUNC_NAME_BINARY_CONST1>*)m2))(*vec1,*
        vec2, skepu::INCLUSIVE); break;
    case 52: (*((skepu::Scan<FUNC_NAME_BINARY_CONST2>*)m2))(*vec1,*
        vec2, skepu::INCLUSIVE); break;
    case 53: (*((skepu::Scan<FUNC_NAME_BINARY_CONST3>*)m2))(*vec1,*
        vec2, skepu::INCLUSIVE); break;
    case 54: (*((skepu::Scan<FUNC_NAME_BINARY_CONST4>*)m2))(*vec1,*
        vec2, skepu::INCLUSIVE); break;
    case 55: (*((skepu::Scan<FUNC_NAME_BINARY_CONST5>*)m2))(*vec1,*
        vec2, skepu::INCLUSIVE); break;
    case 56: (*((skepu::Scan<FUNC_NAME_BINARY_CONST6>*)m2))(*vec1,*
        vec2, skepu::INCLUSIVE); break;
    case 57: (*((skepu::Scan<FUNC_NAME_BINARY_CONST7>*)m2))(*vec1,*
        vec2, skepu::INCLUSIVE); break;
    case 58: (*((skepu::Scan<FUNC_NAME_BINARY_CONST8>*)m2))(*vec1,*
        vec2, skepu::INCLUSIVE); break;
    case 59: (*((skepu::Scan<FUNC_NAME_BINARY_CONST9>*)m2))(*vec1,*
        vec2, skepu::INCLUSIVE); break;
    case 60: (*((skepu::Scan<FUNC_NAME_BINARY_CONST10>*)m2))(*vec1,*
        vec2, skepu::INCLUSIVE); break;

    case 61: (*((skepu::Scan<FUNC_NAME_UNARY_CONST1>*)m2))(*vec1,*vec2
        , skepu::INCLUSIVE); break;
    case 62: (*((skepu::Scan<FUNC_NAME_UNARY_CONST2>*)m2))(*vec1,*vec2
        , skepu::INCLUSIVE); break;
    case 63: (*((skepu::Scan<FUNC_NAME_UNARY_CONST3>*)m2))(*vec1,*vec2
        , skepu::INCLUSIVE); break;
    case 64: (*((skepu::Scan<FUNC_NAME_UNARY_CONST4>*)m2))(*vec1,*vec2
        , skepu::INCLUSIVE); break;
    case 65: (*((skepu::Scan<FUNC_NAME_UNARY_CONST5>*)m2))(*vec1,*vec2
        , skepu::INCLUSIVE); break;
    case 66: (*((skepu::Scan<FUNC_NAME_UNARY_CONST6>*)m2))(*vec1,*vec2
        , skepu::INCLUSIVE); break;
    case 67: (*((skepu::Scan<FUNC_NAME_UNARY_CONST7>*)m2))(*vec1,*vec2
        , skepu::INCLUSIVE); break;
    case 68: (*((skepu::Scan<FUNC_NAME_UNARY_CONST8>*)m2))(*vec1,*vec2
        , skepu::INCLUSIVE); break;
    case 69: (*((skepu::Scan<FUNC_NAME_UNARY_CONST9>*)m2))(*vec1,*vec2
        , skepu::INCLUSIVE); break;
    case 70: (*((skepu::Scan<FUNC_NAME_UNARY_CONST10>*)m2))(*vec1,*
        vec2, skepu::INCLUSIVE); break;

    default: m->skel = NULL; break;
```

```
    }
}

double SkePU_MapReduce_builtin_externalVdouble(void* object1, void*
    object2, void* object3)
{
  skepu::Vector<double>* vec1 = (skepu::Vector<double>*) object1;
  skepu::Vector<double>* vec2 = (skepu::Vector<double>*) object2;
  double res = 0;
  myStruct* m = (myStruct*)object3;
  int funcNum1 = m->funcNum1;
  int funcNum2 = m->funcNum2;
  void * m2 = m->skel;


 switch (funcNum1)
    {
    case 31:
      {
    switch (funcNum2) {
    case 32: res = (*((skepu::MapReduce<FUNC_NAME_BINARY1,
        FUNC_NAME_BINARY2>*)m2))(*vec1,*vec2);break;
    }
    break;}

    case 34:
      {
    switch (funcNum2) {
    case 35: res = (*((skepu::MapReduce<FUNC_NAME_BINARY4,
        FUNC_NAME_BINARY5>*)m2))(*vec1,*vec2);break;
    }
    break;}

    case 36:
      {
    switch (funcNum2) {
    case 40:res = (*((skepu::MapReduce<FUNC_NAME_BINARY6,
        FUNC_NAME_BINARY10>*)m2))(*vec1,*vec2);break;
    }
    break;}

    case 41:
      {switch (funcNum2) {
    case 32: res = (*((skepu::MapReduce<FUNC_NAME_UNARY1,
        FUNC_NAME_BINARY2>*)m2))(*vec1,*vec2);break;
    }
    break;}

    case 61:
      {switch (funcNum2) {
    case 32: res = (*((skepu::MapReduce<FUNC_NAME_UNARY_CONST1,
        FUNC_NAME_BINARY2>*)m2))(*vec1,*vec2);break;
    }
    break;}
    }


  return res;
}

double SkePU_MapReduce_builtin_externalV1double(void* object1, double
    inVal, void* object3)
{
  skepu::Vector<double>* vec1 = (skepu::Vector<double>*) object1;
  double res = 0;
  myStruct* m = (myStruct*)object3;
  int funcNum1 = m->funcNum1;
  int funcNum2 = m->funcNum2;
```

```
  void * m2 = m->skel;


 switch (funcNum1)
    {
    case 31:
      {
    switch (funcNum2) {
    case 32: res = (*((skepu::MapReduce<FUNC_NAME_BINARY1,
        FUNC_NAME_BINARY2>*)m2))(*vec1);
    }
    break;
      }

    case 34:
      {
    switch (funcNum2) {
    case 35: res = (*((skepu::MapReduce<FUNC_NAME_BINARY4,
        FUNC_NAME_BINARY5>*)m2))(*vec1);
    }
    break;
      }

    case 36:
      {
    switch (funcNum2) {
    case 40:res = (*((skepu::MapReduce<FUNC_NAME_BINARY6,
        FUNC_NAME_BINARY10>*)m2))(*vec1);
    }
    break;
      }

    case 41:
      {
    switch (funcNum2) {
    case 32: res = (*((skepu::MapReduce<FUNC_NAME_UNARY1,
        FUNC_NAME_BINARY2>*)m2))(*vec1);
    }
    break;
      }

    case 61:
      {
    switch (funcNum2) {
    case 32:
      {
        (*((skepu::MapReduce<FUNC_NAME_UNARY_CONST1,FUNC_NAME_BINARY2
            >*)m2)).setConstant(inVal);
        res = (*((skepu::MapReduce<FUNC_NAME_UNARY_CONST1,
            FUNC_NAME_BINARY2>*)m2))(*vec1);
      }
    }
    break;
      }

    }

  return res;
}


 double SkePU_MapReduce_builtin_externalMdouble(void* object1, void*
     object2, void* object3)
{
  skepu::Matrix<double>* mat1 = (skepu::Matrix<double>*) object1;
  skepu::Matrix<double>* mat2 = (skepu::Matrix<double>*) object2;
  double res = 0;
  myStruct* m = (myStruct*)object3;
```

```
  int funcNum1 = m->funcNum1;
  int funcNum2 = m->funcNum2;
  void * m2 = m->skel;

 switch (funcNum1)
    {
    case 31:
      {
    switch (funcNum2) {
    case 32: res = (*((skepu::MapReduce<FUNC_NAME_BINARY1,
        FUNC_NAME_BINARY2>*)m2))(*mat1,*mat2); break;
    }
    break;}

    case 34:
      {
    switch (funcNum2) {
    case 35: res = (*((skepu::MapReduce<FUNC_NAME_BINARY4,
        FUNC_NAME_BINARY5>*)m2))(*mat1,*mat2); break;
    }
    break;}

    case 36:
      {
    switch (funcNum2) {
    case 40: res = (*((skepu::MapReduce<FUNC_NAME_BINARY6,
        FUNC_NAME_BINARY10>*)m2))(*mat1,*mat2); break;
    }
    break;}

    case 41:
      {switch (funcNum2) {
    case 32:
      res = (*((skepu::MapReduce<FUNC_NAME_UNARY1,FUNC_NAME_BINARY2>*)
          m2))(*mat1,*mat2); break;
    }
    break;}

    case 61:
      {switch (funcNum2) {
    case 32: res = (*((skepu::MapReduce<FUNC_NAME_UNARY_CONST1,
        FUNC_NAME_BINARY2>*)m2))(*mat1,*mat2); break;
    }
    break;}
    }

  return res;
}


double SkePU_Reduce_builtin_externalMdouble(void* object1, void*
    object2)
{
  skepu::Matrix<double>* mat1 = (skepu::Matrix<double>*) object1;
  double res = 0;
  myStruct* m= (myStruct*)object2;
  int funcNum = m->funcNum1;
  void * m2 = m->skel;

  switch (funcNum)
    {
    case 31: res=(*((skepu::Reduce<FUNC_NAME_BINARY1>*)m2))(*mat1);
        break;
    case 32: res=(*((skepu::Reduce<FUNC_NAME_BINARY2>*)m2))(*mat1);
        break;
    case 33: res=(*((skepu::Reduce<FUNC_NAME_BINARY3>*)m2))(*mat1);
        break;
    case 34: res=(*((skepu::Reduce<FUNC_NAME_BINARY4>*)m2))(*mat1);
```

```
        break;
    case 35: res=(*((skepu::Reduce<FUNC_NAME_BINARY5>*)m2))(*mat1);
        break;
    case 36: res=(*((skepu::Reduce<FUNC_NAME_BINARY6>*)m2))(*mat1);
        break;
    case 37: res=(*((skepu::Reduce<FUNC_NAME_BINARY7>*)m2))(*mat1);
        break;
    case 38: res=(*((skepu::Reduce<FUNC_NAME_BINARY8>*)m2))(*mat1);
        break;
    case 39: res=(*((skepu::Reduce<FUNC_NAME_BINARY9>*)m2))(*mat1);
        break;
    case 40: res=(*((skepu::Reduce<FUNC_NAME_BINARY10>*)m2))(*mat1);
        break;

    case 41: res=(*((skepu::Reduce<FUNC_NAME_UNARY1>*)m2))(*mat1);
        break;
    case 42: res=(*((skepu::Reduce<FUNC_NAME_UNARY2>*)m2))(*mat1);
        break;
    case 43: res=(*((skepu::Reduce<FUNC_NAME_UNARY3>*)m2))(*mat1);
        break;
    case 44: res=(*((skepu::Reduce<FUNC_NAME_UNARY4>*)m2))(*mat1);
        break;
    case 45: res=(*((skepu::Reduce<FUNC_NAME_UNARY5>*)m2))(*mat1);
        break;
    case 46: res=(*((skepu::Reduce<FUNC_NAME_UNARY6>*)m2))(*mat1);
        break;
    case 47: res=(*((skepu::Reduce<FUNC_NAME_UNARY7>*)m2))(*mat1);
        break;
    case 48: res=(*((skepu::Reduce<FUNC_NAME_UNARY8>*)m2))(*mat1);
        break;
      //case 49: res=(*((skepu::Reduce<FUNC_NAME_UNARY9>*)m2))(*mat1);
          break;
      //case 50: res=(*((skepu::Reduce<FUNC_NAME_UNARY10>*)m2))(*mat1)
          ; break;

    case 51: res=(*((skepu::Reduce<FUNC_NAME_BINARY_CONST1>*)m2))(*
        mat1); break;
    case 52: res=(*((skepu::Reduce<FUNC_NAME_BINARY_CONST2>*)m2))(*
        mat1);
    case 53: res=(*((skepu::Reduce<FUNC_NAME_BINARY_CONST3>*)m2))(*
        mat1); break;
    case 54: res=(*((skepu::Reduce<FUNC_NAME_BINARY_CONST4>*)m2))(*
        mat1); break;
    case 55: res=(*((skepu::Reduce<FUNC_NAME_BINARY_CONST5>*)m2))(*
        mat1); break;
    case 56: res=(*((skepu::Reduce<FUNC_NAME_BINARY_CONST6>*)m2))(*
        mat1); break;
    case 57: res=(*((skepu::Reduce<FUNC_NAME_BINARY_CONST7>*)m2))(*
        mat1); break;
    case 58: res=(*((skepu::Reduce<FUNC_NAME_BINARY_CONST8>*)m2))(*
        mat1); break;
    case 59: res=(*((skepu::Reduce<FUNC_NAME_BINARY_CONST9>*)m2))(*
        mat1); break;
    case 60: res=(*((skepu::Reduce<FUNC_NAME_BINARY_CONST10>*)m2))(*
        mat1); break;

    case 61: res=(*((skepu::Reduce<FUNC_NAME_UNARY_CONST1>*)m2))(*mat1
        ); break;
    case 62: res=(*((skepu::Reduce<FUNC_NAME_UNARY_CONST2>*)m2))(*mat1
        ); break;
    case 63: res=(*((skepu::Reduce<FUNC_NAME_UNARY_CONST3>*)m2))(*mat1
        ); break;
    case 64: res=(*((skepu::Reduce<FUNC_NAME_UNARY_CONST4>*)m2))(*mat1
        ); break;
    case 65: res=(*((skepu::Reduce<FUNC_NAME_UNARY_CONST5>*)m2))(*mat1
        ); break;
    case 66: res=(*((skepu::Reduce<FUNC_NAME_UNARY_CONST6>*)m2))(*mat1
        ); break;
```

```
    case 67: res=(*((skepu::Reduce<FUNC_NAME_UNARY_CONST7>*)m2))(*mat1
        ); break;
    case 68: res=(*((skepu::Reduce<FUNC_NAME_UNARY_CONST8>*)m2))(*mat1
        ); break;
    case 69: res=(*((skepu::Reduce<FUNC_NAME_UNARY_CONST9>*)m2))(*mat1
        ); break;
    case 70: res=(*((skepu::Reduce<FUNC_NAME_UNARY_CONST10>*)m2))(*
        mat1); break;

    default: m->skel = NULL; break;
    }

  return res;
}

double SkePU_Reduce_builtin_externalVdouble(void* object1, void*
    object2)
{
  skepu::Vector<double>* vec1 = (skepu::Vector<double>*) object1;
  double res = 0;
  myStruct* m=(myStruct*)object2;
  int funcNum = m->funcNum1;
  void * m2 = m->skel;

  switch (funcNum)
    {
    case 31: res=(*((skepu::Reduce<FUNC_NAME_BINARY1>*)m2))(*vec1);
        break;
    case 32: res=(*((skepu::Reduce<FUNC_NAME_BINARY2>*)m2))(*vec1);
        break;
    case 33: res=(*((skepu::Reduce<FUNC_NAME_BINARY3>*)m2))(*vec1);
        break;
    case 34: res=(*((skepu::Reduce<FUNC_NAME_BINARY4>*)m2))(*vec1);
        break;
    case 35: res=(*((skepu::Reduce<FUNC_NAME_BINARY5>*)m2))(*vec1);
        break;
    case 36: res=(*((skepu::Reduce<FUNC_NAME_BINARY6>*)m2))(*vec1);
        break;
    case 37: res=(*((skepu::Reduce<FUNC_NAME_BINARY7>*)m2))(*vec1);
        break;
    case 38: res=(*((skepu::Reduce<FUNC_NAME_BINARY8>*)m2))(*vec1);
        break;
    case 39: res=(*((skepu::Reduce<FUNC_NAME_BINARY9>*)m2))(*vec1);
        break;
    case 40: res=(*((skepu::Reduce<FUNC_NAME_BINARY10>*)m2))(*vec1);
        break;

    case 41: res=(*((skepu::Reduce<FUNC_NAME_UNARY1>*)m2))(*vec1);
        break;
    case 42: res=(*((skepu::Reduce<FUNC_NAME_UNARY2>*)m2))(*vec1);
        break;
    case 43: res=(*((skepu::Reduce<FUNC_NAME_UNARY3>*)m2))(*vec1);
        break;
    case 44: res=(*((skepu::Reduce<FUNC_NAME_UNARY4>*)m2))(*vec1);
        break;
    case 45: res=(*((skepu::Reduce<FUNC_NAME_UNARY5>*)m2))(*vec1);
        break;
    case 46: res=(*((skepu::Reduce<FUNC_NAME_UNARY6>*)m2))(*vec1);
        break;
    case 47: res=(*((skepu::Reduce<FUNC_NAME_UNARY7>*)m2))(*vec1);
        break;
    case 48: res=(*((skepu::Reduce<FUNC_NAME_UNARY8>*)m2))(*vec1);
        break;
      //case 49: res=(*((skepu::Reduce<FUNC_NAME_UNARY9>*)m2))(*vec1);
          break;
      //case 50: res=(*((skepu::Reduce<FUNC_NAME_UNARY10>*)m2))(*vec1)
          ; break;
```

```
    case 51: res=(*((skepu::Reduce<FUNC_NAME_BINARY_CONST1>*)m2))(*
        vec1); break;
    case 52: res=(*((skepu::Reduce<FUNC_NAME_BINARY_CONST2>*)m2))(*
        vec1);
    case 53: res=(*((skepu::Reduce<FUNC_NAME_BINARY_CONST3>*)m2))(*
        vec1); break;
    case 54: res=(*((skepu::Reduce<FUNC_NAME_BINARY_CONST4>*)m2))(*
        vec1); break;
    case 55: res=(*((skepu::Reduce<FUNC_NAME_BINARY_CONST5>*)m2))(*
        vec1); break;
    case 56: res=(*((skepu::Reduce<FUNC_NAME_BINARY_CONST6>*)m2))(*
        vec1); break;
    case 57: res=(*((skepu::Reduce<FUNC_NAME_BINARY_CONST7>*)m2))(*
        vec1); break;
    case 58: res=(*((skepu::Reduce<FUNC_NAME_BINARY_CONST8>*)m2))(*
        vec1); break;
    case 59: res=(*((skepu::Reduce<FUNC_NAME_BINARY_CONST9>*)m2))(*
        vec1); break;
    case 60: res=(*((skepu::Reduce<FUNC_NAME_BINARY_CONST10>*)m2))(*
        vec1); break;

    case 61: res=(*((skepu::Reduce<FUNC_NAME_UNARY_CONST1>*)m2))(*vec1
        ); break;
    case 62: res=(*((skepu::Reduce<FUNC_NAME_UNARY_CONST2>*)m2))(*vec1
        ); break;
    case 63: res=(*((skepu::Reduce<FUNC_NAME_UNARY_CONST3>*)m2))(*vec1
        ); break;
    case 64: res=(*((skepu::Reduce<FUNC_NAME_UNARY_CONST4>*)m2))(*vec1
        ); break;
    case 65: res=(*((skepu::Reduce<FUNC_NAME_UNARY_CONST5>*)m2))(*vec1
        ); break;
    case 66: res=(*((skepu::Reduce<FUNC_NAME_UNARY_CONST6>*)m2))(*vec1
        ); break;
    case 67: res=(*((skepu::Reduce<FUNC_NAME_UNARY_CONST7>*)m2))(*vec1
        ); break;
    case 68: res=(*((skepu::Reduce<FUNC_NAME_UNARY_CONST8>*)m2))(*vec1
        ); break;
    case 69: res=(*((skepu::Reduce<FUNC_NAME_UNARY_CONST9>*)m2))(*vec1
        ); break;
    case 70: res=(*((skepu::Reduce<FUNC_NAME_UNARY_CONST10>*)m2))(*
        vec1); break;

    default: m->skel = NULL; break;
    }

  return res;
}
// END SKELETON FUNTIONS
```

Listing C.3: *run.sh*

```
#!/bin/bash

cp ../skepu_header.h .;
cp ../ExtObj.lib .;
../../openmodelica/build/bin/omc +s test_modelica_skepu.mos +d=
    failtrace;
rm *.o; rm *.mat; rm *.xml; rm *.libs; rm *.json; rm *.log;
```

Listing C.4: *skepu$_m$atrix.mo Modelica*

```
//*************************************************************
// skepu_matrix.mo
// Author: Kristian Stavaker, kristian.stavaker@liu.se
// Description:A Modelica file with Modelica object for the
//  SkePU library, that calls external C/C++.
//*************************************************************

package skepu_matrix

// SkePU Matrix and Vector
class SkePU_Matrix
  extends ExternalObject;
  function constructor
    input String type1;
    input Integer dim1;
    input Integer dim2;
    input Real initValue;
    output SkePU_Matrix mat;
    external "C" mat = initMyMatrix(type1,dim1,dim2,initValue);
    annotation(Include="#include \"skepu_header.h\"",Library="
        ExtObj.lib");
  end constructor;

  function destructor "Release storage of table"
    input SkePU_Matrix m;
    external "C" closeMyMatrix(m);
    annotation(Include="#include \"skepu_header.h\"",Library="
        ExtObj.lib");
  end destructor;
end SkePU_Matrix;

class SkePU_Vector
  extends ExternalObject;
  function constructor
    input String type1;
    input Integer dim1;
    input Real initValue;
    output SkePU_Vector vec;
    external "C" vec = initMyVector(type1, dim1 ,initValue);
    annotation(Include="#include \"skepu_header.h\"",Library="
        ExtObj.lib");
  end constructor;

  function destructor "Release storage of table"
    input SkePU_Vector v;
    external "C" closeMyVector(v);
     annotation(Include="#include \"skepu_header.h\"",Library="
         ExtObj.lib");
  end destructor;
end SkePU_Vector;

function getMatrixElement
  input SkePU_Matrix m;
  input Integer a1;
  input Integer a2;
  output Real elem;
  external "C" elem = getMatrixElement(m,a1,a2);
 annotation(Include="#include \"skepu_header.h\"",Library="ExtObj.lib"
     );
end getMatrixElement;

function getVectorElement
  input SkePU_Vector v;
  input Integer a1;
  output Real elem;
  external "C" elem=getVectorElement(v,a1);
 annotation(Include="#include \"skepu_header.h\"",Library="ExtObj.lib"
```

```
      );
end getVectorElement;

function assignMatrixElement
  input SkePU_Matrix m;
  input Integer a1;
  input Integer a2;
  input Real elem;
  external "C" assignMatrixElement(m,a1,a2,elem);
 annotation(Include="#include \"skepu_header.h\"",Library="ExtObj.lib"
      );
end assignMatrixElement;

function assignVectorElement
  input SkePU_Vector v;
  input Integer a1;
  input Real elem;
  external "C" assignVectorElement(v,a1,elem);
 annotation(Include="#include \"skepu_header.h\"",Library="ExtObj.lib"
      );
end assignVectorElement;

function displayDataVector
  input SkePU_Vector v;
  input String type1;
  external "C" displayDataVector(v,type1);
 annotation(Include="#include \"skepu_header.h\"",Library="ExtObj.lib"
      );
end displayDataVector;

function displayDataMatrix
  input SkePU_Matrix m;
  input String type1;
  external "C" displayDataMatrix(m,type1);
 annotation(Include="#include \"skepu_header.h\"",Library="ExtObj.lib"
      );
end displayDataMatrix;

function getRandNum
  output Real outNum;
  external "C" outNum=getRandNum();
  annotation(Include="#include \"skepu_header.h\"",Library="ExtObj.lib
       ");
end getRandNum;

function Vector_updateHost
  input SkePU_Vector v;
  external "C" vectorUpdatehost(v);
  annotation(Include="#include \"skepu_header.h\"",Library="ExtObj.lib
       ");
end Vector_updateHost;

end skepu_matrix;
```

Listing C.5: *skepu$_m$odelica.mo Modelica*

```
//*****************************************************************
// skepu_modelica.mo
// Author: Kristian Stavaker, kristian.stavaker@liu.se
// Description: A Modelica file with Modelica object for the
//  SkePU library, that calls external C/C++.
//*****************************************************************
```

```
package skepu_modelica

import skepu_matrix.mo;

// MAP SKELETON
class Map
  extends ExternalObject;
  function constructor
    input String funcName;
    output Map m;
    external "C" m = initMyMap(funcName);
    annotation(Include="#include \"skepu_header.h\"",Library="
        ExtObj.lib");
  end constructor;

  function destructor "Release storage of table"
    input Map m;
    external "C" closeMyMap(m);
    annotation(Include="#include \"skepu_header.h\"",Library="
        ExtObj.lib");
  end destructor;
end Map;

function Map_MM
  import SkePU_Matrix = skepu_matrix.SkePU_Matrix;
  import Map = skepu_modelica.Map;
  input Map map1;
  input SkePU_Matrix mat1;
  input SkePU_Matrix mat2;
  external "C" SkePU_Map_builtin_externalMdouble(mat1,mat2,map1);
  annotation(Include="#include \"skepu_header.h\"",Library="ExtObj.lib
      ");
end Map_MM;

function Map_V
  import SkePU_Vector = skepu_matrix.SkePU_Vector;
  import Map = skepu_modelica.Map;
  input Map map1;
  input SkePU_Vector vec1;
  external "C" SkePU_Map_builtin_externalV1double(vec1,map1);
  annotation(Include="#include \"skepu_header.h\"",Library="ExtObj.lib
      ");
end Map_V;

function Map_VV
  import SkePU_Vector = skepu_matrix.SkePU_Vector;
  import Map = skepu_modelica.Map;
  input Map map1;
  input SkePU_Vector vec1;
  input SkePU_Vector vec2;
  input String type1;
  external "C" SkePU_Map_builtin_externalVdouble(vec1,vec2,map1,type1)
      ;
  annotation(Include="#include \"skepu_header.h\"",Library="ExtObj.lib
      ");
end Map_VV;

function Map_VVV
  import SkePU_Vector = skepu_matrix.SkePU_Vector;
  import Map = skepu_modelica.Map;
  input Map map1;
  input SkePU_Vector vec1;
  input SkePU_Vector vec2;
  input SkePU_Vector vec3;
  input String type1;
  external "C" SkePU_Map_builtin_externalV2double(vec1,vec2,vec3,map1,
      type1);
  annotation(Include="#include \"skepu_header.h\"",Library="ExtObj.lib
```

```
      ");
end Map_VVV;

function Map_setConstant
  import SkePU_Vector = skepu_matrix.SkePU_Vector;
  import Map = skepu_modelica.Map;
  input Map map1;
  input Real val1;
  external "C" SkePU_Map_builtin_setConstant(map1,val1);
  annotation(Include="#include \"skepu_header.h\"",Library="ExtObj.lib
      ");
end Map_setConstant;
// END MAP SKELETON

// REDUCE SKELETON
class Reduce
  extends ExternalObject;
  function constructor
    input String funcName;
    output Reduce m;
    external "C" m = initMyReduce(funcName);
    annotation(Include="#include \"skepu_header.h\"",Library="
        ExtObj.lib");
  end constructor;

  function destructor "Release storage of table"
    input Reduce m;
    external "C" closeMyReduce(m);
    annotation(Include="#include \"skepu_header.h\"",Library="
        ExtObj.lib");
  end destructor;
end Reduce;

function Reduce_M
  import SkePU_Matrix = skepu_matrix.SkePU_Matrix;
  import Reduce = skepu_modelica.Reduce;
  input Reduce reduce1;
  input SkePU_Matrix mat1;
  output Real res;
  external "C" res = SkePU_Reduce_builtin_externalMdouble(mat1,reduce1
      );
  annotation(Include="#include \"skepu_header.h\"",Library="ExtObj.lib
      ");
end Reduce_M;

function Reduce_V
  import SkePU_Vector = skepu_matrix.SkePU_Vector;
  import Reduce = skepu_modelica.Reduce;
  input Reduce reduce1;
  input SkePU_Vector vec1;
  output Real res;
  external "C" res = SkePU_Reduce_builtin_externalVdouble(vec1,reduce1
      );
  annotation(Include="#include \"skepu_header.h\"",Library="ExtObj.lib
      ");
end Reduce_V;
// END REDUCE SKELETON


// MAPREDUCE SKELETON
class MapReduce
  extends ExternalObject;
  function constructor
    input String funcName1;
    input String funcName2;
    output MapReduce m;
    external "C" m = initMyMapReduce(funcName1, funcName2);
    annotation(Include="#include \"skepu_header.h\"",Library="
```

```
        ExtObj.lib");
  end constructor;

  function destructor "Release storage of table"
    input MapReduce m;
    external "C" closeMyMapReduce(m);
    annotation(Include="#include \"skepu_header.h\"",Library="
        ExtObj.lib");
  end destructor;
end MapReduce;

function MapReduce_VV
 import SkePU_Vector = skepu_matrix.SkePU_Vector;
 import MapReduce = skepu_modelica.MapReduce;
 input MapReduce mapreduce1;
 input SkePU_Vector vec1;
 input SkePU_Vector vec2;
 output Real res;
 external "C" res = SkePU_MapReduce_builtin_externalVdouble(vec1,vec2,
     mapreduce1);
 annotation(Include="#include \"skepu_header.h\"",Library="ExtObj.lib"
     );
end MapReduce_VV;

function MapReduce_VR
  import SkePU_Vector = skepu_matrix.SkePU_Vector;
  import MapReduce = skepu_modelica.MapReduce;
  input MapReduce mapreduce1;
  input SkePU_Vector vec1;
  input Real inVal;
  output Real res;
  external "C" res = SkePU_MapReduce_builtin_externalV1double(vec1,
      inVal,mapreduce1);
  annotation(Include="#include \"skepu_header.h\"",Library="ExtObj.lib
      ");
end MapReduce_VR;

function MapReduce_MM
  import SkePU_Matrix = skepu_matrix.SkePU_Matrix;
  import MapReduce = skepu_modelica.MapReduce;
  input MapReduce mapreduce1;
  input SkePU_Matrix mat1;
  input SkePU_Matrix mat2;
  output Real res;
  external "C" res = SkePU_MapReduce_builtin_externalMdouble(mat1,mat2
      ,mapreduce1);
  annotation(Include="#include \"skepu_header.h\"",Library="ExtObj.lib
      ");
end MapReduce_MM;
// END MAPREDUCE SKELETON


// MAPOVERLAP SKELETON
class MapOverlap
  extends ExternalObject;
  function constructor
    input String funcName;
    output MapOverlap m;
    external "C" m = initMyMapOverlap(funcName);
    annotation(Include="#include \"skepu_header.h\"",Library="
        ExtObj.lib");
  end constructor;

  function destructor "Release storage of table"
    input MapOverlap m;
    external "C" closeMyMapOverlap(m);
    annotation(Include="#include \"skepu_header.h\"",Library="
        ExtObj.lib");
```

```
  end destructor;
end MapOverlap;

function MapOverlap_VV
  import SkePU_Vector = skepu_matrix.SkePU_Vector;
  import MapOverlap = skepu_modelica.MapOverlap;
  input MapOverlap mapOverlap1;
  input SkePU_Vector vec1;
  input SkePU_Vector vec2;
  input Real r1;
  external "C" SkePU_MapOverlap_builtin_externalVdouble(vec1,vec2,r1,
      mapOverlap1);
  annotation(Include="#include \"skepu_header.h\"",Library="ExtObj.lib
      ");
end MapOverlap_VV;
// END MAPOVERLAP SKELETON


// MAPARRAY SKELETON
class MapArray
  extends ExternalObject;
  function constructor
    input String funcName;
    output MapArray m;
    external "C" m = initMyMapArray(funcName);
    annotation(Include="#include \"skepu_header.h\"",Library="
        ExtObj.lib");
  end constructor;

  function destructor "Release storage of table"
    input MapArray m;
    external "C" closeMyMapArray(m);
    annotation(Include="#include \"skepu_header.h\"",Library="
        ExtObj.lib");
  end destructor;
end MapArray;

function MapArray_VVV
  import SkePU_Vector = skepu_matrix.SkePU_Vector;
  import MapArray = skepu_modelica.MapArray;
  input MapArray mapArray1;
  input SkePU_Vector vec1;
  input SkePU_Vector vec2;
  input SkePU_Vector r1;
  input String type1;
  external "C" SkePU_MapArray_builtin_externalVdouble(vec1,vec2,r1,
      mapArray1,type1);
  annotation(Include="#include \"skepu_header.h\"",Library="ExtObj.lib
      ");
end MapArray_VVV;

function MapArray_VMM
  import SkePU_Vector = skepu_matrix.SkePU_Vector;
  import SkePU_Matrix = skepu_matrix.SkePU_Matrix;
  import MapArray = skepu_modelica.MapArray;
  input MapArray mapArray1;
  input SkePU_Vector vec;
  input SkePU_Matrix mat1;
  input SkePU_Matrix mat2;
  external "C" SkePU_MapArray_builtin_externalVMMdouble(vec,mat1,mat2,
      mapArray1);
  annotation(Include="#include \"skepu_header.h\"",Library="ExtObj.lib
      ");
end MapArray_VMM;
// END MAPARRAY SKELETON

// SCAN SKELETON
class Scan
```

```
  extends ExternalObject;
  function constructor
    input String funcName;
    output Scan m;
    external "C" m = initMyScan(funcName);
    annotation(Include="#include \"skepu_header.h\"",Library="
        ExtObj.lib");
  end constructor;

  function destructor "Release storage of table"
    input Scan m;
    external "C" closeMyScan(m);
    annotation(Include="#include \"skepu_header.h\"",Library="
        ExtObj.lib");
  end destructor;
end Scan;

function Scan_VV
  import SkePU_Vector = skepu_matrix.SkePU_Vector;
  import Scan = skepu_modelica.Scan;
  input Scan scan1;
  input SkePU_Vector vec1;
  input SkePU_Vector vec2;
  external "C" SkePU_Scan_builtin_externalVdouble(vec1,vec2,scan1);
  annotation(Include="#include \"skepu_header.h\"",Library="ExtObj.lib
      ");
end Scan_VV;
// END SCAN SKELETON

// GENERATE SKELETON
class Generate
  extends ExternalObject;
  function constructor
    input String funcName;
    output Generate m;
    external "C" m = initMyGenerate(funcName);
    annotation(Include="#include \"skepu_header.h\"",Library="
        ExtObj.lib");
  end constructor;

  function destructor "Release storage of table"
    input Generate m;
    external "C" closeMyGenerate(m);
    annotation(Include="#include \"skepu_header.h\"",Library="
        ExtObj.lib");
  end destructor;
end Generate;

function Generate_V
  import SkePU_Vector = skepu_matrix.SkePU_Vector;
  import Generate = skepu_modelica.Generate;
  input Generate generate1;
  input Integer numElem;
  input SkePU_Vector vec1;
  input String type1;
  external "C" SkePU_Generate_builtin_externalVdouble(numElem,vec1,
      generate1,type1);
  annotation(Include="#include \"skepu_header.h\"",Library="ExtObj.lib
      ");
end Generate_V;

function Generate_setConstant
  import Generate = skepu_modelica.Generate;
  input Generate generate1;
  input Integer elem;
  external "C" SkePU_Generate_builtin_setConstant(generate1,elem);
  annotation(Include="#include \"skepu_header.h\"",Library="ExtObj.lib
      ");
```

```
end Generate_setConstant;
// END GENERATE SKELETON

end skepu_modelica;
```

Listing C.6: *test_modelica_skepu.mos Modelica*

```
//*************************************************************************
    ********
// test_modelica_skepu.mos
// Author: Kristian Stavaker, kristian.stavaker@liu.se
// Description: A Modelica script file for testing the Modelica-SkePU
    library.
//See SkePU/skepu/tests/Makefile: -DSKEPU_OPENCL
//*************************************************************************
    ********
setCommandLineOptions({"+d=noevalfunc","+locale=en"});
loadModel(Modelica);
getErrorString();
loadFile("skepu_matrix.mo");
getErrorString();
loadFile("skepu_modelica.mo");
getErrorString();
loadFile("test_modelica_skepu.mo");
loadFile("seq_emb_rk_SkePU.mo");
getErrorString();
//checkModel(test_modelica_skepu.main1);
//checkModel(test_modelica_skepu.main2);
//checkModel(test_modelica_skepu.main3);
//checkModel(test_modelica_skepu.main4);
//checkModel(test_modelica_skepu.main5);
//checkModel(test_modelica_skepu.main6);
//checkModel(test_modelica_skepu.mandelbrot);
//checkModel(test_modelica_skepu.lufactor);
//checkModel(test_modelica_skepu.mse);
//checkModel(test_modelica_skepu.taylor_series);
//checkModel(test_modelica_skepu.psnr);
//checkModel(test_modelica_skepu.ppmcc);
//checkModel(test_modelica_skepu.nbody);
//checkModel(test_modelica_skepu.SPH_Fluid_Dynamics);
//checkModel(seq_emb_rk_SkePU.seq_emb_rk_implSkePU);
//instantiateModel(test_modelica_skepu.main1);
//instantiateModel(test_modelica_skepu.main2);
//instantiateModel(test_modelica_skepu.main3);
//instantiateModel(test_modelica_skepu.main4);
//instantiateModel(test_modelica_skepu.main5);
//instantiateModel(test_modelica_skepu.main6);
//instantiateModel(test_modelica_skepu.mandelbrot);
//instantiateModel(test_modelica_skepu.lufactor);
//instantiateModel(test_modelica_skepu.mse);
//instantiateModel(test_modelica_skepu.taylor_series);
//instantiateModel(test_modelica_skepu.psnr);
//instantiateModel(test_modelica_skepu.ppmcc);
//instantiateModel(test_modelica_skepu.nbody);
//instantiateModel(test_modelica_skepu.SPH_Fluid_Dynamics);
//instantiateModel(seq_emb_rk_SkePU.seq_emb_rk_implSkePU);
//simulate(test_modelica_skepu.main1, tolerance=1e-5,
    numberOfIntervals=1, method="euler");
//simulate(test_modelica_skepu.main2, tolerance=1e-5,
    numberOfIntervals=1, method="euler");
//simulate(test_modelica_skepu.main3, tolerance=1e-5,
    numberOfIntervals=1, method="euler");
//simulate(test_modelica_skepu.main4, tolerance=1e-5,
```

```
    numberOfIntervals=1, method="euler");
//simulate(test_modelica_skepu.main5, tolerance=1e-5,
    numberOfIntervals=1, method="euler");
//simulate(test_modelica_skepu.main6, tolerance=1e-5,
    numberOfIntervals=1, method="euler");
//simulate(test_modelica_skepu.mandelbrot, tolerance=1e-5,
    numberOfIntervals=1, method="euler");
//simulate(test_modelica_skepu.lufactor, tolerance=1e-5,
    numberOfIntervals=1, method="euler");
//simulate(test_modelica_skepu.mse, tolerance=1e-5, numberOfIntervals=
    1, method="euler");
//simulate(test_modelica_skepu.taylor_series, tolerance=1e-5,
    numberOfIntervals=1, method="euler");
//simulate(test_modelica_skepu.psnr, tolerance=1e-5, numberOfIntervals
    =1, method="euler");
//simulate(test_modelica_skepu.ppmcc, tolerance=1e-5,
    numberOfIntervals=1, method="euler");
//simulate(test_modelica_skepu.SPH_Fluid_Dynamics, tolerance=1e-5,
    numberOfIntervals=1, method="euler");
simulate(seq_emb_rk_SkePU.seq_emb_rk_implSkePU, tolerance=1e-5,
    numberOfIntervals=1, method="euler");
//getErrorString();
//g++ -I/home/krsta/SkePU/skepu/include  -c -o ExtObj.lib
    skepu_cpp.cpp -Wno-write-strings;
```

C.3 Modelica-SkePU Test Suite Models

C.3.1 Basic SkePU Models

Listing C.7: *main1 C++*

```
#include <iostream>
#include "skepu/matrix.h"
#include "skepu/map.h"

UNARY_FUNC(square_f, int, a,
    return a*a;
    )

int main()
{
    skepu::Map<square_f> square(new square_f);
    skepu::Matrix<int> m(5, 5, 3);
    skepu::Matrix<int> r(5, 5);
    square(m,r);

    std::cout << "Result: " << r << "\n";

    return 0;
}
```

Listing C.8: *main1 Modelica*

```
// Map Example
 function macro1
   external "SkePU" square_f();
   annotation(MacroType="UNARY_FUNC",Type1="double");
 end macro1;

 class main1
   import skepu_matrix.*;
   import skepu_modelica.*;

   Map square = Map(funcName="square_f");
   SkePU_Matrix m = SkePU_Matrix(type1="double",dim1=5,dim2=5,
        initValue=3.0);
   SkePU_Matrix r = SkePU_Matrix(type1="double",dim1=5,dim2=5,
        initValue=0.0);
 algorithm
   macro1();
 equation
   Map_MM(square,m,r);
   displayDataMatrix(r,"double");
 end main1;
```

Listing C.9: *main2 C++*

```
#include <iostream>
```

```
#include "skepu/matrix.h"
#include "skepu/reduce.h"

BINARY_FUNC(plus_f, float, a, b,
    return a+b;
    )

int main()
{
    skepu::Reduce<plus_f> globalSum(new plus_f);
    skepu::Matrix<float> m(25, 40, (float)3.5);
    float r = globalSum(m);
    std::cout << "Result: " << r << "\n";

    return 0;
}
```

Listing C.10: *main2 Modelica*

```
// Reduce Example
 function macro2
   external "SkePU" plus_f();
   annotation(MacroType="BINARY_FUNC",Type1="double");
 end macro2;

 class main2
   import skepu_matrix.*;
   import skepu_modelica.*;

   Reduce globalSum = Reduce(funcName="plus_f");
   SkePU_Matrix m = SkePU_Matrix(type1="double",dim1=25,dim2=40,
        initValue=3.5);
   Real r(start=0.0);
 algorithm
   macro2();
 equation
   r = Reduce_M(globalSum,m);
   Modelica.Utilities.Streams.print("Result = " + String(r));
 end main2;
```

Listing C.11: *main3 C++*

```
#include <iostream>

#include "skepu/vector.h"
#include "skepu/mapreduce.h"

BINARY_FUNC(plus_f, double, a, b,
    return a+b;
    )

BINARY_FUNC(mult_f, double, a, b,
    return a*b;
    )

int main()
{
```

```
    skepu::MapReduce<mult_f, plus_f> dotProduct(new mult_f,new plus_f)
        ;
    skepu::Vector<double> v1(500,4);
    skepu::Vector<double> v2(500,2);
    double r = dotProduct(v1,v2);
    std::cout << "Result: " << r << "\n";
    return 0;
}
```

Listing C.12: *main3 Modelica*

```
// MapReduce Example
function macro2
   external "SkePU" plus_f();
     annotation(MacroType="BINARY_FUNC",Type1="double");
 end macro2;

function macro3
   external "SkePU" mult_f();
   annotation(MacroType="BINARY_FUNC",Type1="double");
 end macro3;

 class main3
   import skepu_matrix.*;
   import skepu_modelica.*;

   MapReduce dotProduct = MapReduce(funcName1="mult_f",funcName2="
       plus_f");
   SkePU_Vector v1 = SkePU_Vector(type1="double",dim1=500, initValue=4
       .0);
   SkePU_Vector v2 = SkePU_Vector(type1="double",dim1=500, initValue=2
       .0);
   Real r(start=0.0);
 algorithm
   macro2();
   macro3();
 equation
   r = MapReduce_VV(dotProduct,v1,v2);
   Modelica.Utilities.Streams.print("Result = " + String(r));
 end main3;
```

Listing C.13: *main4 C++*

```
#include <iostream>

#include "skepu/vector.h"
#include "skepu/mapoverlap.h"

OVERLAP_FUNC(over_f, float, 2, b,
    return a[-2]*0.4f + a[-1]*0.2f + a[0]*0.1f +
        a[1]*0.2f + a[2]*0.4f;
    )

int main()
{
    skepu::MapOverlap<over_f> conv(new over_f);
    skepu::Vector<double> v(15,10);
    skepu::Vector<double> r;
```

```
    conv(v, r, skepu::CONSTANT, (float)1);
    std::cout << "Result: " << r << "\n";
    return 0;
}
```

Listing C.14: *main4 Modelica*

```
// MapOverlap Example
 function macro4
   external "SkePU" over_f();
   annotation(MacroType="OVER_FUNC",Type1="double",Over="2");
 end macro4;

 class main4
   import skepu_matrix.*;
   import skepu_modelica.*;
   MapOverlap conv = MapOverlap(funcName="over_f");
   SkePU_Vector v = SkePU_Vector(type1="double",dim1=15, initValue=10
       .0);
   SkePU_Vector r = SkePU_Vector(type1="double",dim1=15, initValue=0.0
       );
 algorithm
   macro4();
 equation
   MapOverlap_VV(conv,v,r,1.0);
   displayDataVector(r,"double");
 end main4;
```

Listing C.15: *main5 C++*

```
#include <iostream>

#include "skepu/vector.h"
#include "skepu/maparray.h"

ARRAY_FUNC(arr_f, double, a, b,
    int index = (int)b;
    return a[index];
    )

int main()
{
    skepu::MapArray<arr_f> reverse(new arr_f);
    skepu::Vector<double> v1(10);
    skepu::Vector<double> v2(10);
    skepu::Vector<double> r;
    for (int i = 0; i < 10; ++i)
    {
        v1[i] = i+1;
        v2[i] = 10 - i - 1;
    }
    reverse(v1, v2, r);
    std::cout << "Result: " << r << "\n";
    return 0;
}
```

Listing C.16: *main5 Modelica*

```
// MapArray Example
 function macro5
   external "SkePU" arr_f();
   annotation(MacroType="ARR_FUNC",Type1="double");
 end macro5;

 class main5
   import skepu_matrix.*;
   import skepu_modelica.*;

   MapArray reverse = MapArray(funcName="arr_f");
   SkePU_Vector v1 = SkePU_Vector(type1="double",dim1=10, initValue=0
       .0);
   SkePU_Vector v2 = SkePU_Vector(type1="double",dim1=10, initValue=0
       .0);
   SkePU_Vector r = SkePU_Vector(type1="double",dim1=10, initValue=0.0
       );
 algorithm
   macro5();
   for i in 0:9 loop
     assignVectorElement(v1,i,i+1);
     assignVectorElement(v2,i,10-i-1);
   end for;
 equation
   MapArray_VVV(reverse,v1,v2,r);
   displayDataVector(r,"double");
 end main5;
```

Listing C.17: *main6 C++*

```
#include <iostream>
#include "skepu/matrix.h"
#include "skepu/scan.h"

BINARY_FUNC(plus_f, int, a, b,
    return a+b;
    )

int main()
{
    skepu::Scan<plus_f> prefixSum(new plus_f);
    skepu::Vector<int> v(10,1);
    skepu::Vector<int> r;
    prefixSum(v, r, skepu::INCLUSIVE);
    std::cout << "Result: " << r << "\n";
    return 0;
}
```

Listing C.18: *main6 Modelica*

```
// Scan Example
function macro2
  external "SkePU" plus_f();
  annotation(MacroType="BINARY_FUNC",Type1="double");
end macro2;
```

```
class main6
  import skepu_matrix.*;
  import skepu_modelica.*;

  Scan prefixSum = Scan(funcName="plus_f");
  SkePU_Vector v = SkePU_Vector(type1="double",dim1=10, initValue=1.0)
      ;
  SkePU_Vector r = SkePU_Vector(type1="double",dim1=10, initValue=0.0)
      ;
algorithm
  macro2();
equation
  Scan_VV(prefixSum,v,r);
  displayDataVector(r,"double");
end main6;
```

C.3.2 Mandelbrot Fractals

Listing C.19: *main6*

```
// Mandelbrot Example
 function macro6
   external "SkePU" mandelbrote_f();
   annotation(MacroType="BINARY_FUNC",Type1="double");
 end macro6;

 class mandelbrot
   import skepu_matrix.*;
   import skepu_modelica.*;

   parameter Integer WIDTH= 40; //96;
   parameter Integer HEIGHT= 30; //72;
   parameter Integer ITER =1;
   parameter Real CENTER_X=-0.73;
   parameter Real CENTER_Y=-0.16;
   parameter Integer ZOOM=27615;
   SkePU_Vector in_def_img =
        SkePU_Vector(type1="double",dim1=WIDTH*HEIGHT,initValue=0.0);
   SkePU_Vector inout_img =
        SkePU_Vector(type1="double",dim1=WIDTH*HEIGHT,initValue=0.0);
   Map map_squ = Map(funcName="mandelbrote_f");
   Real startx(start = 0.0);
   Real starty(start = 0.0);
   Real dx(start = 0.0);
 algorithm
   //macro6();
   startx:=CENTER_X - (WIDTH/(ZOOM*2.0));
   starty:=CENTER_Y - (HEIGHT/(ZOOM*2.0));
   dx:= 1.0/ZOOM;
   for x in 0:(WIDTH-1) loop
     for y in 0:(HEIGHT-1) loop
       assignVectorElement(in_def_img,
                 x+y*WIDTH,startx + x*dx);
     end for;
   end for;

   for k in 0:(HEIGHT*WIDTH-1) loop
     assignVectorElement(inout_img,k,0.0);
   end for;
   for i in 1:ITER loop
```

```
    Map_VVV(map_squ,inout_img,in_def_img,inout_img);
  end for;
equation
  displayDataVector(inout_img,"double");
end mandelbrot;
```

C.3.3 LU Factorization

Listing C.20: *LU Factorization*

```
// LU Decomposition Example
function macro7
  external "SkePU" factorize_f();
  annotation(MacroType="ARRAY_FUNC",Type1="double");
end macro7;

function macro20
  external "SkePU" indexer_f();
  annotation(MacroType="GENERATE_FUNC",Type1="double");
end macro20;

function equalZero
  input Real inArg;
  output Boolean outArg;
algorithm
  outArg := (inArg == 0);
end equalZero;

class lufactor
  import skepu_matrix.*;
  import skepu_modelica.*;

  parameter Integer NTrials=1;
  parameter Integer N=10;
  parameter Integer RAND_MAX=2147483647;
  MapArray lu_factorize = MapArray(funcName="factorize_f");
  Generate set_indices = Generate(funcName="indexer_f");
  SkePU_Vector indices = SkePU_Vector(type1="double",dim1=N*N+1,
      initValue=0.0);
  SkePU_Vector A = SkePU_Vector(type1="double",dim1=N*N+1,initValue=0
      .0);
  SkePU_Vector LU = SkePU_Vector(type1="double",dim1=N*N+1,initValue=
      0.0);
  Real n(start=0.0);
algorithm
  macro7();
  macro20();
  for p in 0:(NTrials-1) loop

  // INIT MATRIX
   for i in 0:(N-1) loop
    for j in 0:(N-1) loop
       n := ceil(10*
                  (/*getRandNum()/RAND_MAX*/
                      1.0 -0.5));
        if equalZero(n) then
       n := 1.0;
     end if;
         assignVectorElement(A,i*N+j,n);
      end for;
    end for;
```

```
    // END INIT MATRIX

    // FACTORIZATION
    Generate_V(set_indices,N*N+1,indices,"double");

    for i in 0:(N-1) loop
      MapArray_VVV(lu_factorize,A,indices,LU);
      MapArray_VVV(lu_factorize,LU,indices,A);
    end for;
    // END FACTORIZATION
  end for;
  displayDataVector(indices,"double");
  displayDataVector(A,"double");
  displayDataVector(LU,"double");

  //Reset to zero
  for i in 0:(N-1) loop
    for j in 0:(N-1) loop
      assignVectorElement(LU,i*N+j,0.0);
      assignVectorElement(A,i*N+j,0.0);
      assignVectorElement(indices,i*N+j,0.0);
    end for;
  end for;
  assignVectorElement(LU,N*N,0.0);
  assignVectorElement(A,N*N,0.0);
  assignVectorElement(indices,N*N,0.0);
equation
end lufactor;
```

C.3.4 Mean Square Error (MSE)

Listing C.21: *Mean Square Error (MSE)*

```
// Mean Squared Error (MSE) Example
function macro8
  external "SkePU" diff_f();
  annotation(MacroType="BINARY_FUNC",Type1="double");
end macro8;

function macro9
  external "SkePU" sum_f();
  annotation(MacroType="BINARY_FUNC",Type1="double");
end macro9;

class mse
  import skepu_matrix.*;
  import skepu_modelica.*;

  parameter Integer ROWS=16; //00;
  parameter Integer COLS=12; //00;
  Map map_diff =
       Map(funcName="diff_f");
  Reduce red_sum =
       Reduce(funcName="sum_f");
  SkePU_Vector in_act_img = SkePU_Vector(type1="double",dim1=(ROWS*
       COLS),initValue=7.0);
  SkePU_Vector in_comp_img = SkePU_Vector(type1="double",dim1=(ROWS*
       COLS),initValue=1.5);
  SkePU_Vector out_img = SkePU_Vector(type1="double",dim1=(ROWS*COLS)
       ,initValue=0.0);
  Real mse(start=0.0);
```

```
  algorithm
    macro8();
    macro9();
    Map_VVV(map_diff,in_act_img,
         in_comp_img, out_img);
    mse := (Reduce_V(red_sum,out_img)/ROWS*COLS);
  equation
    displayDataVector(out_img,"double");
    Modelica.Utilities.Streams.print("Result = " + String(mse));
  end mse;
```

C.3.5 Pearson Product-Moment Correlation Coefficient (PPMCC)

Listing C.22: *Pearson Product-Moment Correlation Coefficient (PPMCC)*

```
// Pearson Product-Moment Correlation Coefficient (PPMCC)
 class ppmcc
    import skepu_matrix.*;
    import skepu_modelica.*;
    import Modelica.Math.*;

    parameter Integer N=50;
    SkePU_Vector X = SkePU_Vector(type1="double",dim1=N,initValue=3.5);
    SkePU_Vector Y = SkePU_Vector(type1="double",dim1=N,initValue=2.0);
    Reduce sumReduce = Reduce(funcName="plus_f");
    MapReduce dotProduct = MapReduce(funcName1="mult_f",funcName2="
        plus_f");
    MapReduce sumSquare = MapReduce(funcName1="square_f",funcName2="
        plus_f");
    Real res(start=0.0);
    Real sumX(start=0.0);
    Real sumY(start=0.0);
    Real sumPr(start=0.0);
    Real sumSqX(start=0.0);
    Real sumSqY(start=0.0);
 algorithm
    macro1();
    macro2();
    macro3();
    sumX := Reduce_V(sumReduce,X);
    sumY := Reduce_V(sumReduce,Y);
    Modelica.Utilities.Streams.print("Result = " + String(sumX));
    Modelica.Utilities.Streams.print("Result = " + String(sumY));
    sumPr := MapReduce_VV(dotProduct,X,Y);
    Modelica.Utilities.Streams.print("Result = " + String(sumPr));
    sumSqX := MapReduce_VR(sumSquare,X,0.0);
    Modelica.Utilities.Streams.print("Result = " + String(sumSqX));
    sumSqY := MapReduce_VR(sumSquare,Y,0.0);
    Modelica.Utilities.Streams.print("Result = " + String(sumSqY));
    res := ((N * sumPr - sumX * sumY) /
         sqrt((N * sumSqX - sumX*sumX) *
              (N * sumSqY - sumY*sumY)) );
    Modelica.Utilities.
         Streams.print("Result = " + String(res));
 equation
 end ppmcc;
```

C.3.6 Peak Signal to Noise Ratio (PSNR)

Listing C.23: *Peak Signal to Noise Ratio (PSNR)*

```
// PSNR Example
 class psnr
   import skepu_matrix.*;
   import skepu_modelica.*;

   parameter Integer ROWS=16;
   parameter Integer COLS=12;
   parameter Integer R=255;
   Map map_diff = Map(funcName="diff_f");
   Reduce red_sum = Reduce(funcName="sum_f");
   SkePU_Vector in_act_img = SkePU_Vector(type1="double",
            dim1=(ROWS*COLS),initValue=4.0);
   SkePU_Vector in_comp_img =SkePU_Vector(type1="double",
            dim1=(ROWS*COLS),initValue=2.0);
   SkePU_Vector out_img = SkePU_Vector(type1="double",
            dim1=(ROWS*COLS),initValue=0.0);
   Real mse(start=0.0);
   Real psnr(start=0.0);
 algorithm
   macro7();
   macro6();
   Map_VVV(map_diff,in_act_img, in_comp_img, out_img);
   mse := (Reduce_V(red_sum,out_img)/
        ROWS*COLS);
   psnr := 10 * log((R*R)/mse);
 equation
   Modelica.Utilities.Streams.print("Result = " + String(psnr));
 end psnr;
```

C.3.7 Taylor Series Calculation

Listing C.24: *Taylor Series Calculation*

```
// Taylor Series
 function macro19
   external "SkePU" nth_term();
   annotation(MacroType="UNARY_FUNC_CONSTANT",Type1="double",Type="
       double");
 end macro19;

 function macro20
   external "SkePU" lcg_init();
   annotation(MacroType="GENERATE_FUNC",Type1="double",Type2="double")
       ;
 end macro20;

 class taylor_series
   import skepu_matrix2.*;
   import skepu_modelica2.*;

   parameter Integer N = 100;
     MapReduce taylor = MapReduce(funcName1 = "nth_term",funcName2 =
         "plus_f");
```

```
    Generate vec_init = Generate(funcName = "lcg_init");
    SkePU_Vector v0 = SkePU_Vector(type1="double",dim1=N,initValue=0.0)
        ;
    Real result(start=0.0);
  algorithm
    macro2();
    macro19();
    macro20();
    Generate_V(vec_init,N,v0,"double");
    result := MapReduce_VR(taylor,v0,1.0);
    Modelica.Utilities.
        Streams.print("Result = " + String(result));
  equation
  end taylor_series;
```

C.3.8 Smooth Particle Hydrodynamics (SPH), Fluid Dynamics Shocktube simulation

Listing C.25: *Smooth Particle Hydrodynamics (SPH), Fluid Dynamics Shocktube simulation*

```
// SPH Fluid Dynamics
 function macro13
    external "SkePU" init();
    annotation(MacroType="GENERATE_FUNC",Type1="ParticleSPH",Type2="int
        ");
 end macro13;

 function macro14
    external "SkePU" assign();
    annotation(MacroType="UNARY_FUNC",Type1="ParticleSPH");
 end macro14;

 function macro15
    external "SkePU" updatecell();
    annotation(MacroType="ARRAY_FUNC",Type1="ParticleSPH");
 end macro15;

 function macro16
    external "SkePU" computedensity();
    annotation(MacroType="ARRAY_FUNC",Type1="ParticleSPH");
 end macro16;

 function macro17
    external "SkePU" updateforce();
    annotation(MacroType="ARRAY_FUNC",Type1="ParticleSPH");
 end macro17;

 function macro18
    external "SkePU" updateposition();
    annotation(MacroType="UNARY_FUNC",Type1="ParticleSPH");
 end macro18;

 class SPH_Fluid_Dynamics
    import skepu_matrix.*;
    import skepu_modelica.*;

    parameter Integer XYLEN = 100;
    parameter Integer NTrials = 3;
    parameter Integer XLEN = XYLEN;
```

```
  parameter Integer YLEN = XYLEN;
  parameter Integer ZLEN = 1;
  parameter Integer NPARTICLES =
       XLEN*YLEN*ZLEN;
  parameter Integer timesteps =
       100;
  Generate sph_init =
       Generate(funcName="sph_init");
  Map sph_assign =
       Map(funcName="sph_assign");
  MapArray sph_update_cell =
       MapArray(funcName="sph_updatecell");
  MapArray sph_compute_density =
       MapArray(funcName="sph_computedensity");
  MapArray sph_update_force =
       MapArray(funcName="sph_updateforce");
  Map sph_update_position =
       Map(funcName="sph_updateposition");
  SkePU_Vector fluid1 =
       SkePU_Vector(type1="ParticleSPH",
           dim1=NPARTICLES,initValue=0.0);
  SkePU_Vector fluid2 =
       SkePU_Vector(type1="ParticleSPH",dim1=NPARTICLES,initValue=0.0
           );
algorithm
  macro13();
  macro14();
  macro15();
  macro16();
  macro17();
  macro18();
  for i in 0:NTrials-1 loop
    Generate_V(sph_init,
           NPARTICLES, fluid1,"ParticleSPH");
    Map_VV(sph_assign,
           fluid1,fluid2,"ParticleSPH");
    MapArray_VVV(sph_update_cell,
           fluid2,fluid1,fluid1,"ParticleSPH");

    for i in 0:timesteps-1 loop
      MapArray_VVV(sph_compute_density,
             fluid1,fluid2,fluid2,"ParticleSPH");
      MapArray_VVV(sph_update_force,
             fluid2,fluid1,fluid1,"ParticleSPH");
      Map_VV(sph_update_position,
             fluid1,fluid1,"ParticleSPH");
    end for;
  end for;
  displayDataVector(fluid1,"ParticleSPH");
  displayDataVector(fluid2,"ParticleSPH");
equation
end SPH_Fluid_Dynamics;
```

C.3.9 A Runge-Kutta ODE solver

Listing C.26: *A Runge-Kutta ODE solver*

```
package seq_emb_rk_SkePU
import skepu_modelica.*;
import skepu_matrix.*;
```

```
function macro1
  external "SkePU" bruss_eval_f();
  annotation(MacroType="ARRAY_FUNC",Type1="double");
end macro1;

function macro2
  external "SkePU" absmax_f();
  annotation(MacroType="BINARY_FUNC",Type1="double");
end macro2;

function macro3
  external "SkePU" axpy_f();
  annotation(MacroType="BINARY_FUNC_CONSTANT",Type1="double");
end macro3;

function macro4
  external "SkePU" sub_f();
  annotation(MacroType="BINARY_FUNC",Type1="double");
end macro4;

function macro5
  external "SkePU" zero_f();
  annotation(MacroType="UNARY_FUNC",Type1="double");
end macro5;

function macro6
  external "SkePU" copy_f();
  annotation(MacroType="UNARY_FUNC",Type1="double");
end macro6;

function macro7
  external "SkePU" absaxpy_f();
  annotation(MacroType="BINARY_FUNC_CONSTANT",Type1="double");
end macro7;

function macro8
  external "SkePU" scale_f();
  annotation(MacroType="UNARY_FUNC_CONSTANT",Type1="double");
end macro8;

function macro9
  external "SkePU" absquot_f();
  annotation(MacroType="BINARY_FUNC",Type1="double");
end macro9;

function macro10
  external "SkePU" maximum_f();
  annotation(MacroType="BINARY_FUNC",Type1="double");
end macro10;

class seq_emb_rk_implSkePU
  import skepu_matrix.*;
  import skepu_modelica.*;

  constant Integer ode_size = 32;
  constant Integer s = 7;
  constant Integer ord = 5;
  constant Integer bruss_grid_size = 4;
  constant Real t0 = 0.0;
  constant Real bf = 1.000000E-03;
  constant Real H = 4.0;

  MapArray bruss_eval = MapArray(funcName="bruss_eval_f");
  Reduce absmax = Reduce(funcName="absmax_f");
  Map axpy = Map(funcName="axpy_f");
  MapReduce subAbsmax = MapReduce(funcName1="sub_f", funcName2="
      absmax_f");
  Map zero = Map(funcName="zero_f");
```

```
  Map copy = Map(funcName="copy_f");
  Map absaxpy = Map(funcName="absaxpy_f");
  Map scale = Map(funcName="scale_f");
  MapReduce absquotmax = MapReduce(funcName1="absquot_f",funcName2="
      maximum_f");

  // INPUT VARIABLES
  Real _y0[ode_size];
  Real b[s];
  Real bs[s];
  Real a[s,s];
  // INPUT VARIABLES END

  Real error_max,h,old_h,t,t_e,d0,d1,d2,h0,h1;
  Real bbs[s];
  Real stagevec[s,ode_size];

  SkePU_Vector g_eval_index = SkePU_Vector(type1="double",dim1=
      ode_size, initValue=0.0);
  SkePU_Vector y = SkePU_Vector(type1="double",dim1=ode_size,
      initValue=0.0);
  SkePU_Vector f_t0 = SkePU_Vector(type1="double",dim1=ode_size,
      initValue=0.0);
  SkePU_Vector y1 = SkePU_Vector(type1="double",dim1=ode_size,
      initValue=0.0);
  SkePU_Vector f_t0h0 = SkePU_Vector(type1="double",dim1=ode_size,
      initValue=0.0);
  SkePU_Vector tempStage = SkePU_Vector(type1="double",dim1=ode_size,
      initValue=0.0);
  SkePU_Vector old_y = SkePU_Vector(type1="double",dim1=ode_size,
      initValue=0.0);
  SkePU_Vector help = SkePU_Vector(type1="double",dim1=ode_size,
      initValue=0.0);
  SkePU_Vector help1 = SkePU_Vector(type1="double",dim1=ode_size,
      initValue=0.0);
  SkePU_Vector err_vector = SkePU_Vector(type1="double",dim1=ode_size,
       initValue=0.0);
  SkePU_Vector yscal = SkePU_Vector(type1="double",dim1=ode_size,
      initValue=0.0);
  SkePU_Vector y0 = SkePU_Vector(type1="double",dim1=ode_size,
      initValue=0.0);
algorithm
  // INITIALIZATION, bruss_mix_start
  for i in 0:bruss_grid_size-1 loop
    for j in 0:bruss_grid_size-1 loop
      _y0[2 * bruss_grid_size * i + 2 * j + 1] := 0.5 + j / (
          bruss_grid_size - 1);
    end for;
  end for;

  for i in 0:bruss_grid_size-1 loop
    for j in 0:bruss_grid_size-1 loop
      _y0[2 * bruss_grid_size * i + 2 * j + 2] :=
    1.0 + (5.0 * i) / (bruss_grid_size - 1);
   end for;
  end for;
  // END INITIALIZATION, bruss_mix_start

  // Quelle: Hairer, Bd. I, Butcher 2003 (S. 194)
  b[1] := 35.0 / 384.0;
  b[2] := 0.0;
  b[3] := 500.0 / 1113.0;
  b[4] := 125.0 / 192.0;
  b[5] := -2187.0 / 6784.0;
  b[6] := 11.0 / 84.0;
  b[7] := 0.0;
  bs[1] := 5179.0 / 57600;
  bs[2] := 0.0;
```

```
  bs[3] := 7571.0 / 16695;
  bs[4] := 393.0 / 640.0;
  bs[5] := -92097.0 / 339200;
  bs[6] := 187.0 / 2100.0;
  bs[7] := 1.0 / 40.0;
  a[1,1] := 0.0;
  a[1,2] := 0.0;
  a[1,3] := 0.0;
  a[1,4] := 0.0;
  a[1,5] := 0.0;
  a[1,6] := 0.0;
  a[1,7] := 0.0;
  a[2,1] := 1.0 / 5.0;
  a[2,2] := 0.0;
  a[2,3] := 0.0;
  a[2,4] := 0.0;
  a[2,5] := 0.0;
  a[2,6] := 0.0;
  a[2,7] := 0.0;
  a[3,1] := 3.0 / 40.0;
  a[3,2] := 9.0 / 40.0;
  a[3,3] := 0.0;
  a[3,4] := 0.0;
  a[3,5] := 0.0;
  a[3,6] := 0.0;
  a[3,7] := 0.0;
  a[4,1] := 44.0 / 45.0;
  a[4,2] := -56.0 / 15.0;
  a[4,3] := 32.0 / 9.0;
  a[4,4] := 0.0;
  a[4,5] := 0.0;
  a[4,6] := 0.0;
  a[4,7] := 0.0;
  a[5,1] := 19372.0 / 6561.0;
  a[5,2] := -25360.0 / 2187.0;
  a[5,3] := 64448.0 / 6561.0;
  a[5,4] := -212.0 / 729.0;
  a[5,5] := 0.0;
  a[5,6] := 0.0;
  a[5,7] := 0.0;
  a[6,1] := 9017.0 / 3168.0;
  a[6,2] := -355.0 / 33.0;
  a[6,3] := 46732.0 / 5247.0;
  a[6,4] := 49.0 / 176.0;
  a[6,5] := -5103.0 / 18656.0;
  a[6,6] := 0.0;
  a[6,7] := 0.0;
  a[7,1] := 35.0 / 384.0;
  a[7,2] := 0.0;
  a[7,3] := 500.0 / 1113.0;
  a[7,4] := 125.0 / 192.0;
  a[7,5] := -2187.0 / 6784.0;
  a[7,6] := 11.0 / 84.0;
  a[7,7] := 0.0;
  // INITIALIZATION STOP

  // FUNCTION: INIT SKEPU
  for i in 0:(ode_size-1) loop
    assignVectorElement(g_eval_index,i,i);
  end for;
  // END FUNCTION: INIT SKEPU

  t := t0;
  t_e := t0 + H;
  for i in 1:s loop
    bbs[i] := b[i] - bs[i];
    //Modelica.Utilities.Streams.print(String(bbs[i]));
  end for;
```

```
for i in 0:(ode_size-1) loop
   assignVectorElement(y,i,_y0[i+1]);
   assignVectorElement(y0,i,_y0[i+1]);
end for;

// FUNCTION: INITIAL STEP SKEPU
//Modelica.Utilities.Streams.print("H = " + String(H) + "\n");
d0 := Reduce_V(absmax,y);
//displayDataVector(y,"double");
//Modelica.Utilities.Streams.print("d0 = " + String(d0) + "\n");

MapArray_VVV(bruss_eval,y,g_eval_index,f_t0,"double");

d1 := Reduce_V(absmax,f_t0);
//Modelica.Utilities.Streams.print("d1 = " + String(d1) + "\n");

if ((d0 < 1E-5) or (d1 < 1E-5)) then
  h0 := 1E-6;
else
  h0 := 0.01 * (d0 / d1);
end if;
//Modelica.Utilities.Streams.print("h0 = " + String(h0) + "\n");

Map_setConstant(axpy,h0);

Map_VVV(axpy,f_t0,y,y1,"double");

MapArray_VVV(bruss_eval,y1,g_eval_index,f_t0h0,"double");
d2 := MapReduce_VV(subAbsmax,f_t0h0,f_t0);
d2 := d2 / h0;
//Modelica.Utilities.Streams.print("d2 = " + String(d2) + "\n");

if (max(d1,d2) < 1E-15) then
  h1 := max(1E-6, h0 * 1E-3);
else
  h1 := (0.01 / max(d1, d2))^(1.0 / (ord + 1.0));
end if;
h0 := min(100.0 * h0, h1);
//Modelica.Utilities.Streams.print("h1 = " + String(h1) + "\n");
//Modelica.Utilities.Streams.print("h0 = " + String(h0) + "\n");
h := min(h0, H);
// END INITIAL STEP SKEPU

//Modelica.Utilities.Streams.print("h = " + String(h) + "\n");
//Modelica.Utilities.Streams.print("y0 = \n");
//displayDataVector(y0,"double");

Modelica.Utilities.Streams.print("=== WHILE LOOP START ===\n");

while (t < t_e) loop
  Modelica.Utilities.Streams.print("t = " + String(t) + " h = " +
       String(h) + "\n");
  displayDataVector(y,"double");

  for i in 0:(s-1) loop
    Map_V(zero,help1);

    for j in 0:(i-1) loop
      Map_setConstant(axpy,a[i+1,j+1]);

  // Temp storage//
  for k in 0:(ode_size-1) loop
     assignVectorElement(tempStage,k,stagevec[j+1,k+1]);
      end for;

      Map_VVV(axpy,tempStage /*stagevec[j+1]*/, help1, help1,"double
          ");
```

```
      end for;

      Map_setConstant(axpy,h);
      Map_VVV(axpy,help1, y, help1,"double");

    // Temp storage//
    for k in 0:(ode_size-1) loop
       assignVectorElement(tempStage,k,stagevec[i+1,k+1]);
        end for;

      MapArray_VVV(bruss_eval,help1,g_eval_index,tempStage /*stagevec[
          i+1]*/,"double");
      // Temp storage//
    for k in 0:(ode_size-1) loop
       stagevec[i+1,k+1] := getVectorElement(tempStage,k);
        end for;


    end for;

   Map_V(zero,help);
   Map_V(zero,help1);

   for i in 0:(s-1) loop
      Map_setConstant(axpy,bbs[i+1]);

      // Temp storage //
    for k in 0:(ode_size-1) loop
       assignVectorElement(tempStage,k,stagevec[i+1,k+1]);
        end for;

      Map_VVV(axpy,tempStage /*stagevec[i+1]*/, help, help,"double");

      Map_setConstant(axpy,b[i+1]);
      // Temp storage //
      for k in 0:(ode_size-1) loop
        assignVectorElement(tempStage,k,stagevec[i+1,k+1]);
      end for;

      Map_VVV(axpy,tempStage /*stagevec[i+1]*/, help1, help1,"double")
          ;
   end for;

    for i in 0:(ode_size-1) loop
      assignVectorElement(old_y,i,getVectorElement(y,i));
    end for;

    Map_setConstant(absaxpy,h);
    // Temp storage //
      for k in 0:(ode_size-1) loop
        assignVectorElement(tempStage,k,stagevec[1,k+1]);
      end for;

    Map_VVV(absaxpy,y, tempStage /*stagevec[1]*/, yscal,"double");

    Map_setConstant(axpy,h);
    Map_VVV(axpy,help1, y, y,"double");
    Map_setConstant(scale,h);
    Map_VV(scale,help,err_vector,"double");

    old_h := h;
    error_max := MapReduce_VV(absquotmax,err_vector, yscal);
    error_max := error_max / bf;

    Modelica.Utilities.Streams.print("Error max: \n" + String(
        error_max));

    if (error_max <= 1.0) then     // accept
```

```
      h := h * max((1.0 / 3.0), 0.9 * (1.0 / error_max)^(1.0 / (ord +
          1.0)));
      t := t + old_h;
    else                          // reject
      h := max(0.1 * h, 0.9 * h * (1.0 / error_max)^(1.0 / ord));
      for i in 0:(ode_size-1) loop
        assignVectorElement(y,i,getVectorElement(old_y,i));
      end for;
    end if;

      h := min(h, t_e - t);
    end while;

    Modelica.Utilities.Streams.print("=== WHILE LOOP END ===\n");

    displayDataVector(y,"double");

    // SET TO ZERO
    Map_V(zero,help);
    Map_V(zero,help1);
    Map_V(zero,y0);
    Map_V(zero,y);
    Map_V(zero,g_eval_index);
    Map_V(zero,f_t0);
    Map_V(zero,y1);
    Map_V(zero,f_t0h0);
    Map_V(zero,tempStage);
    Map_V(zero,old_y);
    Map_V(zero,err_vector);
    Map_V(zero,yscal);
    Map_V(zero,f_t0h0);
    d0 := 0.0;
    d1 := 0.0;
    d2 := 0.0;
    h0 := 0.0;
    h1 := 0.0;
    error_max := 0.0;
    h := 0.0;
    old_h := 0.0;
    t := 0.0;
    t_e := 0.0;
    for i in 1:s loop
      for j in 1:ode_size loop
        stagevec[i,j] := 0.0;
      end for;
    end for;
    // -----------
equation
end seq_emb_rk_implSkePU;

end seq_emb_rk_SkePU;
```

Listing C.27: $skepu_macro_functions.h$

```
  //+=======+//
  // MACROS //
 //+=======+//
 #include <math.h>

 #define FUNC_NAME_GENERATE1 lcg_init
 #define FUNC_NAME_GENERATE2 indexer_f
 #define FUNC_NAME_GENERATE3 initNB
 #define FUNC_NAME_GENERATE4 sph_init
```

```
#define FUNC_NAME_GENERATE5 lcg_init
#define FUNC_NAME_GENERATE6 lcg_init
#define FUNC_NAME_GENERATE7 lcg_init
#define FUNC_NAME_GENERATE8 lcg_init
#define FUNC_NAME_GENERATE9 lcg_init
#define FUNC_NAME_GENERATE10 lcg_init

#define FUNC_NAME_OVERLAP1 over_f
#define FUNC_NAME_OVERLAP2 over_f
#define FUNC_NAME_OVERLAP3 over_f
#define FUNC_NAME_OVERLAP4 over_f
#define FUNC_NAME_OVERLAP5 over_f
#define FUNC_NAME_OVERLAP6 over_f
#define FUNC_NAME_OVERLAP7 over_f
#define FUNC_NAME_OVERLAP8 over_f
#define FUNC_NAME_OVERLAP9 over_f
#define FUNC_NAME_OVERLAP10 over_f

#define FUNC_NAME_ARRAY1 arr_f
#define FUNC_NAME_ARRAY2 factorize_f
#define FUNC_NAME_ARRAY3 arr_f
#define FUNC_NAME_ARRAY4 arr_f
#define FUNC_NAME_ARRAY5 arr_f
#define FUNC_NAME_ARRAY6 bruss_eval_f
#define FUNC_NAME_ARRAY7 sph_updatecell
#define FUNC_NAME_ARRAY8 sph_computedensity
#define FUNC_NAME_ARRAY9 sph_updateforce
#define FUNC_NAME_ARRAY10 move

#define FUNC_NAME_BINARY1 mult_f
#define FUNC_NAME_BINARY2 plus_f
#define FUNC_NAME_BINARY3 diff_f
#define FUNC_NAME_BINARY4 sub_f
#define FUNC_NAME_BINARY5 absmax_f
#define FUNC_NAME_BINARY6 absquot_f
#define FUNC_NAME_BINARY7 sum_f
#define FUNC_NAME_BINARY8 mandelBrote_f
#define FUNC_NAME_BINARY9 mult_f
#define FUNC_NAME_BINARY10 maximum_f

#define FUNC_NAME_UNARY1 square_f
#define FUNC_NAME_UNARY2 copy_f
#define FUNC_NAME_UNARY3 zero_f
#define FUNC_NAME_UNARY4 zero_f
#define FUNC_NAME_UNARY5 zero_f
#define FUNC_NAME_UNARY6 zero_f
#define FUNC_NAME_UNARY7 zero_f
#define FUNC_NAME_UNARY8 zero_f
#define FUNC_NAME_UNARY9 sph_updateposition
#define FUNC_NAME_UNARY10 sph_assign

#define FUNC_NAME_BINARY_CONST1 axpy_f
#define FUNC_NAME_BINARY_CONST2 absaxpy_f
#define FUNC_NAME_BINARY_CONST3 absaxpy_f
#define FUNC_NAME_BINARY_CONST4 absaxpy_f
#define FUNC_NAME_BINARY_CONST5 absaxpy_f
#define FUNC_NAME_BINARY_CONST6 absaxpy_f
#define FUNC_NAME_BINARY_CONST7 absaxpy_f
#define FUNC_NAME_BINARY_CONST8 absaxpy_f
#define FUNC_NAME_BINARY_CONST9 absaxpy_f
#define FUNC_NAME_BINARY_CONST10 absaxpy_f

#define FUNC_NAME_UNARY_CONST1 nth_term
#define FUNC_NAME_UNARY_CONST2 scale_f
#define FUNC_NAME_UNARY_CONST3 scale_f
#define FUNC_NAME_UNARY_CONST4 scale_f
#define FUNC_NAME_UNARY_CONST5 scale_f
#define FUNC_NAME_UNARY_CONST6 scale_f
```

```
#define FUNC_NAME_UNARY_CONST7 scale_f
#define FUNC_NAME_UNARY_CONST8 scale_f
#define FUNC_NAME_UNARY_CONST9 scale_f
#define FUNC_NAME_UNARY_CONST10 scale_f

#define TYPE_NAME1 ParticleNB
#define TYPE_NAME2 ParticleSPH
#define float double

UNARY_FUNC_CONSTANT(nth_term, float, float, t, x,
                    float temp_x = pow(x, t);
                    return (((int)t)%2==0?-1:1)*temp_x/t;
                   )

// UNARY AND BINARY FUNCTIONS
UNARY_FUNC(square_f, double, a,
       return a*a;
       )
/*int square_f(double a)
{
  return a*a;
  }*/

BINARY_FUNC(plus_f, double, a, b,
        return a+b;
        )
/*double plus_f(double a, double b)
{
  return a+b;
  }*/

BINARY_FUNC(diff_f, double, a, b,
        return ((a-b)*(a-b));
        )
/*double diff_f(double a, double b)
{
  return ((a-b)*(a-b));
}*/

BINARY_FUNC(sum_f, double, a, b,
        return a+b;
        )
/*double sum_f(double a, double b)
{
  return a+b;
  }*/

BINARY_FUNC(mult_f, double, a, b,
        return a*b;
        )
/*double mult_f(double a, double b)
{
  return a*b;
  }*/

// OVERLAP FUNCTIONS
OVERLAP_FUNC(over_f, double, 2, a,
         return a[-2]*0.4f + a[-1]*0.2f + a[0]*0.1f +
         a[1]*0.2f + a[2]*0.4f;
         )
/*double over_f(double *a)
{
  return a[-2]*0.4f + a[-1]*0.2f + a[0]*0.1f +
    a[1]*0.2f + a[2]*0.4f;
    }*/

// ARRAY FUNCTIONS
ARRAY_FUNC(arr_f, double, a, b,
```

```
        int index = (int)b;
        return a[index];
        )
/*double arr_f(double[] a, double b)
{
  int index = (int)b;
  return a[index];
  }*/

GENERATE_FUNC(lcg_init, double, double, index, seed,
              return index+1;
           )
/*double lcg_init(double seed, int index)
{
  return (double)(index+1);
  }*/

BINARY_FUNC(mandelBrote_f, double, a, b,
        return a*a+b;
        )

GENERATE_FUNC(indexer_f, int, int, index, seed,
              return index+1;
             )

#define PROBLEM_SIZE 1000
#define NLU PROBLEM_SIZE
#define D(i,j)    (i*NLU + j)

ARRAY_FUNC(factorize_f, float, A, ind,
           int index = (int)(ind-1);
           if(index > NLU*NLU)
           return A[index];
           int col = index%NLU;
           int row = ((index-col)/NLU)%NLU;
           int iteration = (int)A[NLU*NLU];
           int LorU = (int)10*(A[NLU*NLU] - (float)iteration);

           if(index == NLU*NLU)
{
   if(LorU == 0)
         return (A[index] + 0.1f);

      return (iteration + 1.0f);
   }

if(LorU == 0)
{
if((col == iteration) && (row > col))
   {
      return A[index]/A[D(col,col)];
   }
}
else if(LorU == 1)
{
if( (col > iteration) && (row > iteration) )
   {
      return (A[index] - A[D(row,iteration)]*A[D(iteration,col)]);
   }
}

return A[index];
          )


struct ParticleNB
{
   double id;
```

```
    double x, y, z;
    double vx, vy, vz;
    double ax, ay, az;
    double m;
};

// some parameter constants. can change here....
#define NP 2 // Number of particles in 1 direction
#define G 1
#define time_steps 2
#define delta_t 1

/*
   Array user-function that is used for applying nbody computation,
   All elements from parr and a single element (named 'p_1') are
        accessible
   to produce one output element of the same type.
*/
ARRAY_FUNC(move, ParticleNB, parr, p_1,
           int i = p_1.id;
           p_1.ax = 0.0;
           p_1.ay = 0.0;
           p_1.az = 0.0;

           double rij = 0;
           double dum = 0;

           for(int j=0; j<NP*NP*NP; ++j)
{
if(i!=j)
   {
      ParticleNB pj = parr[j];

      rij = sqrt((p_1.x-pj.x)*(p_1.x-pj.x) + (p_1.y-pj.y)*(p_1.y-pj.y
           ) + (p_1.z-pj.z)*(p_1.z-pj.z));

      dum = G * (pj.m) / pow(rij,3);

      p_1.ax =  p_1.ax + dum * (p_1.x-pj.x);
      p_1.ay =  p_1.ay + dum * (p_1.y-pj.y);
      p_1.az =  p_1.az + dum * (p_1.z-pj.z);
   }
}

p_1.x = parr[i].x + delta_t * parr[i].vx + ((delta_t*delta_t)/2)*(
     parr[i].ax);
        p_1.y = parr[i].y + delta_t * parr[i].vy + ((delta_t*delta_t)
             /2)*(parr[i].ay);
        p_1.z = parr[i].z + delta_t * parr[i].vz + ((delta_t*delta_t)
             /2)*(parr[i].az);

        p_1.vx = parr[i].vx + (delta_t/2)*(parr[i].ax + p_1.ax);
        p_1.vy = parr[i].vy + (delta_t/2)*(parr[i].ay + p_1.ay);
        p_1.vz = parr[i].vz + (delta_t/2)*(parr[i].az + p_1.az);

        return p_1;
          )

/*
   Generate user-function that is used for initializing particles
        array.
*/
GENERATE_FUNC(initNB, ParticleNB, int, index, seed,
              int s = index;
              int d = NP/2+1;
              int i = s%NP;
              int j = ((s-i)/NP)%NP;
              int k = (((s-i)/NP)-j)/NP;
```

```
                ParticleNB p;

                p.id = s;

                p.x = i-d+1;
                p.y = j-d+1;
                p.z = k-d+1;

                p.vx = 0.0;
                p.vy = 0.0;
                p.vz = 0.0;
                p.ax = 0.0;
                p.ay = 0.0;
                p.az = 0.0;

                p.m = 1;

                return p;
               )

#define XYLEN 30
#define XLEN              XYLEN
#define YLEN              XYLEN
#define ZLEN              1
#define NPARTICLES          (XLEN*YLEN*ZLEN)
#define SMOOTHING_LENGTH      (1.00/NPARTICLES)
#define SEARCH_RADIUS          (1*SMOOTHING_LENGTH)
#define MAX_FLOAT            3.40282347e+36
#define GRID_CACHE          10
#define MASS              0.00020543
#define PI              3.1415926
#define STIFF              1.5
#define VISCOSITY          0.2
#define TIME_STEP          0.003
#define EPSILON          0.00001

#define GLASS_R           0.05
#define GLASS_BOTTOM              -0.08
#define GLASS_TOP           0.06
#define GLASS_THICKNESS       0.01

#define poly6_coef         (315.0/(64.0*PI*pow(SMOOTHING_LENGTH,9)))
#define grad_poly6_coef          (945.0/(32.0*PI*pow(SMOOTHING_LENGTH
    ,9)))
#define lap_poly6_coef         (945.0/(32.0*PI*pow(SMOOTHING_LENGTH,9)
    ))
#define grad_spiky_coef          (-45.0/(PI*pow(SMOOTHING_LENGTH,6)))
#define lap_vis_coef         (45.0/(PI*pow(SMOOTHING_LENGTH,6)))

struct ParticleSPH
{
   int id;             // id of the particle
   double x,y,z;         // coordinate of the particle
   double vx,vy,vz;        // velocity of the particle
   double vhx,vhy,vhz;         // velocity half
   double ax,ay,az;        // acceleration of particle
   double m;             // mass of the particle
   double p;             // pressure of the particle
   double d;             // density of particle
   int pool[GRID_CACHE];     // pool
   int neighbours[GRID_CACHE]; // neighbours list
};



#define s  0.006
#define cx 0.0
```

```
#define cy 0.0
#define cz 0.035
#define s2 0.001
#define s3 0.0001

GENERATE_FUNC(sph_init, ParticleSPH, int, index, seed,

              int z = index%ZLEN;
              int y = ((index-z)/ZLEN)%YLEN;
              int x = (((index-z)/ZLEN)-y)/XLEN;

              double rand1 = (((int)(10*s*x*s2+4*cz*s3+5*y*z + 10*s*y
                   ))%RAND_MAX) - 0.5;
              double rand2 = (((int)(9*s*y*s2+3*z*s3+3*y*z*s + 5*x*y)
                   )%RAND_MAX) - 0.5;

              ParticleSPH p;

              p.id     = index;
              p.x        = s * (x -XLEN/2)- cx + s2 * rand1;
              p.y        = s * (y - YLEN/2) - cy + s2 * rand2;
              p.z        = 0.8 * s * z - cz;
              p.vx        = 0.0;
              p.vy        = 0.0;
              p.vz        = 0.0;
              p.ax        = 0.0;
              p.ay        = 0.0;
              p.az        = 0.0;
              p.vhx    = 0.0;
              p.vhy    = 0.0;
              p.vhz    = 0.0;
              p.d        = 0.0;
              p.p        = 0.0;
              p.m        = 0.0;

              p.pool[0]    = 0;
              p.pool[1]    = 0;
              p.pool[2]    = 0;
              p.pool[3]    = 0;

              return p;
             )

ARRAY_FUNC(sph_updatecell, ParticleSPH, parr, p,

           int i = p.id;
           p = parr[i];
           double dist = 0.0;

           for(int j = 0; j < NPARTICLES; j++)
{
if( i != j)
   {
      ParticleSPH pj = parr[j];

      dist = ((p.x-pj.x)*(p.x-pj.x) + (p.y-pj.y)*(p.y-pj.y) + (p.z-pj
           .z)*(p.z-pj.z));

      if(dist < (SMOOTHING_LENGTH*SMOOTHING_LENGTH))
      {
         if(p.pool[0] < GRID_CACHE-1)
         {
            p.pool[0]++;
            p.pool[p.pool[0]] = j;
         }
      }
   }
}
```

```
return p;
          )

ARRAY_FUNC(sph_computedensity, ParticleSPH, parr, pi,

            int i = pi.id;
            pi = parr[i];
            pi.neighbours[0] = 0;

            pi.d = 0;
            pi.p = 0;

            double h2_r2 = 0.0;
            double dist  = 0.0;

            for (int j=0; j<NPARTICLES; j++)
{
if (i!=j)
   {
      ParticleSPH pj = parr[j];

      dist = ((pi.x-pj.x)*(pi.x-pj.x) + (pi.y-pj.y)*(pi.y-pj.y) + (pi
           .z-pj.z)*(pi.z-pj.z));

      if((dist < (SMOOTHING_LENGTH*SMOOTHING_LENGTH)) && (pi.
           neighbours[0] < GRID_CACHE-1))
      {
         h2_r2 = (SMOOTHING_LENGTH*SMOOTHING_LENGTH) - dist;

         pi.d += 2 * MASS * h2_r2 * h2_r2 * h2_r2;
         pi.neighbours[0]++;
         pi.neighbours[pi.neighbours[0]] = j;
      }
   }
}

if(pi.neighbours[0] < GRID_CACHE-1)
{
for (int k=pi.neighbours[0]+1; k<GRID_CACHE; k++)
   {
      pi.neighbours[k]=0;
   }
}

pi.d *= poly6_coef;
        pi.d  = (pi.d < 0.00001)?0.0:(1.0 / pi.d);
        pi.p  = STIFF * (pi.d - 1000.0);

        return pi;
          )

ARRAY_FUNC(sph_updateforce, ParticleSPH, parr, pi,

            pi = parr[pi.id];
            pi.ax = 0.0;
            pi.ay = 0.0;
            pi.az = 0.0;

            double dist;
            double h_r;
            double grad_spiky;
            double lap_vis;
            double dist_x;
            double dist_y;
            double dist_z;
            double force_x;
            double force_y;
            double force_z;
```

```
            double vdiff_x;
            double vdiff_y;
            double vdiff_z;
            double prod;

            for (int j=1; j<pi.neighbours[0]; j++)
{
ParticleSPH pj = parr[pi.neighbours[j]];

   dist_x = pi.x - pj.x;
   dist_y = pi.y - pj.y;
   dist_z = pi.z - pj.z;

   dist = sqrt(dist_x * dist_x + dist_y * dist_y + dist_z * dist_z);

   if (dist<SMOOTHING_LENGTH*SMOOTHING_LENGTH)
      dist=SMOOTHING_LENGTH*SMOOTHING_LENGTH;

   h_r          =     SMOOTHING_LENGTH - dist;
   grad_spiky    =    grad_spiky_coef * pi.d * pj.d * 2 * MASS * h_r;
   lap_vis        =   lap_vis_coef * pi.d * pj.d * 2 * MASS * h_r;
   prod          =     (-0.5 * (pi.p + pj.p) * grad_spiky * h_r / dist
       );

   force_x         =    prod*dist_x;
   force_y         =    prod*dist_y;
   force_z         =    prod*dist_z;

   vdiff_x         =    (VISCOSITY*lap_vis) * (pj.vx - pi.vx);
   vdiff_y         =    (VISCOSITY*lap_vis) * (pj.vy - pi.vy);
   vdiff_z         =    (VISCOSITY*lap_vis) * (pj.vz - pi.vz);

   force_x     +=    vdiff_x;
   force_y     +=    vdiff_y;
   force_z        +=    vdiff_z;

   pi.ax         +=    force_x;
   pi.ay         +=    force_y;
   pi.az         +=    force_z;
}

return pi;
          )

UNARY_FUNC(sph_updateposition, ParticleSPH, pi,

            double e        =    1.0;
            double sphere_radius=     0.004;
            double stiff    =    30000.0;
            double damp        =    128.0;


            double col_x    =    0.0;
            double col_y    =    0.0;
            double col_z    =    0.0;
            double pre_px    =    0.0;
            double pre_py    =    0.0;
            double pre_pz    =    0.0;
            double vhx        =    0.0;
            double vhy        =    0.0;
            double vhz        =    0.0;

            pre_px        =    pi.x + TIME_STEP * pi.vhx;
            pre_py        =    pi.y + TIME_STEP * pi.vhy;
            pre_pz        =    pi.z + TIME_STEP * pi.vhz;

            vhx            =    pi.vhx + TIME_STEP * pi.ax;
            vhy            =    pi.vhy + TIME_STEP * pi.ay;
```

```
            vhz             =     pi.vhz + TIME_STEP * pi.az;

            pi.x         =    pi.x + TIME_STEP * vhx;
            pi.y         =    pi.y + TIME_STEP * vhy;
            pi.z         =    pi.z + TIME_STEP * vhz;

            pi.vx         =    0.5 * (pi.vhx + vhx);
            pi.vy         =    0.5 * (pi.vhy + vhy);
            pi.vz         =    0.5 * (pi.vhz + vhz);

            pi.vhx         =    vhx;
            pi.vhy         =    vhy;
            pi.vhz         =    vhz;

            return pi;
           )

UNARY_FUNC(sph_assign, ParticleSPH, pi,
            return pi;
           )

// ---

BINARY_FUNC_CONSTANT(axpy_f, double, double, a, b, alpha,
    return a*alpha+b;
)

UNARY_FUNC(copy_f, double, a,
    return a;
)

BINARY_FUNC_CONSTANT(absaxpy_f, double, double, a, b, h,
    return ( (fabs(a) + fabs(h * b)) + 1.0e-30 );
)

UNARY_FUNC(zero_f, double, a,
    return 0.0f;
)

UNARY_FUNC_CONSTANT(scale_f, double, double, a, h,
    return h*a;
)

BINARY_FUNC(absquot_f, double, a, b,
    return ( fabs(a / b) );
)

BINARY_FUNC(maximum_f, double, a, b,
    return ( fmax(a,b) );
)

BINARY_FUNC(sub_f, double, a, b,
    return ( a - b );
)

BINARY_FUNC(absmax_f, double, a, b,
    return ( fmax( fabs(a), fabs(b) ) );
)

ARRAY_FUNC(bruss_eval_f, double, y, d_i,
    uint i = (uint)d_i;
    double alpha = 0.000200f;
    uint N = 16;
    double N1 = (double) N - 1.0f;
    uint N2 = N + N;
    uint k = i / N2;
    uint j = i - k * N2;
    uint v = i & 1;
```

```
    j >>= 1;

    if (!v)
    {
        if (k == 0)
        {
            if (j == 0)
                return 1.0 + y[i] * y[i] * y[i + 1] - 4.4 * y[i] +
                    alpha * N1 * N1 * (2.0 * y[i + N2] + 2.0 * y[i +
                     2] - 4.0 * y[i]);

            if (j == N - 1)
                return 1.0 + y[i] * y[i] * y[i + 1] - 4.4 * y[i] +
                    alpha * N1 * N1 * (2.0 * y[i + N2] + 2.0 * y[i -
                     2] - 4.0 * y[i]);

            return 1.0 + y[i] * y[i] * y[i + 1] - 4.4 * y[i] + alpha
                 * N1 * N1 * (2.0 * y[i + N2] + y[i - 2] + y[i + 2] -
                  4.0 * y[i]);
        }
        else if (k == N - 1)
        {
            if (j == 0)
                return 1.0 + y[i] * y[i] * y[i + 1] - 4.4 * y[i] +
                    alpha * N1 * N1 * (2.0 * y[i - N2] + 2.0 * y[i +
                     2] - 4.0 * y[i]);

            if (j == N - 1)
                return 1.0 + y[i] * y[i] * y[i + 1] - 4.4 * y[i] +
                    alpha * N1 * N1 * (2.0 * y[i - N2] + 2.0 * y[i -
                     2] - 4.0 * y[i]);

            return 1.0 + y[i] * y[i] * y[i + 1] - 4.4 * y[i] + alpha
                 * N1 * N1 * (2.0 * y[i - N2] + y[i - 2] + y[i + 2] -
                  4.0 * y[i]);
        }
        else
        {
            if (j == 0)
                return 1.0 + y[i] * y[i] * y[i + 1] - 4.4 * y[i] +
                    alpha * N1 * N1 * (y[i - N2] + y[i + N2] + 2.0 *
                     y[i + 2] - 4.0 * y[i]);

            if (j == N - 1)
                return 1.0 + y[i] * y[i] * y[i + 1] - 4.4 * y[i] +
                    alpha * N1 * N1 * (y[i - N2] + y[i + N2] + 2.0 *
                     y[i - 2] - 4.0 * y[i]);

            return 1.0 + y[i] * y[i] * y[i + 1] - 4.4 * y[i] + alpha
                 * N1 * N1 * (y[i - N2] + y[i + N2] + y[i - 2] + y[i
                 + 2] - 4.0 * y[i]);
        }
    }
    else
    {
        if (k == 0)
        {
            if (j == 0)
                return 3.4 * y[i - 1] - y[i - 1] * y[i - 1] * y[i] +
                    alpha * N1 * N1 * (2.0 * y[i + N2] + 2.0 * y[i +
                     2] - 4.0 * y[i]);

            if (j == N - 1)
                return 3.4 * y[i - 1] - y[i - 1] * y[i - 1] * y[i] +
                    alpha * N1 * N1 * (2.0 * y[i + N2] + 2.0 * y[i -
                     2] - 4.0 * y[i]);

            return 3.4 * y[i - 1] - y[i - 1] * y[i - 1] * y[i] +
```

```
                    alpha * N1 * N1 * (2.0 * y[i + N2] + y[i - 2] + y[i
                    + 2] - 4.0 * y[i]);
            }
            else if (k == N - 1)
            {
                if (j == 0)
                    return 3.4 * y[i - 1] - y[i - 1] * y[i - 1] * y[i] +
                        alpha * N1 * N1 * (2.0 * y[i - N2] + 2.0 * y[i +
                         2] - 4.0 * y[i]);

                if (j == N - 1)
                    return 3.4 * y[i - 1] - y[i - 1] * y[i - 1] * y[i] +
                        alpha * N1 * N1 * (2.0 * y[i - N2] + 2.0 * y[i -
                         2] - 4.0 * y[i]);

                return 3.4 * y[i - 1] - y[i - 1] * y[i - 1] * y[i] +
                    alpha * N1 * N1 * (2.0 * y[i - N2] + y[i - 2] + y[i
                    + 2] - 4.0 * y[i]);
            }
            else
            {
                if (j == 0)
                    return 3.4 * y[i - 1] - y[i - 1] * y[i - 1] * y[i] +
                        alpha * N1 * N1 * (y[i - N2] + y[i + N2] + 2.0 *
                         y[i + 2] - 4.0 * y[i]);

                if (j == N - 1)
                    return 3.4 * y[i - 1] - y[i - 1] * y[i - 1] * y[i] +
                        alpha * N1 * N1 * (y[i - N2] + y[i + N2] + 2.0 *
                         y[i - 2] - 4.0 * y[i]);

                return 3.4 * y[i - 1] - y[i - 1] * y[i - 1] * y[i] +
                    alpha * N1 * N1 * (y[i - N2] + y[i + N2] + y[i - 2]
                    + y[i + 2] - 4.0 * y[i]);
            }
        }
)
```

Bibliography

[1] Amd graphics, [accessed 8 april, 2015]. http://www.amd.com/en-us/products/graphics.

[2] Comsol multiphysics, [accessed 8 april, 2015]. http://www.comsol.com/.

[3] HPC-Openmodelica for Multiscale Simulations of Techninal Systems and Applications for the Development of Energy-Efficient Working Machinery, [accessed 8 April, 2015]. http://www.hpc-om.de/.

[4] Mali Cost Efficient Graphics, [accessed 8 April, 2015]. http://www.arm.com/products/multimedia/mali-cost-efficient-graphics/index.php.

[5] Maple and MapleSim by MapleSoft, [accessed 8 April, 2015]. http://www.maplesoft.com/products.

[6] *NVIDIA CUDA Programming Guide [accessed 23 April, 2015]*, v. 2.0 edition. http://www.nvidia.com/object/cuda_home_new.html.

[7] *OpenACC Specification [accessed 23 April, 2015]*, v. 2.0a edition. http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf.

[8] SkePU - Autotunable Multi-Backend Skeleton Programming Framework for Multicore CPU and Multi-GPU Systems, [accessed 8 April, 2015]. http://www.ida.liu.se/ chrke/skepu/.

[9] The Functional Mockup Interface (FMI). https://www.fmi-standard.org/.

[10] The HiFlow3 Multi-Purpose Finite Element Software, [accessed 8 April, 2015]. http://www.hiflow3.org/.

[11] The ITEA3 MODELISAR Project, [accessed 8 April, 2015]. https://itea3.org/project/modelisar.html.

[12] Engineering Mathematics and Computing Lab (EMCL) Publication Database, September 2014. http://emcl.iwr.uni-heidelberg.de/79.html.

[13] Preprint Series of Engineering Mathematics and Computing Lab (EMCL), September 2014. https://journals.ub.uni-heidelberg.de/index.php/emcl-pp/issue/archive.

[14] Adrian Pop and Kristian Stavåker and Peter Fritzson. Exception Handling for Modelica. In *Proceedings of the 6th International Modelica Conference (Modelica'2008), Bielefeld, Germany, March.3-4.*, 2008.

[15] Afshin Hemmati Moghadam and Mahder Gebremedhin and Kristian Stavåker and Peter Fritzson. Simulation and Benchmarking of Modelica Models on Multi-Core Architectures with Explicit Parallel Algorithmic Language Extensions. *MCC'11 Workshop, Linköping, Sweden, November 23-25.*, 2011.

[16] Alexander Siemers and Dag Fritzson and Peter Fritzson. Meta-Modeling for Multi-Physics Co-Simulations applied for OpenModelica. In *Proceedings of International Congress on Methodologies for Emerging Technologies in Automation (ANIPLA), November*, 2006.

[17] Alexandre Ern and Jean-Luc Guermond. *Theory and Practice of Finite Elements.* Springer, 2010.

[18] Alfred V. Aho and Monica S. Lam and Ravi Sethi and Jeffrey D. Ullman. *Compilers - Principles, Techniques, and Tools. Second edition.* Pearson International Edition, 2007.

[19] Niclas Andersson. *Compilation of Mathematical Models to Parallel Code. Licentiate thesis 563.* Linköping University, 1996.

[20] Andreas Heckmann and Martin Otter and Stefan Dietz and José Díaz López. The DLR FlexibleBodies library to model large motions of beams and of flexible bodies exported from finite element programs. In *Proceedings of the 5th International Modelica Conference (Modelica'2006), Vienna, Austria, September 4-5*, 2006.

[21] Peter Aronsson. *Automatic Parallelization of Equation-Based Simulation Programs.* PhD thesis, Linköping University, 2006. Dissertation No. 1022.

[22] David Broman. *Safety, Security, and Semantic Aspects of Equation-Based Object-Oriented Languages and Environment. Licentiate thesis 1337.* Linköping University, 2007.

[23] The Cell project at IBM Research, [accessed April 8, 2015]. http://www.research.ibm.com/cell/.

[24] Francois Cellier and Ernesto Kofman. *Continuous System Simulation.* Springer, 2006.

[25] Christoph Kessler and Peter Fritzson and Mattias Eriksson. NestStepModelica: Mathematical Modeling and Bulk-synchronous Parallel Simulation. In *Proceedings of the 8th International Conference on Applied Parallel Computing: State of the Art in Scientific Computing*, 2006.

[26] C.L. Dym and I.H. Shames. *Solid Mechanics: A Variational Approach, Augmented Edition*. Springer New York, 2013.

[27] DASSL, Solution of Differential Algebraic Equation, [accessed April 8, 2015]. http://www.oecd-nea.org/tools/abstract/detail/nesc9918/.

[28] Usman Dastgeer. *Skeleton Programming for Heterogeneous GPU-based Systems. Licentiate Thesis No. 1504*. Linköping University, 2011.

[29] Usman Dastgeer. *Performance-aware Component Composition for GPU-based Systems*. PhD thesis, Linköping University, 2014. Dissertation No. 1581.

[30] David Black-Schaffer, Division of Computer Systems, Department of Information Technology, Uppsala University, [accessed 8 April, 2015]. *Introduction to OpenCL*, 2011. http://www.it.uu.se/katalog/davbl791/intro_to_opencl.pdf.

[31] Dietrich Braess. *Finite Elemente*. Springer, 2007.

[32] Ulrich Drepper. What every programmer should know about memory, 2007.

[33] Ernst Hairer and Syvert P. Norsett and Gerhard Wanner. *Solving Ordinary Differential Equations*. Springer, 2008.

[34] Farid Dshabarow. Support for Dymola in the Modeling and Simulation of Physical Systems with Distributed Parameters. Master's thesis, Department of Computer Science, Institute of Computational Science, ETH Zürich, 2007.

[35] Fermi, Next Generation CUDA Architecture, [accessed April 8, 2015]. http://www.nvidia.com/object/fermi_architecture.html.

[36] Xenofon Flores. *Exploiting Model Structure for Efficient Hybrid Dynamical Systems Simulation*. PhD thesis, ETH Zurich, 2014.

[37] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, 2004.

[38] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: a Cyber-Physical Approach*. IEEE Press, 2015.

[39] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20(1):359--392, December 1998.

[40] Harald Kosch and László Böszörményi and Hermann Hellwagner, editor. *Euro-Par 2003. Parallel Processing, 9th International Euro-Par Conference, Klagenfurt, Austria, August 26-29, 2003. Proceedings*, volume 2790 of *Lecture Notes in Computer Science*. Springer, 2003.

[41] Harro Heuser. *Lehrbuch der Analysis*. Teubner, 2002.

[42] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards*, 49:409--436, 1952.

[43] Håkan Lundvall and Kristian Stavåker and Peter Fritzson and Christoph Kessler. Automatic Parallelization of Simulation Code for Equation-based Models with Software Pipelining and Measurements on Three Platforms. *MCC'08 Workshop, Ronneby, Sweden, November 27-28*, 2008.

[44] Hopsan - An Integrated Simulation Environment for Simulation of Fluid Power Systems, [accessed April 8, 2015]. http://www.iei.liu.se/flumes/system-simulation/hopsan?l=en.

[45] Johan Enmyren and Christoph W. Kessler. SkePU: A multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of the 4th International Workshop on High-Level Parallel Programming and Applications (HLPP-2010), Baltimore, Maryland, USA. ACM, September*, 2010.

[46] Johan Enmyren and Usman Dastgeer and Christoph Kessler. Towards a Tunable Multi-Backend Skeleton Programming Framework for Multi-GPU Systems. In *Proceedings of the MCC-2010 Third Swedish Workshop on Multicore Computing, Gothenburg, Sweden, November*, 2010.

[47] Karl Johan Åström and Tore Hägglund. *PID Controllers: Theory, Design, and Tuning. 2nd edition*. Instrument Society of American, 1995.

[48] Ernesto Kofman. *Discrete Event Based Simulation and Control of Continuous Systems*. PhD thesis, School of Electronic Engineering - FCEIA Universidad Nacional de Rosario, 2003.

[49] Kristian Stavåker and Chen Song and Martin Wlotzka and Vincent Heuveline and Peter Fritzson. PDE Modeling with Modelica via FMI Import of HiFlow3 C++ Components with Parallel Multi-Core Simulations. *Proceedings of SIMS 55th Conference, Aalborg, Denmark*, 2014.

[50] Kristian Stavåker and Staffan Ronnås and Martin Wlotzka and Vincent Heuveline and Peter Fritzson. PDE Modeling with Modelica via FMI Import of HiFlow3 C++ Components. *Proceedings of SIMS 54th Conference, Bergen, Norway*, 2013.

[51] Kristian Stavåker and Adrian Pop and Peter Fritzson. Compiling and Using Pattern Matching in Modelica. In *Proceedings of the 6th International Modelica Conference (Modelica'2008), Bielefeld, Germany, March.3-4.*, 2008.

[52] Kristian Stavåker and Daniel Rolls and Jing Guo and Peter Fritzson and Sven-Bodo Scholz. Compilation of Modelica Array Computations into Single Assignment C for Efficient Execution on CUDA-enabled GPUs. *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, EOOLT, Oslo, Norway, October 3.*, 2010.

[53] Kristian Stavåker and Peter Fritzson. Generation of Simulation Code from Equation-Based Models for Execution on CUDA-Enabled GPUs. *MCC'10 Workshop, Gothenburg, Sweden, November 18-19*, 2010.

[54] Levon Saldamli and Bernhard Bachmann and Hans-Jürg Wiesmann and Peter Fritzson. A Framework for Describing and Solving PDE Models in Modelica. *Proceedings of the 4th International Modelica Conference, Hamburg, Germany, March 7-8*, 2005.

[55] Håkan Lundvall. *Automatic Parallelization using Pipelining for Equation-Based Simulation Languages. Licentiate thesis 1381.* Linköping University, 2008.

[56] Mahder Gebremedhin and Afshin Hemmati Moghadam and Peter Fritzson and Kristian Stavåker. A Data-Parallel Algorithmic Modelica Extension for Efficient Execution on Multi-Core Platforms. In *Proceedings of the 9th International Modelica Conference (Modelica'2012), Munich, September 3-5*, 2012.

[57] Manuel Ljubijankić and Christoph Nytsch-Geusen. 3D/1D Co-Simulation von Raumlufströmungen und einer Luftheizung am Beispiel eines Thermischen Modellhauses. *Fourth German-Austrian IBPSA Conference, BauSIM 2012, Berlin University of the Arts*, 2012.

[58] Manuel Ljubijankić and Christoph Nytsch-Geusen and Jörg Rädler and Martin Löffler. Numerical Coupling of Modelica and CFD for Building Energy Supply Systems. *8th International Modelica Conference*, 2011.

[59] Martin Sjölund and Robert Braun and Peter Fritzson and Petter Krus. Towards Efficient Distributed Simulation in Modelica using Transmission Line Modeling. *3rd International Workshop on Equation-Based*

Object-Oriented Modeling Languages and Tools, EOOLT, Oslo, Norway, October 3, 2010.

[60] Martina Maggio and Kristian Stavåker and Filippo Donida and 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 (Modelica'2009), Como, Italy, September 20-22*, 2009.

[61] Mats G. Larson and Fredrik Bengzon. *The Finite Element Method - Theory, Implementation and Applications*. Springer Verlag, 2013.

[62] Matthias Korch and Thomas Rauber. Scalable Parallel RK Solvers for ODEs Derived by the Method of Lines. In Harald Kosch and László Böszörményi and Hermann Hellwagner [40], pages 830--839.

[63] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.0*. High-Performance Computing Center Stuttgart (HLRS), 2012. http://www.mpi-forum.org.

[64] Michael Hind. Changing The Foundation How the Multicore Era Has Impacted Software and What the Future Holds. IBM T.J. Watson Research Center. *ACACES 2014 Summer School July 14-18 Fiuggi Italy, [accessed April 8, 2015]*, 2014.

[65] Tom Mitchell. *Machine Learning*. McGraw Hill, 1997.

[66] Modelica and the Modelica Association, [accessed April 8, 2015]. http://www.modelica.org.

[67] Modelica Association. *FMI for Model Exchange and Co-Simulation*, v. 2.0 edition, 2014. https://www.fmi-standard.org/downloads [accessed March 26, 2015].

[68] Murray Cole. A s̈keletaläpproach to the exploitation of parallelism. In *CONPAR Conference, New York, NY, USA. Cambridge University Press.*, 1989.

[69] CUDA Zone, [accessed April 8, 2015]. http://www.nvidia.com/object/cuda_home_new.html.

[70] OpenCL, [accessed April 8, 2015]. http://www.khronos.org/opencl/.

[71] The OpenModelica Project, [accessed April 8, 2015]. http://www.openmodelica.org.

[72] OSMC. *OpenModelica User Guide, [accessed 23 September, 2011]*, 2011.

[73] Per Östlund. Simulation of Modelica Models on the CUDA Architecture. LIU-IDA/LITH-EX-A--09/062--SE. Master's thesis, Linköping University, 2009.

[74] Per Östlund and Kristian Stavåker and Peter Fritzson. Parallel Simulation of Equation-Based Models on CUDA-Enabled GPUs. *POOSC Workshop, Reno, Nevada, October 18*, 2010.

[75] Peter Kilpatrick. Introduction to Parallel Computing Concepts. *ParaPhrase International Summer School in Parallel Patterns, [accessed 8 April, 2015]*, 2014.

[76] Levon Saldamli. *PDEModelica A High-Level Language for Modeling with Partial Differential Equations*. PhD thesis, Linköping University, 2006. Dissertation No. 1016.

[77] Samir Khan. Battery Models and Discretizing PDEs for System Simulation. Adept Scientific plc. 2012.

[78] Samir Khan. Discretizing PDEs for MapleSim. Adept Scientific plc. 2012.

[79] Single Assignment C Homepage, [accessed April 8, 2015]. http://www.sac-home.org.

[80] Kristian Stavåker. *Contributions to Parallel Simulation of Equation-Based Models on Graphics Processing Units. Licentiate Thesis No. 1507*. Linköping University, 2011.

[81] SUNDIALS (SUite of Nonlinear and DIfferential/ALgebraic equation Solvers), [accessed April 8, 2015]. http://acts.nersc.gov/sundials/index.html.

[82] High Performance Computing - Supercomputing with Tesla GPUs, [accessed April 8, 2015]. http://www.nvidia.com/object/tesla_computing_solutions.html.

[83] Thomas Rauber and Gudula Rünger. Iterated Runge-Kutta Methods on Distributed Memory Multiprocessors. In *PDP*, pages 12--19. IEEE Computer Society, 1995.

[84] Thomas Rauber and Gudula Rünger. Parallel Execution of Embedded and Iterated Runge-Kutta Methods. *Concurrency - Practice and Experience*, 11(7):367--385, 1999.

[85] Usman Dastgeer and Christoph Kessler. Flexible Runtime Support for Efficient Skeleton Programming on Heterogeneous GPU-based Systems. In *Proceedings of the ParCo 2011: International Conference on Parallel Computing, Ghent, Belgium*, 2011.

[86] Usman Dastgeer and Johan Enmyren and Christoph Kessler. Auto-tuning SkePU: A Multi-Backend Skeleton Programming Framework for Multi-GPU Systems. In *Proceedings of the IWMSE-2011, Hawaii, USA, May, ACM (ACM DL). A previous version of this article was also presented at: Proc. Fourth Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG-2011), January 23, 2011, in conjunction with HiPEAC-2011 conference, Heraklion, Greece.*, 2011.

[87] Victor W Lee, et. al. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU, Intel Corporation. In *Proceedings of the 37th annual international symposium on Computer architecture, ISCA '10*, 2010.

[88] Vincent Heuveline and Eva Ketelaer and Staffan Ronnås and Mareike Schmidtobreick and Martin Wlotzka. Scalability Study of HiFlow3 based on a Fluid Flow Channel Benchmark. *8th BFG/bwGRID Workshop Proceedings*, 2012.

[89] Will Schroeder and Ken Martin and Bill Lorensen. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics.* Kiteware, Inc., 2006.

[90] Xenofon Floros and Federico Bergero and Francois Cellier and Ernesto Kofman. Automated Simulation of Modelica Models with QSS Methods - The Discontinuous Case. In *Proceedings of the 8th International Modelica Conference (Modelica'2010), Dresden, Germany, Mars 20-22*, 2011.

[91] Xenofon Floros and Francois Cellier and Ernesto Kofman. Discretizing Time or States? A Comparative Study between DASSL and QSS. *3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, EOOLT, Oslo, Norway, October 3.*, 2010.

[92] Youssef Saad. *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics (SIAM), 2000.

[93] Zhihua Li and Ling Zheng and Huili Zhang. Solving PDE Models in Modelica. *2008 International Symposium on Information Science and Engieering. ISISE 08*, 1:53--57, 2008.

Glossary

BLT Block Lower Triangular. 21, 24, 25, 51, 79

CUDA Compute Unified Device Architecture. 11, 13, 31, 32, 47, 48, 50--52, 55, 56, 63, 70, 125, 130

DAE Differential Algebraic Equation. 17--19, 21, 29, 39, 47, 48, 88--90, 125, 126, 128, 129

FMI Functional Mockup Interface. 10--12, 89, 90, 99, 101, 102, 107, 108, 128

FPGA Field Programmable Gate Arrays. 30

GPGPU General-Purpose Computing on Graphics Processing Units. 31, 124, 125, 130

GPU Graphics Processing Unit. 7, 8, 10, 11, 13, 30--32, 42, 47, 48, 53, 55, 57, 58, 60, 62, 63, 70, 73, 76, 77, 86, 124--127, 129, 130

MIMD Multiple Instruction Multiple Data. 30, 50

MISD Multiple Instruction Single Data. 30

MPI Message Passing Interface. 104, 106, 108

ODE Ordinary Differential Equation. 10, 17, 18, 21, 29, 34, 47--49, 88, 94, 125, 126, 128, 129

PDE Partial Differential Equation. 3, 8, 10--13, 19--21, 39, 88--90, 92, 96, 99--103, 105--108, 111, 124, 127--131

PELAB Programming Environments Laboratory. 7, 33, 127

PID Proportional-Integral Derivative. 89, 90, 95, 99--102, 105, 107, 108

QSS Quantized State System. 13, 40, 47--51, 53, 125--127

SAC Single Assignment C. v, 13, 63, 64, 66, 68

SIMD Single Instruction Multiple Data. 30, 43, 50, 51, 57

SISD Single Instruction Single Data. 30

TLM Transmission Line Modeling. 2, 37, 39, 127

Department of Computer and Information Science
Linköpings universitet

Dissertations

Linköping Studies in Science and Technology
Linköping Studies in Arts and Science
Linköping Studies in Statistics
Linköpings Studies in Information Science

Linköping Studies in Science and Technology

No 14 **Anders Haraldsson:** A Program Manipulation System Based on Partial Evaluation, 1977, ISBN 91-7372-144-1.

No 17 **Bengt Magnhagen:** Probability Based Verification of Time Margins in Digital Designs, 1977, ISBN 91-7372-157-3.

No 18 **Mats Cedwall**: Semantisk analys av process-beskrivningar i naturligt språk, 1977, ISBN 91- 7372-168-9.

No 22 **Jaak Urmi:** A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978, ISBN 91-7372-188-3.

No 33 **Tore Risch:** Compilation of Multiple File Queries in a Meta-Database System 1978, ISBN 91- 7372-232-4.

No 51 **Erland Jungert:** Synthesizing Database Structures from a User Oriented Data Model, 1980, ISBN 91-7372-387-8.

No 54 **Sture Hägglund:** Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980, ISBN 91-7372-404-1.

No 55 **Pär Emanuelson:** Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation, 1980, ISBN 91-7372-403-3.

No 58 **Bengt Johnsson, Bertil Andersson:** The Human-Computer Interface in Commercial Systems, 1981, ISBN 91-7372-414-9.

No 69 **H. Jan Komorowski:** A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981, ISBN 91-7372-479-3.

No 71 **René Reboh:** Knowledge Engineering Techniques and Tools for Expert Systems, 1981, ISBN 91-7372-489-0.

No 77 **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982, ISBN 91- 7372-527-7.

No 94 **Hans Lunell:** Code Generator Writing Systems, 1983, ISBN 91-7372-652-4.

No 97 **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983, ISBN 91-7372-660-5.

No 109 **Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation, 1984, ISBN 91-7372-801-2.

No 111 **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984, ISBN 91-7372- 805-5.

No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons, 1987, ISBN 91-7870-133-3.

No 165 **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning, 1987, ISBN 91-7870-183-X.

No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems, 1987, ISBN 91-7870-225-9.

No 174 **Johan Fagerström:** A Paradigm and System for Design of Distributed Systems, 1988, ISBN 91-7870-301-8.

No 192 **Dimiter Driankov:** Towards a Many Valued Logic of Quantified Belief, 1988, ISBN 91-7870-374-3.

No 213 **Lin Padgham:** Non-Monotonic Inheritance for an Object Oriented Knowledge Base, 1989, ISBN 91-7870-485-5.

No 214 **Tony Larsson:** A Formal Hardware Description and Verification Method, 1989, ISBN 91-7870-517-7.

No 221 **Michael Reinfrank:** Fundamentals and Logical Foundations of Truth Maintenance, 1989, ISBN 91-7870-546-0.

No 239 **Jonas Löwgren:** Knowledge-Based Design Support and Discourse Management in User Interface Management Systems, 1991, ISBN 91-7870-720-X.

No 244 **Henrik Eriksson:** Meta-Tool Support for Knowledge Acquisition, 1991, ISBN 91-7870-746-3.

No 252 **Peter Eklund:** An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies, 1991, ISBN 91-7870-784-6.

No 258 **Patrick Doherty:** NML3 - A Non-Monotonic Formalism with Explicit Defaults, 1991, ISBN 91-7870-816-8.

No 260 **Nahid Shahmehri:** Generalized Algorithmic Debugging, 1991, ISBN 91-7870-828-1.

No 264 **Nils Dahlbäck:** Representation of Discourse-Cognitive and Computational Aspects, 1992, ISBN 91-7870-850-8.

No 265 **Ulf Nilsson:** Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs, 1992, ISBN 91-7870-858-3.

No 270 **Ralph Rönnquist:** Theory and Practice of Tense-bound Object References, 1992, ISBN 91-7870-873-7.

No 273 **Björn Fjellborg:** Pipeline Extraction for VLSI Data Path Synthesis, 1992, ISBN 91-7870-880-X.

No 276 **Staffan Bonnier:** A Formal Basis for Horn Clause Logic with External Polymorphic Functions, 1992, ISBN 91-7870-896-6.

No 277 **Kristian Sandahl:** Developing Knowledge Management Systems with an Active Expert Methodology, 1992, ISBN 91-7870-897-4.

No 281 **Christer Bäckström:** Computational Complexity of Reasoning about Plans, 1992, ISBN 91-7870-979-2.

No 292 **Mats Wirén:** Studies in Incremental Natural Language Analysis, 1992, ISBN 91-7871-027-8.

No 297 **Mariam Kamkar:** Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.

No 302 **Tingting Zhang:** A Study in Diagnosis Using Classification and Defaults, 1993, ISBN 91-7871-078-2

No 312 **Arne Jönsson:** Dialogue Management for Natural Language Interfaces - An Empirical Approach, 1993, ISBN 91-7871-110-X.

No 338 **Simin Nadjm-Tehrani:** Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification, 1994, ISBN 91-7871-237-8.

No 371 **Bengt Savén:** Business Models for Decision Support and Learning. A Study of Discrete-Event Manufacturing Simulation at Asea/ABB 1968-1993, 1995, ISBN 91-7871-494-X.

No 375 **Ulf Söderman:** Conceptual Modelling of Mode Switching Physical Systems, 1995, ISBN 91-7871-516-4.

No 383 **Andreas Kågedal:** Exploiting Groundness in Logic Programs, 1995, ISBN 91-7871-538-5.

No 396 **George Fodor:** Ontological Control, Description, Identification and Recovery from Problematic Control Situations, 1995, ISBN 91-7871-603-9.

No 413 **Mikael Pettersson:** Compiling Natural Semantics, 1995, ISBN 91-7871-641-1.

No 414 **Xinli Gu:** RT Level Testability Improvement by Testability Analysis and Transformations, 1996, ISBN 91-7871-654-3.

No 416 **Hua Shu:** Distributed Default Reasoning, 1996, ISBN 91-7871-665-9.

No 429 **Jaime Villegas:** Simulation Supported Industrial Training from an Organisational Learning Perspective - Development and Evaluation of the SSIT Method, 1996, ISBN 91-7871-700-0.

No 431 **Peter Jonsson:** Studies in Action Planning: Algorithms and Complexity, 1996, ISBN 91-7871-704-3.

No 437 **Johan Boye:** Directional Types in Logic Programming, 1996, ISBN 91-7871-725-6.

No 439 **Cecilia Sjöberg:** Activities, Voices and Arenas: Participatory Design in Practice, 1996, ISBN 91-7871-728-0.

No 448 **Patrick Lambrix:** Part-Whole Reasoning in Description Logics, 1996, ISBN 91-7871-820-1.

No 452 **Kjell Orsborn:** On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications, 1996, ISBN 91-7871-827-9.

No 459 **Olof Johansson:** Development Environments for Complex Product Models, 1996, ISBN 91-7871-855-4.

No 461 **Lena Strömbäck:** User-Defined Constructions in Unification-Based Formalisms, 1997, ISBN 91-7871-857-0.

No 462 **Lars Degerstedt:** Tabulation-based Logic Programming: A Multi-Level View of Query Answering, 1996, ISBN 91-7871-858-9.

No 475 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av hur ekonomiska styrsystem utformas och används efter företagsförvärv, 1997, ISBN 91-7871-914-3.

No 480 **Mikael Lindvall:** An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution, 1997, ISBN 91-7871-927-5.

No 485 **Göran Forslund:** Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997, ISBN 91-7871-938-0.

No 494 **Martin Sköld:** Active Database Management Systems for Monitoring and Control, 1997, ISBN 91-7219-002-7.

No 495 **Hans Olsén:** Automatic Verification of Petri Nets in a CLP framework, 1997, ISBN 91-7219-011-6.

No 498 **Thomas Drakengren:** Algorithms and Complexity for Temporal and Spatial Formalisms, 1997, ISBN 91-7219-019-1.

No 502 **Jakob Axelsson:** Analysis and Synthesis of Heterogeneous Real-Time Systems, 1997, ISBN 91-7219-035-3.

No 503 **Johan Ringström:** Compiler Generation for Data-Parallel Programming Languages from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.

No 512 **Anna Moberg:** Närhet och distans - Studier av kommunikationsmönster i satellitkontor och flexibla kontor, 1997, ISBN 91-7219-119-8.

No 520 **Mikael Ronström:** Design and Modelling of a Parallel Data Server for Telecom Applications, 1998, ISBN 91-7219-169-4.

No 522 **Niclas Ohlsson:** Towards Effective Fault Prevention - An Empirical Study in Software Engineering, 1998, ISBN 91-7219-176-7.

No 526 **Joachim Karlsson:** A Systematic Approach for Prioritizing Software Requirements, 1998, ISBN 91-7219-184-8.

No 530 **Henrik Nilsson:** Declarative Debugging for Lazy Functional Languages, 1998, ISBN 91-7219-197-x.

No 555 **Jonas Hallberg:** Timing Issues in High-Level Synthesis, 1998, ISBN 91-7219-369-7.

No 561 **Ling Lin:** Management of 1-D Sequence Data - From Discrete to Continuous, 1999, ISBN 91-7219-402-2.

No 563 **Eva L Ragnemalm:** Student Modelling based on Collaborative Dialogue with a Learning Companion, 1999, ISBN 91-7219-412-X.

No 567 **Jörgen Lindström:** Does Distance matter? On geographical dispersion in organisations, 1999, ISBN 91-7219-439-1.

No 582 **Vanja Josifovski:** Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration, 1999, ISBN 91-7219-482-0.

No 589 **Rita Kovordányi:** Modeling and Simulating Inhibitory Mechanisms in Mental Image Reinterpretation - Towards Cooperative Human-Computer Creativity, 1999, ISBN 91-7219-506-1.

No 592 **Mikael Ericsson:** Supporting the Use of Design Knowledge - An Assessment of Commenting Agents, 1999, ISBN 91-7219-532-0.

No 593 **Lars Karlsson:** Actions, Interactions and Narratives, 1999, ISBN 91-7219-534-7.

No 594 **C. G. Mikael Johansson:** Social and Organizational Aspects of Requirements Engineering Methods - A practice-oriented approach, 1999, ISBN 91-7219-541-X.

No 595 **Jörgen Hansson:** Value-Driven Multi-Class Overload Management in Real-Time Database Systems, 1999, ISBN 91-7219-542-8.

No 596 **Niklas Hallberg:** Incorporating User Values in the Design of Information Systems and Services in the Public Sector: A Methods Approach, 1999, ISBN 91-7219-543-6.

No 597 **Vivian Vimarlund:** An Economic Perspective on the Analysis of Impacts of Information Technology: From Case Studies in Health-Care towards General Models and Theories, 1999, ISBN 91-7219-544-4.

No 598 **Johan Jenvald:** Methods and Tools in Computer-Supported Taskforce Training, 1999, ISBN 91-7219-547-9.

No 607 **Magnus Merkel:** Understanding and enhancing translation by parallel text processing, 1999, ISBN 91-7219-614-9.

No 611 **Silvia Coradeschi:** Anchoring symbols to sensory data, 1999, ISBN 91-7219-623-8.

No 613 **Man Lin:** Analysis and Synthesis of Reactive Systems: A Generic Layered Architecture Perspective, 1999, ISBN 91-7219-630-0.

No 618 **Jimmy Tjäder:** Systemimplementering i praktiken - En studie av logiker i fyra projekt, 1999, ISBN 91-7219-657-2.

No 627 **Vadim Engelson:** Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing, 2000, ISBN 91-7219-709-9.

No 637 **Esa Falkenroth:** Database Technology for Control and Simulation, 2000, ISBN 91-7219-766-8.

No 639 **Per-Arne Persson:** Bringing Power and Knowledge Together: Information Systems Design for Autonomy and Control in Command Work, 2000, ISBN 91-7219-796-X.

No 660 **Erik Larsson:** An Integrated System-Level Design for Testability Methodology, 2000, ISBN 91-7219-890-7.

No 688 **Marcus Bjäreland:** Model-based Execution Monitoring, 2001, ISBN 91-7373-016-5.

No 689 **Joakim Gustafsson:** Extending Temporal Action Logic, 2001, ISBN 91-7373-017-3.

No 720 **Carl-Johan Petri:** Organizational Information Provision - Managing Mandatory and Discretionary Use of Information Technology, 2001, ISBN-91-7373-126-9.

No 724 **Paul Scerri:** Designing Agents for Systems with Adjustable Autonomy, 2001, ISBN 91 7373 207 9.

No 725 **Tim Heyer:** Semantic Inspection of Software Artifacts: From Theory to Practice, 2001, ISBN 91 7373 208 7.

No 726 **Pär Carlshamre:** A Usability Perspective on Requirements Engineering - From Methodology to Product Development, 2001, ISBN 91 7373 212 5.

No 732 **Juha Takkinen:** From Information Management to Task Management in Electronic Mail, 2002, ISBN 91 7373 258 3.

No 745 **Johan Åberg:** Live Help Systems: An Approach to Intelligent Help for Web Information Systems, 2002, ISBN 91-7373-311-3.

No 746 **Rego Granlund:** Monitoring Distributed Teamwork Training, 2002, ISBN 91-7373-312-1.

No 757 **Henrik André-Jönsson:** Indexing Strategies for Time Series Data, 2002, ISBN 917373-346-6.

No 747 **Anneli Hagdahl:** Development of IT-supported Interorganisational Collaboration - A Case Study in the Swedish Public Sector, 2002, ISBN 91-7373-314-8.

No 749 **Sofie Pilemalm:** Information Technology for Non-Profit Organisations - Extended Participatory Design of an Information System for Trade Union Shop Stewards, 2002, ISBN 91-7373-318-0.

No 765 **Stefan Holmlid:** Adapting users: Towards a theory of use quality, 2002, ISBN 91-7373-397-0.

No 771 **Magnus Morin:** Multimedia Representations of Distributed Tactical Operations, 2002, ISBN 91-7373-421-7.

No 772 **Pawel Pietrzak:** A Type-Based Framework for Locating Errors in Constraint Logic Programs, 2002, ISBN 91-7373-422-5.

No 758 **Erik Berglund:** Library Communication Among Programmers Worldwide, 2002, ISBN 91-7373-349-0.

No 774 **Choong-ho Yi:** Modelling Object-Oriented Dynamic Systems Using a Logic-Based Framework, 2002, ISBN 91-7373-424-1.

No 779 **Mathias Broxvall:** A Study in the Computational Complexity of Temporal Reasoning, 2002, ISBN 91-7373-440-3.

No 793 **Asmus Pandikow:** A Generic Principle for Enabling Interoperability of Structured and Object-Oriented Analysis and Design Tools, 2002, ISBN 91-7373-479-9.

No 785 **Lars Hult:** Publika Informationstjänster. En studie av den Internetbaserade encyklopedins bruksegenskaper, 2003, ISBN 91-7373-461-6.

No 800 **Lars Taxén:** A Framework for the Coordination of Complex Systems´ Development, 2003, ISBN 91-7373-604-X

No 808 **Klas Gäre:** Tre perspektiv på förväntningar och förändringar i samband med införande av informationssystem, 2003, ISBN 91-7373-618-X.

No 821 **Mikael Kindborg:** Concurrent Comics - programming of social agents by children, 2003, ISBN 91-7373-651-1.

No 823 **Christina Ölvingson:** On Development of Information Systems with GIS Functionality in Public Health Informatics: A Requirements Engineering Approach, 2003, ISBN 91-7373-656-2.

No 828 **Tobias Ritzau:** Memory Efficient Hard Real-Time Garbage Collection, 2003, ISBN 91-7373-666-X.

No 833 **Paul Pop:** Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems, 2003, ISBN 91-7373-683-X.

No 852 **Johan Moe:** Observing the Dynamic Behaviour of Large Distributed Systems to Improve Development and Testing – An Empirical Study in Software Engineering, 2003, ISBN 91-7373-779-8.

No 867 **Erik Herzog:** An Approach to Systems Engineering Tool Data Representation and Exchange, 2004, ISBN 91-7373-929-4.

No 872 **Aseel Berglund:** Augmenting the Remote Control: Studies in Complex Information Navigation for Digital TV, 2004, ISBN 91-7373-940-5.

No 869 **Jo Skåmedal:** Telecommuting's Implications on Travel and Travel Patterns, 2004, ISBN 91-7373-935-9.

No 870 **Linda Askenäs:** The Roles of IT - Studies of Organising when Implementing and Using Enterprise Systems, 2004, ISBN 91-7373-936-7.

No 874 **Annika Flycht-Eriksson:** Design and Use of Ontologies in Information-Providing Dialogue Systems, 2004, ISBN 91-7373-947-2.

No 873 **Peter Bunus:** Debugging Techniques for Equation-Based Languages, 2004, ISBN 91-7373-941-3.

No 876 **Jonas Mellin:** Resource-Predictable and Efficient Monitoring of Events, 2004, ISBN 91-7373-956-1.

No 883 **Magnus Bång:** Computing at the Speed of Paper: Ubiquitous Computing Environments for Healthcare Professionals, 2004, ISBN 91-7373-971-5

No 882 **Robert Eklund:** Disfluency in Swedish human-human and human-machine travel booking dialogues, 2004, ISBN 91-7373-966-9.

No 887 **Anders Lindström:** English and other Foreign Linguistic Elements in Spoken Swedish. Studies of Productive Processes and their Modelling using Finite-State Tools, 2004, ISBN 91-7373-981-2.

No 889 **Zhiping Wang:** Capacity-Constrained Production-inventory systems - Modelling and Analysis in both a traditional and an e-business context, 2004, ISBN 91-85295-08-6.

No 893 **Pernilla Qvarfordt:** Eyes on Multimodal Interaction, 2004, ISBN 91-85295-30-2.

No 910 **Magnus Kald:** In the Borderland between Strategy and Management Control - Theoretical Framework and Empirical Evidence, 2004, ISBN 91-85295-82-5.

No 918 **Jonas Lundberg:** Shaping Electronic News: Genre Perspectives on Interaction Design, 2004, ISBN 91-85297-14-3.

No 900 **Mattias Arvola:** Shades of use: The dynamics of interaction design for sociable use, 2004, ISBN 91-85295-42-6.

No 920 **Luis Alejandro Cortés:** Verification and Scheduling Techniques for Real-Time Embedded Systems, 2004, ISBN 91-85297-21-6.

No 929 **Diana Szentivanyi:** Performance Studies of Fault-Tolerant Middleware, 2005, ISBN 91-85297-58-5.

No 933 **Mikael Cäker:** Management Accounting as Constructing and Opposing Customer Focus: Three Case Studies on Management Accounting and Customer Relations, 2005, ISBN 91-85297-64-X.

No 937 **Jonas Kvarnström:** TALplanner and Other Extensions to Temporal Action Logic, 2005, ISBN 91-85297-75-5.

No 938 **Bourhane Kadmiry:** Fuzzy Gain-Scheduled Visual Servoing for Unmanned Helicopter, 2005, ISBN 91-85297-76-3.

No 945 **Gert Jervan:** Hybrid Built-In Self-Test and Test Generation Techniques for Digital Systems, 2005, ISBN: 91-85297-97-6.

No 946 **Anders Arpteg:** Intelligent Semi-Structured Information Extraction, 2005, ISBN 91-85297-98-4.

No 947 **Ola Angelsmark:** Constructing Algorithms for Constraint Satisfaction and Related Problems - Methods and Applications, 2005, ISBN 91-85297-99-2.

No 963 **Calin Curescu:** Utility-based Optimisation of Resource Allocation for Wireless Networks, 2005, ISBN 91-85457-07-8.

No 972 **Björn Johansson:** Joint Control in Dynamic Situations, 2005, ISBN 91-85457-31-0.

No 974 **Dan Lawesson:** An Approach to Diagnosability Analysis for Interacting Finite State Systems, 2005, ISBN 91-85457-39-6.

No 979 **Claudiu Duma:** Security and Trust Mechanisms for Groups in Distributed Services, 2005, ISBN 91-85457-54-X.

No 983 **Sorin Manolache:** Analysis and Optimisation of Real-Time Systems with Stochastic Behaviour, 2005, ISBN 91-85457-60-4.

No 986 **Yuxiao Zhao:** Standards-Based Application Integration for Business-to-Business Communications, 2005, ISBN 91-85457-66-3.

No 1004 **Patrik Haslum:** Admissible Heuristics for Automated Planning, 2006, ISBN 91-85497-28-2.

No 1005 **Aleksandra Tešanovic:** Developing Reusable and Reconfigurable Real-Time Software using Aspects and Components, 2006, ISBN 91-85497-29-0.

No 1008 **David Dinka:** Role, Identity and Work: Extending the design and development agenda, 2006, ISBN 91-85497-42-8.

No 1009 **Iakov Nakhimovski:** Contributions to the Modeling and Simulation of Mechanical Systems with Detailed Contact Analysis, 2006, ISBN 91-85497-43-X.

No 1013 **Wilhelm Dahllöf:** Exact Algorithms for Exact Satisfiability Problems, 2006, ISBN 91-85523-97-6.

No 1016 **Levon Saldamli:** PDEModelica - A High-Level Language for Modeling with Partial Differential Equations, 2006, ISBN 91-85523-84-4.

No 1017 **Daniel Karlsson:** Verification of Component-based Embedded System Designs, 2006, ISBN 91-85523-79-8

No 1018 **Ioan Chisalita:** Communication and Networking Techniques for Traffic Safety Systems, 2006, ISBN 91-85523-77-1.

No 1019 **Tarja Susi:** The Puzzle of Social Activity - The Significance of Tools in Cognition and Cooperation, 2006, ISBN 91-85523-71-2.

No 1021 **Andrzej Bednarski:** Integrated Optimal Code Generation for Digital Signal Processors, 2006, ISBN 91-85523-69-0.

No 1022 **Peter Aronsson:** Automatic Parallelization of Equation-Based Simulation Programs, 2006, ISBN 91-85523-68-2.

No 1030 **Robert Nilsson:** A Mutation-based Framework for Automated Testing of Timeliness, 2006, ISBN 91-85523-35-6.

No 1034 **Jon Edvardsson:** Techniques for Automatic Generation of Tests from Programs and Specifications, 2006, ISBN 91-85523-31-3.

No 1035 **Vaida Jakoniene:** Integration of Biological Data, 2006, ISBN 91-85523-28-3.

No 1045 **Genevieve Gorrell:** Generalized Hebbian Algorithms for Dimensionality Reduction in Natural Language Processing, 2006, ISBN 91-85643-88-2.

No 1051 **Yu-Hsing Huang:** Having a New Pair of Glasses - Applying Systemic Accident Models on Road Safety, 2006, ISBN 91-85643-64-5.

No 1054 **Åsa Hedenskog:** Perceive those things which cannot be seen - A Cognitive Systems Engineering perspective on requirements management, 2006, ISBN 91-85643-57-2.

No 1061 **Cécile Åberg:** An Evaluation Platform for Semantic Web Technology, 2007, ISBN 91-85643-31-9.

No 1073 **Mats Grindal:** Handling Combinatorial Explosion in Software Testing, 2007, ISBN 978-91-85715-74-9.

No 1075 **Almut Herzog:** Usable Security Policies for Runtime Environments, 2007, ISBN 978-91-85715-65-7.

No 1079 **Magnus Wahlström:** Algorithms, measures, and upper bounds for Satisfiability and related problems, 2007, ISBN 978-91-85715-55-8.

No 1083 **Jesper Andersson:** Dynamic Software Architectures, 2007, ISBN 978-91-85715-46-6.

No 1086 **Ulf Johansson:** Obtaining Accurate and Comprehensible Data Mining Models - An Evolutionary Approach, 2007, ISBN 978-91-85715-34-3.

No 1089 **Traian Pop:** Analysis and Optimisation of Distributed Embedded Systems with Heterogeneous Scheduling Policies, 2007, ISBN 978-91-85715-27-5.

No 1091 **Gustav Nordh:** Complexity Dichotomies for CSP-related Problems, 2007, ISBN 978-91-85715-20-6.

No 1106 **Per Ola Kristensson:** Discrete and Continuous Shape Writing for Text Entry and Control, 2007, ISBN 978-91-85831-77-7.

No 1110 **He Tan:** Aligning Biomedical Ontologies, 2007, ISBN 978-91-85831-56-2.

No 1112 **Jessica Lindblom:** Minding the body - Interacting socially through embodied action, 2007, ISBN 978-91-85831-48-7.

No 1113 **Pontus Wärnestål:** Dialogue Behavior Management in Conversational Recommender Systems, 2007, ISBN 978-91-85831-47-0.

No 1120 **Thomas Gustafsson:** Management of Real-Time Data Consistency and Transient Overloads in Embedded Systems, 2007, ISBN 978-91-85831-33-3.

No 1127 **Alexandru Andrei:** Energy Efficient and Predictable Design of Real-time Embedded Systems, 2007, ISBN 978-91-85831-06-7.

No 1139 **Per Wikberg:** Eliciting Knowledge from Experts in Modeling of Complex Systems: Managing Variation and Interactions, 2007, ISBN 978-91-85895-66-3.

No 1143 **Mehdi Amirijoo:** QoS Control of Real-Time Data Services under Uncertain Workload, 2007, ISBN 978-91-85895-49-6.

No 1150 **Sanny Syberfeldt:** Optimistic Replication with Forward Conflict Resolution in Distributed Real-Time Databases, 2007, ISBN 978-91-85895-27-4.

No 1155 **Beatrice Alenljung:** Envisioning a Future Decision Support System for Requirements Engineering - A Holistic and Human-centred Perspective, 2008, ISBN 978-91-85895-11-3.

No 1156 **Artur Wilk:** Types for XML with Application to Xcerpt, 2008, ISBN 978-91-85895-08-3.

No 1183 **Adrian Pop:** Integrated Model-Driven Development Environments for Equation-Based Object-Oriented Languages, 2008, ISBN 978-91-7393-895-2.

No 1185 **Jörgen Skågeby:** Gifting Technologies - Ethnographic Studies of End-users and Social Media Sharing, 2008, ISBN 978-91-7393-892-1.

No 1187 **Imad-Eldin Ali Abugessaisa:** Analytical tools and information-sharing methods supporting road safety organizations, 2008, ISBN 978-91-7393-887-7.

No 1204 **H. Joe Steinhauer:** A Representation Scheme for Description and Reconstruction of Object Configurations Based on Qualitative Relations, 2008, ISBN 978-91-7393-823-5.

No 1222 **Anders Larsson:** Test Optimization for Core-based System-on-Chip, 2008, ISBN 978-91-7393-768-9.

No 1238 **Andreas Borg:** Processes and Models for Capacity Requirements in Telecommunication Systems, 2009, ISBN 978-91-7393-700-9.

No 1240 **Fredrik Heintz:** DyKnow: A Stream-Based Knowledge Processing Middleware Framework, 2009, ISBN 978-91-7393-696-5.

No 1241 **Birgitta Lindström:** Testability of Dynamic Real-Time Systems, 2009, ISBN 978-91-7393-695-8.

No 1244 **Eva Blomqvist:** Semi-automatic Ontology Construction based on Patterns, 2009, ISBN 978-91-7393-683-5.

No 1249 **Rogier Woltjer:** Functional Modeling of Constraint Management in Aviation Safety and Command and Control, 2009, ISBN 978-91-7393-659-0.

No 1260 **Gianpaolo Conte:** Vision-Based Localization and Guidance for Unmanned Aerial Vehicles, 2009, ISBN 978-91-7393-603-3.

No 1262 **AnnMarie Ericsson:** Enabling Tool Support for Formal Analysis of ECA Rules, 2009, ISBN 978-91-7393-598-2.

No 1266 **Jiri Trnka:** Exploring Tactical Command and Control: A Role-Playing Simulation Approach, 2009, ISBN 978-91-7393-571-5.

No 1268 **Bahlol Rahimi:** Supporting Collaborative Work through ICT - How End-users Think of and Adopt Integrated Health Information Systems, 2009, ISBN 978-91-7393-550-0.

No 1274 **Fredrik Kuivinen:** Algorithms and Hardness Results for Some Valued CSPs, 2009, ISBN 978-91-7393-525-8.

No 1281 **Gunnar Mathiason:** Virtual Full Replication for Scalable Distributed Real-Time Databases, 2009, ISBN 978-91-7393-503-6.

No 1290 **Viacheslav Izosimov:** Scheduling and Optimization of Fault-Tolerant Distributed Embedded Systems, 2009, ISBN 978-91-7393-482-4.

No 1294 **Johan Thapper:** Aspects of a Constraint Optimisation Problem, 2010, ISBN 978-91-7393-464-0.

No 1306 **Susanna Nilsson:** Augmentation in the Wild: User Centered Development and Evaluation of Augmented Reality Applications, 2010, ISBN 978-91-7393-416-9.

No 1313 **Christer Thörn:** On the Quality of Feature Models, 2010, ISBN 978-91-7393-394-0.

No 1321 **Zhiyuan He:** Temperature Aware and Defect-Probability Driven Test Scheduling for System-on-Chip, 2010, ISBN 978-91-7393-378-0.

No 1333 **David Broman:** Meta-Languages and Semantics for Equation-Based Modeling and Simulation, 2010, ISBN 978-91-7393-335-3.

No 1337 **Alexander Siemers:** Contributions to Modelling and Visualisation of Multibody Systems Simulations with Detailed Contact Analysis, 2010, ISBN 978-91-7393-317-9.

No 1354 **Mikael Asplund:** Disconnected Discoveries: Availability Studies in Partitioned Networks, 2010, ISBN 978-91-7393-278-3.

No 1359 **Jana Rambusch**: Mind Games Extended: Understanding Gameplay as Situated Activity, 2010, ISBN 978-91-7393-252-3.

No 1373 **Sonia Sangari**: Head Movement Correlates to Focus Assignment in Swedish,2011,ISBN 978-91-7393-154-0.

No 1374 **Jan-Erik Källhammer**: Using False Alarms when Developing Automotive Active Safety Systems, 2011, ISBN 978-91-7393-153-3.

No 1375 **Mattias Eriksson**: Integrated Code Generation, 2011, ISBN 978-91-7393-147-2.

No 1381 **Ola Leifler**: Affordances and Constraints of Intelligent Decision Support for Military Command and Control – Three Case Studies of Support Systems, 2011, ISBN 978-91-7393-133-5.

No 1386 **Soheil Samii**: Quality-Driven Synthesis and Optimization of Embedded Control Systems, 2011, ISBN 978-91-7393-102-1.

No 1419 **Erik Kuiper**: Geographic Routing in Intermittently-connected Mobile Ad Hoc Networks: Algorithms and Performance Models, 2012, ISBN 978-91-7519-981-8.

No 1451 **Sara Stymne**: Text Harmonization Strategies for Phrase-Based Statistical Machine Translation, 2012, ISBN 978-91-7519-887-3.

No 1455 **Alberto Montebelli**: Modeling the Role of Energy Management in Embodied Cognition, 2012, ISBN 978-91-7519-882-8.

No 1465 **Mohammad Saifullah**: Biologically-Based Interactive Neural Network Models for Visual Attention and Object Recognition, 2012, ISBN 978-91-7519-838-5.

No 1490 **Tomas Bengtsson**: Testing and Logic Optimization Techniques for Systems on Chip, 2012, ISBN 978-91-7519-742-5.

No 1481 **David Byers**: Improving Software Security by Preventing Known Vulnerabilities, 2012, ISBN 978-91-7519-784-5.

No 1496 **Tommy Färnqvist**: Exploiting Structure in CSP-related Problems, 2013, ISBN 978-91-7519-711-1.

No 1503 **John Wilander**: Contributions to Specification, Implementation, and Execution of Secure Software, 2013, ISBN 978-91-7519-681-7.

No 1506 **Magnus Ingmarsson**: Creating and Enabling the Useful Service Discovery Experience, 2013, ISBN 978-91-7519-662-6.

No 1547 **Wladimir Schamai**: Model-Based Verification of Dynamic System Behavior against Requirements: Method, Language, and Tool, 2013, ISBN 978-91-7519-505-6.

No 1551 **Henrik Svensson**: Simulations, 2013, ISBN 978-91-7519-491-2.

No 1559 **Sergiu Rafiliu**: Stability of Adaptive Distributed Real-Time Systems with Dynamic Resource Management, 2013, ISBN 978-91-7519-471-4.

No 1581 **Usman Dastgeer**: Performance-aware Component Composition for GPU-based Systems, 2014, ISBN 978-91-7519-383-0.

No 1602 **Cai Li**: Reinforcement Learning of Locomotion based on Central Pattern Generators, 2014, ISBN 978-91-7519-313-7.

No 1652 **Roland Samlaus**: An Integrated Development Environment with Enhanced Domain-Specific Interactive Model Validation, 2015, ISBN 978-91-7519-090-7.

No 1663 **Hannes Uppman**: On Some Combinatorial Optimization Problems: Algorithms and Complexity, 2015, ISBN 978-91-7519-072-3.

No 1664 **Martin Sjölund**: Tools and Methods for Analysis, Debugging, and Performance Improvement of Equation-Based Models, 2015, ISBN 978-91-7519-071-6.

No 1666 **Kristian Stavåker**: Contributions to Simulation of Modelica Models on Data-Parallel Multi-Core Architectures, 2015, ISBN 978-91-7519-068-6.

Linköping Studies in Arts and Science

No 504 **Ing-Marie Jonsson:** Social and Emotional Characteristics of Speech-based In-Vehicle Information Systems: Impact on Attitude and Driving Behaviour, 2009, ISBN 978-91-7393-478-7.

No 586 **Fabian Segelström:** Stakeholder Engagement for Service Design: How service designers identify and communicate insights, 2013, ISBN 978-91-7519-554-4.

No 618 **Johan Blomkvist:** Representing Future Situations of Service: Prototyping in Service Design, 2014, ISBN 978-91-7519-343-4.

No 620 **Marcus Mast:** Human-Robot Interaction for Semi-Autonomous Assistive Robots, 2014, ISBN 978-91-7519-319-9.

Linköping Studies in Statistics

No 9 **Davood Shahsavani:** Computer Experiments Designed to Explore and Approximate Complex Deterministic Models, 2008, ISBN 978-91-7393-976-8.

No 10 **Karl Wahlin:** Roadmap for Trend Detection and Assessment of Data Quality, 2008, ISBN 978-91-7393-792-4.

No 11 **Oleg Sysoev:** Monotonic regression for large multivariate datasets, 2010, ISBN 978-91-7393-412-1.

No 13 **Agné Burauskaite-Harju:** Characterizing Temporal Change and Inter-Site Correlations in Daily and Subdaily Precipitation Extremes, 2011, ISBN 978-91-7393-110-6.

Linköping Studies in Information Science

No 1 **Karin Axelsson:** Metodisk systemstrukturering- att skapa samstämmighet mellan informationssystemarkitektur och verksamhet, 1998. ISBN-9172-19-296-8.

No 2 **Stefan Cronholm:** Metodverktyg och användbarhet - en studie av datorstödd metodbaserad systemutveckling, 1998, ISBN-9172-19-299-2.

No 3 **Anders Avdic:** Användare och utvecklare - om anveckling med kalkylprogram, 1999. ISBN-91-7219-606-8.

No 4 **Owen Eriksson:** Kommunikationskvalitet hos informationssystem och affärsprocesser, 2000, ISBN 91-7219-811-7.

No 5 **Mikael Lind:** Från system till process - kriterier för processbestämning vid verksamhetsanalys, 2001, ISBN 91-7373-067-X.

No 6 **Ulf Melin:** Koordination och informationssystem i företag och nätverk, 2002, ISBN 91-7373-278-8.

No 7 **Pär J. Ågerfalk:** Information Systems Actability - Understanding Information Technology as a Tool for Business Action and Communication, 2003, ISBN 91-7373-628-7.

No 8 **Ulf Seigerroth:** Att förstå och förändra systemutvecklingsverksamheter - en taxonomi för metautveckling, 2003, ISBN91-7373-736-4.

No 9 **Karin Hedström:** Spår av datoriseringens värden – Effekter av IT i äldreomsorg, 2004, ISBN 91-7373-963-4.

No 10 **Ewa Braf:** Knowledge Demanded for Action - Studies on Knowledge Mediation in Organisations, 2004, ISBN 91-85295-47-7.

No 11 **Fredrik Karlsson:** Method Configuration method and computerized tool support, 2005, ISBN 91-85297-48-8.

No 12 **Malin Nordström:** Styrbar systemförvaltning - Att organisera systemförvaltningsverksamhet med hjälp av effektiva förvaltningsobjekt, 2005, ISBN 91-85297-60-7.

No 13 **Stefan Holgersson:** Yrke: POLIS - Yrkeskunskap, motivation, IT-system och andra förutsättningar för polisarbete, 2005, ISBN 91-85299-43-X.

No 14 **Benneth Christiansson, Marie-Therese Christiansson:** Mötet mellan process och komponent - mot ett ramverk för en verksamhetsnära kravspecifikation vid anskaffning av komponentbaserade informationssystem, 2006, ISBN 91-85643-22-X.