Simulating Partial Differential Equations using the Explicit Parallelism of ParModelica

by

Gustaf Thorslund

LIU-IDA/LITH-EX-A-15/032–SE

2015-06-30
Final thesis

Simulating Partial Differential Equations using the Explicit Parallelism of ParModelica

by

Gustaf Thorslund

LIU-IDA/LITH-EX-A–15/032–SE

2015-06-30

Technical supervisor: Mahder Gebremedhin
Supervisor: Lena Buffoni
Examiner: Peter Fritzson
Abstract

The Modelica language is a modelling and programming language for modelling cyber-physical systems using equations and algorithms. In this thesis two suggested extensions of the Modelica language are covered. Those are Partial Differential Equations (PDE) and explicit parallelism in algorithmic code. While PDEs are not yet supported by the Modelica language, this thesis presents a framework for solving PDEs using the algorithmic part of the Modelica language, including parallel extensions. Different numerical solvers have been implemented using the explicit parallel constructs suggested for Modelica by the ParModelica language extensions, and implemented as part of OpenModelica. The solvers have been evaluated using different models, and it can be seen how bigger models are suitable for a parallel solver. The intention has been to write a framework suitable for modelling and parallel simulation of PDEs. This work can, however, also be seen as a case study of how to write a custom solver using parallel algorithmic Modelica and how to evaluate the performance of a parallel solver.
# Contents

## 4 Methods

4.1 Prestudy .................................................. 21
4.1.1 Partial Differential Equation (PDE) Applications ..... 21
4.1.2 PDEs in Modelica ..................................... 21
4.1.3 Numerical Solvers Suitable for PDEs ................. 22
4.1.4 Algorithmic Modelica and ParModelica ............. 22
4.2 Design ............................................... 22
4.3 Implementation ....................................... 23
4.4 First Use Case — Heat in Plane ...................... 23
4.5 Second Use Case — Heat in Rod ..................... 24

## 5 Results

5.1 Prestudy ................................................ 25
5.1.1 PDE Applications .................................... 25
5.1.2 PDEs in Modelica .................................... 26
5.1.3 Numerical Solvers Suitable for PDEs ............... 26
5.1.4 Algorithmic Modelica and ParModelica .......... 27
5.2 Design ............................................... 28
5.2.1 User Defined State Derivative Function ........... 28
5.2.2 Representation of Types for Solver ............... 29
5.2.3 How to Optimise Scalability ..................... 29
5.3 Implementation ...................................... 31
5.3.1 User Defined State Derivative and Settings ........ 32
5.3.2 Types Used by the Solver .......................... 33
5.3.3 Solvers .......................................... 33
5.4 First Use Case — Heat in Plane .................... 34
5.4.1 Poor Insulation and Constant Temperature .... 34
5.4.2 Perfect Insulation .................................. 34
5.4.3 Heat Added or Removed at Constant Rate Depending on Location .......................... 35
5.4.4 Poor Insulation with Different Temperature Outside Depending on Location along the Boundary .... 35
5.4.5 Constant Temperature Depending on Location .... 37
5.4.6 Constant Temperature Depending on Location Combined with Periodic Boundary .................. 37
5.5 Second Use Case — Heat in Rod ..................... 39

## 6 Discussion

6.1 Results ................................................ 41
6.1.1 Prestudy .......................................... 41
6.1.2 Design .......................................... 41
6.1.3 Implementation .................................... 41
6.1.4 Use Cases ......................................... 42
6.2 Method ............................................... 42
6.2.1 Prestudy .......................................... 42
6.2.2 Design .......................................... 42
Acronyms

DAE  Differential Algebraic Equation. 10

GPGPU  General Purpose Graphic Processing Unit. 15

GPU  Graphic Processing Unit. 18

NUMA  Non-Uniform Memory Access. 18

ODE  Ordinary Differential Equation. 9, 13, 15

OMC  Open Modelica Compiler. 6

PDE  Partial Differential Equation. vi, 1, 2, 5, 6, 11, 13, 21–23, 25, 26, 33, 41–43, 45–47

SIMD  Single Instruction Multiple Data. 18, 46

SM  Streaming Multiprocessor. 18

SMP  Symmetric Multiprocessing. 17

UMA  Uniform Memory Access. 17
Chapter 1

Introduction

This chapter will first introduce the concept of modelling physical systems. Furthermore, the motivation and limitations for this master thesis are discussed. Finally, an introduction to the results is given. Chapter 2 will give a background of the tools and previous research. In Chapter 3 mathematics and parallel computing are covered. Methods, results, and discussion of results are covered by Chapters 4 to 6. This thesis is finally summarised in Chapter 7.

1.1 Motivation

To understand the behaviour of a system, it is desirable to write down known relations of the system as equations. Together the equations will form a model of the system. If the equations contain derivatives with respect to one variable, they describe an ordinary differential equation. If, however, the equations contain derivatives with respect to more than one variable, they describe a partial differential equation.

Modelica\(^1\) is an object oriented language\(^2\) for modelling complex physical systems using equations. The model can then be simulated using a numerical solver. However, Modelica does not currently support modelling partial differential equations. There are suggested extensions for PDEs in [15, 4].

OpenModelica\(^3\) is an open source\(^4\) implementation of the Modelica language, and an active research area. To have an open source implementation of the Modelica language is of benefit for the whole Modelica community. During research, language extensions have been implemented using OpenModelica. The first was PDEModelica[15], implementing support for PDEs.

\(^1\)http://www.modelica.org/ accessed June 2015
\(^2\)The Modelica language is an open standard and can be downloaded for free. There is also a book[4] available with many examples of how to use the language.
\(^3\)http://www.openmodelica.org/ accessed June 2015
\(^4\)http://opensource.org/ accessed June 2015
The second was ParModelica [5], with support for explicit parallelism in algorithms. Finally there is MetaModelica, with support for modelling programming languages, and used for implementing OpenModelica. Among the extensions, MetaModelica is actively used in the development of OpenModelica, while ParModelica has been revived during this thesis. PDEModelica has not been maintained for a while, so here it has only been used as reference.

Given that a PDE can describe a model in several dimensions, the required computations can grow exponentially with the size of the model. For this reason it is desirable to run as much as possible of the simulation in parallel.

This thesis aims to bring the concept of PDEs back under the Modelica radar, and to see how efficiently PDEs can be simulated using the explicit parallelism in ParModelica.

1.2 Goal

To be able to simulate a PDE with a solver written in the algorithmic subset of Modelica, and the parallel extension provided by ParModelica. Add a control system to the simulation, and show how the output varies/is controlled.

1.3 Limitations

The aim of this master thesis is only to implement a solver for PDEs. Due to it not being fully integrated into the OpenModelica compiler and runtime, the user will need to write a small program to describe the model. Care has been taken to make this as easy as possible. Section 5.3.1 gives a short example, and Sections 5.4 and 5.5 present use cases covering two and three dimensional models together with different boundary conditions.

While the models used in the examples and use cases intend to model a physical system, the main intent is to show the necessary computations. For this reason they are made as simple as possible, while still keeping the possible computational complexity.

Furthermore, the thesis will only evaluate the performance achievable using ParModelica. Possible bottlenecks caused by the current ParModelica implementation will be discussed, and when possible different approaches using ParModelica will be used. A native implementation using OpenCL is, however, beyond the scope of this thesis.

Finally, when this thesis started, the ParModelica implementation had become unstable, due to other more active OpenModelica development. Due to this, the initial design was done using a bigger set of algorithmic Modelica than supported by the current ParModelica implementation. Once enough parts of ParModelica worked, the initial performance measurements were
not satisfying. Hence, a number of iterations of the design had to be done, causing a race against time, with focus on good performance. This will be discussed further in Chapter 7.

1.4 Previous Work

This work is based on ParModelica [5] as tool, and PDEModelica [15] as reference. They are further discussed in Section 2.1.2.
Chapter 2

Background

Modelica is an object oriented and equation based language for modelling and simulation, using differential equations together with algebraic equations. It is also possible to write algorithmic code in Modelica. Currently Modelica only support derivatives with respect to time in equations. There are suggestions to add support for spatial derivatives, used in PDEs, to the Modelica language.

First, this chapter will give an introduction to the OpenModelica project and the OpenModelica compiler.

2.1 Tools Used

During this work a number of Open Source tools have been used. Apart of those in their own subsections I would like to mention Debian GNU/Linux as the main operating system, GCC for compilation, Emacs for text editing, \LaTeX{} for typesetting and TikZ for pictures.

2.1.1 ParModelica

Furthermore there is ParModelica[5], where explicit parallelism has been added to the algorithmic subset of Modelica. Similar to CUDA and OpenCL, it adds the concept of device memory and functions to be called on the device and within the device. This is covered further in Section 5.1.4.

2.1.2 OpenModelica

There are a number of implementations of Modelica, one of them is the Open Source implementation OpenModelica\(^1\). To have an Open Source implementation of the Modelica language is of great value for the Modelica

\(^1\)http://www.openmodelica.org/ accessed June 2015
community, both for users and as a platform for further research. It was also one of the main motivations when starting this master thesis.

The OpenModelica compiler contain several stages pictured in Figure 2.1. Each stage take care of one part of the compilation process.

**MetaModelica**

Within the OpenModelica project there are also implementations with suggested extensions to the Modelica language. To begin with, there is MetaModelica, intended for modelling programming languages, and used in the implementation of the Open Modelica Compiler (OMC).

### 2.2 Previous Research

**Partial Differential Equations in Modelica**

An extensive work on PDEs within Modelica has been done [15], suggesting language extensions to the Modelica language to support fields and describing spatial domains. Those extensions were implemented in PDEModelica.
Unfortunately, PDEModelica has not been maintained during the development of OpenModelica. However, the work is, nevertheless, a good reference for further work.

**Parallelisation in Modelica**

Within the OpenModelica project there have been various approaches to parallelise simulations. This is a list of a few of them:

- Implementation of a solver targeting the CUDA architecture[13, 14]
- Extending the algorithmic Modelica with explicit parallelism[5, 5, 8, 17, 6]
- Utilisation of the model structure[16]
- Compilation of unexpanded Modelica array equations[17]
- Using parallel skeletons from Modelica[17]
- Task based analysis of equation based languages[7, 6]
Chapter 3

Theory

This chapter presents theories around differential equations, numerical methods, and parallel computing.

3.1 Differential Equations

A differential equation is an equation where the value of a function is expressed using the function and the derivatives of the function with respect to one or more variables.

3.1.1 Ordinary Differential Equations (ODE)

An Ordinary Differential Equation (ODE) is a differential equation only depending on derivatives of one variable, generally time.

Figure 3.1 shows a ball attached to a spring, that is attached to the ceiling. There are two forces acting on the ball, it is the spring force $F_S$ and the gravitation force $F_g$. Using Newton’s second law ($\sum_i F_i = ma$), we can write an equation for the ball:

$$ky - mg = m\ddot{y}$$  \hspace{1cm} (3.1a)

![Figure 3.1: Ball attached to spring](image-url)
In this case it is possible to calculate an analytic expression for the function, and plot it like in Figure 3.2. This is generally not the case, and then a numerical solution plays an important role. In Modelica the model can be expressed as:

```model BallOscillating
    parameter Real k "Spring constant";
    parameter Real m "Mass";
    parameter Real g = 9.81 "Gravitation";
    Real y;
    Real dy;
    equation
        der(y) = dy;
        k*y - m*g = m*der(dy);
end BallOscillating;
```

### 3.1.2 Differential Algebraic Equations (DAE)

A Differential Algebraic Equation (DAE) adds constraints that do not depend on differentiated variables, such as Pythagorean theorem $c^2 = x^2 + y^2$ can be used to constrain the distance of an object moving around a point.
This is used in the pendulum example in [4]:

\[ m\ddot{v}_x = -\frac{x}{L}F \quad (3.2a) \]
\[ m\ddot{v}_y = -\frac{y}{L} - mg \quad (3.2b) \]
\[ \dot{x} = v_x \quad (3.2c) \]
\[ \dot{y} = v_y \quad (3.2d) \]
\[ x^2 + y^2 = L^2 \quad (3.2e) \]

### 3.1.3 Partial Differential Equations (PDE)

A PDE also depends on derivatives with respect to other variables than time. For example coordinates in space, also known as spatial derivatives. The equation for a vibrating string with constant tension can be expressed as [12, F-5.1]:

\[ \rho_t\frac{\partial^2\xi(x,t)}{\partial t^2} = F\frac{\partial^2\xi(x,t)}{\partial x^2} + f_y(x) \quad (3.3) \]

### 3.1.4 Explicit Form

In control theory and modelling it is often desirable to put an equation into explicit state form, see [9, 11, 4]. In the general form we have the state vector \( \vec{x}(t) \), the state derivative \( \dot{\vec{x}}(t) \), the input vector \( \vec{u}(t) \), and the output vector \( \vec{y}(t) \). In the general case we have the equations:

\[ \dot{\vec{x}}(t) = \vec{f}(\vec{x}(t), \vec{u}(t)) \quad (3.4a) \]
\[ \dot{\vec{y}}(t) = \vec{g}(\vec{x}(t), \vec{u}(t)) \quad (3.4b) \]

In case \( f \) and \( g \) are linear matrix notation can be used instead:

\[ \dot{\vec{x}}(t) = A\vec{x}(t) + B\vec{u}(t) \quad (3.5a) \]
\[ \dot{\vec{y}}(t) = C\vec{x}(t) + D\vec{u}(t) \quad (3.5b) \]
This thesis will only use the general form in Equation (3.4). The equation in (3.2) is not in explicit form.
3.2 Numerics

To give a better understanding of the implementation, this section covers the algorithms involved in simulating mathematical models. For further reading, see: [4, 11, 3], or another book covering numerical analysis or applications of numerical analysis.

3.2.1 Approximating Derivatives

If the value of a function $f(x)$ is known at a point $x_i$, and at another point $x_{i\delta}$ close to $x_i$, then the derivative of the function can be approximated at $x_i$:

$$
\dot{f}(x_j) \approx \frac{f(x_i) - f(x_{i\delta})}{x_i - x_{i\delta}} \quad (3.6)
$$

If $x_{i\delta} < x_i$ it is a backward approximation, while if $x_{i\delta} > x_i$ it is a forward approximation. If the value of $f(x)$ is available on both sides of $x_i$, such as $x_{i\delta-} < x_i < x_{i\delta}$, the derivative with respect to $x$ can be approximated as:

$$
\dot{f}(x_j) \approx \frac{f(x_i) - f(x_{i\delta})}{x_i - x_{i\delta}} \quad (3.7)
$$

This is called a centre approximation, since $x_i$ lies between two points. Figure 3.5 illustrates how the first order derivatives with respect to $x$ are approximated at one point.

If all values of $f(x_i)$ are separated by $h$, the approximations can be written as:

$$
\dot{f}_-(x_i) = \frac{f_i(x_i) - f_{i-1}(x_{i-1})}{h} \quad (3.8a)
$$

$$
\dot{f}_c(x_i) = \frac{f_i(x_{i+1}) - f_{i-1}(x_{i-1})}{2h} \quad (3.8b)
$$

$$
\dot{f}_+(x_i) = \frac{f_{i+1}(x_i) - f_i(x_i)}{h} \quad (3.8c)
$$

3.2.2 Discretisation

To be able to solve a PDE over space and time, one approach is to discretise the PDE over space and this way get a system of ODEs. If we take the heat conduction equation from [12], with $\nabla^2$ expanded to two dimensions, and calculate it at $n_x \times n_y$ discrete points in space we get:

$$
\frac{\partial T_{i,j}}{\partial t} = \kappa_{i,j} \nabla^2_{i,j} T + \left( \frac{\kappa h}{\lambda} \right)_{i,j} = \kappa_{i,j} \left( \frac{\partial^2 T_{i,j}}{\partial x^2} + \frac{\partial^2 T_{i,j}}{\partial y^2} \right) + \left( \frac{\kappa h}{\lambda} \right)_{i,j} \quad (3.9)
$$
(a) Backward: $\frac{\Delta y}{\Delta x} = 1$

(b) Centre: $\frac{\Delta y}{\Delta x} = 2$

(c) Forward: $\frac{\Delta y}{\Delta x} = 3$

Figure 3.5: Derivative approximations

Boundary

$\begin{align*}
T_{1,1} & \quad T_{2,1} & \quad T_{3,1} & \quad T_{4,1} & \quad T_{5,1} & \quad T_{6,1} \\
T_{1,2} & \quad T_{2,2} & \quad T_{3,2} & \quad T_{4,2} & \quad T_{5,2} & \quad T_{6,2} \\
T_{1,3} & \quad T_{2,3} & \quad T_{3,3} & \quad T_{4,3} & \quad T_{5,3} & \quad T_{6,3} \\
T_{1,4} & \quad T_{2,4} & \quad T_{3,4} & \quad T_{4,4} & \quad T_{5,4} & \quad T_{6,4}
\end{align*}$

Figure 3.6: Method of lines
Figure 3.7: Example of stencils used for calculating spatial derivatives. The red box symbolises the destination, while the numbers are the weights to use when summing up the neighbouring values. They are all approximations, and some can be derived in different ways, resulting in different weights.

Figure 3.6 shows how $T$ has been discretised over a grid with $10 \times 8$ points. The derivatives in Equation (3.9) can be approximated with:

\[
\frac{\partial^2 T}{\partial x^2} = \frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2} \tag{3.10a}
\]
\[
\frac{\partial^2 T}{\partial y^2} = \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y^2} \tag{3.10b}
\]

Those approximations can be derived using Taylor series, see for example [18] or [3].

As seen in Equation (3.10), the discretisations (in one direction) will depend on the points on both sides. This is called a central difference, while there are also forward and backward differences, only depending on points at one side.

Due to the dependency of points at the sides, the boundaries need to be treated specially. How they are treated depends on the boundary condition of the model. In the heat conduction case one may assume the temperature is constant at the borders, so for example $T_{0,j} = T_{1,j}$, and expand the values at the boundaries. Other models may have other boundary conditions.

Due to the amount of points, with one ODE at each point, the method of lines approach will produce, this can result in fairly large matrices if using an implicit solver. If, on the other hand, an explicit solver is used, this gives a potential for lots of parallelism. If running on a General Purpose Graphic Processing Unit (GPGPU) each thread can have its own point.
3.2.3 Euler

Given the equation Equation (3.4a), we want to find a way to calculate $x(t)$. At the point $t = t_n$ the differential equation is:

$$
\dot{x}(t_n) = f(x(t_n), u(t_n)) \quad (3.11)
$$

The value of $\dot{x}(t_n)$ can be approximated with:

$$
\dot{x}(t_n) \approx \frac{x_{n+1} - x_n}{h} \quad (3.12)
$$

Equation (3.11) & (3.12) $\implies$

$$
x(t_{n+1}) \approx x(t_n) + hf(x(t_n), u(t_n)) \quad (3.13)
$$

This is known as Euler’s method. When working on vectors, all values can be calculated in parallel.

3.2.4 Runge-Kutta

The Runge-Kutta method is a more accurate method than Euler’s method. It calls the function $f$ for several intermediate points in the time step, and uses the values from those points to calculate a new value:

$$
k_1 = hf(x_n, x_n) \quad (3.14)
$$

$$
k_2 = hf(t_n + \frac{h}{2}, x_n + k_1) \quad (3.15)
$$

$$
k_3 = hf(t_n + \frac{h}{2}, x_n + k_2) \quad (3.16)
$$

$$
k_4 = hf(t_n + h, x_n + k_3) \quad (3.17)
$$

$$
x_{n+1} = x_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \quad (3.18)
$$

3.2.5 Runge-Kutta with Variable Step Length

If the value of $x_{n+1}$ is approximated with different order of error, the values can be compared to get an estimate of the local error. In [1], parameters for calculating both a third and second order approximation using four compu-
tations of $k$ are suggested:

$$k_1 = f(x_n, x_n)$$  
(3.19a)

$$k_2 = f(t_n + \frac{1}{2}h, x_n + \frac{1}{2}hk_1)$$  
(3.19b)

$$k_3 = f(t_n + \frac{3}{4}h, x_n + \frac{3}{4}hk_2)$$  
(3.19c)

$$x_{n+1}^{(3)} = x_n + (\frac{2}{9}k_1 + \frac{1}{3}k_2 + \frac{4}{9}k_3)h$$  
(3.19d)

$$k_4 = f(t_n + h, x_{n+1})$$  
(3.19e)

$$x_{n+1}^{(2)} = x_n + (\frac{7}{24}k_1 + \frac{1}{4}k_2 + \frac{1}{3}k_3 + \frac{1}{8}k_4)h$$  
(3.19f)

Using the two predictions $x_{n+1}^{(3)}$ and $x_{n+1}^{(2)}$ of third and second order, it is possible to estimate the error during the step. The error can be used to decide if the step should be accepted or restarted with a shorter step size. It is also possible to estimate a new step size.

### 3.3 Parallel Computing

Parallel computing is a fairly old concept within computer science and there are different architectures for different purposes. It is also common with heterogeneous architectures containing different parallel architectures within the same system. This section aims to give an overview of different architectures for parallel computing used today.

#### 3.3.1 Memory Hierarchy

A common bottleneck within computer hardware is memory access. To speed up memory access there are different levels of memory, with as fast memory as possible close to the CPU. This memory is a small portion of the total memory. Usually memory transfer between different levels of the hierarchy is done transparently to the user. In this case the faster memory close to the CPU is called cache memory.

#### 3.3.2 Multiple CPUs or CPU Cores with Shared Memory

Most modern CPUs have multiple CPU cores within the same CPU. Those cores can be utilised to perform parallel computations within one CPU. A host can also have multiple CPUs in different sockets. Each CPU will typically have cache memory dedicate to each CPU core. Furthermore, the CPUs will have a common cache for all cores within the CPU. This is referred to as Symmetric Multiprocessing (SMP). If all CPUs within a host share access to all memory, the memory access is referred to as Uniform
Memory Access (UMA). If the memory is partitioned between the CPUs, it is referred to as Non-Uniform Memory Access (NUMA). In this case the CPUs will communicate with each other to get access to memory that belongs to another CPU.

3.3.3 Computer Cluster

If multiple computers are connected to each other using a network with the purpose to distribute computations or data between them, it is referred to as a cluster and the individual computers are referred to as nodes. Here special software is usually utilised for communication between the nodes.

3.3.4 Single Instruction Multiple Data (SIMD)

When a CPU is capable to perform the same instruction on multiple data at once, it is referred to as Single Instruction Multiple Data (SIMD). It is common for modern CPUs to have a set of SIMD instructions.

3.3.5 General-Purpose Computing on Graphics Processing Units (GPGPU)

A Graphic Processing Unit (GPU) can be used as a computation device attached to a host. Within a GPU there are multiple Streaming Multiprocessor (SM) or CUDA cores able to handle multiple one thread each. The SM or cores are simplified compared to a CPU, so it is the amount of them that makes the GPU powerful. The GPU will have its own memory, divided into a bigger global memory, and a smaller and faster local memory. The local memory can be used as a user controlled cache. GPUs usually has its own cache too, giving a transparent memory hierarchy. When a GPU device is used within a host computer, it will result in a heterogeneous system. The host can either be used just to control the device, or carry out its own computations.
General-Purpose computing on Graphics Processing Units (GPGPU)

Figure 3.8: Computer equipped with GPU
Chapter 4

Methods

This chapter covers methods used during prestudy of the fields: PDE applications, PDEs in Modelica, numerical solvers, and algorithmic code in Modelica and ParModelica. Moreover, methods considered for designing and implementing a framework for solving PDEs in parallel are discussed. Finally, different use cases for test and evaluation are introduced.

4.1 Prestudy

To be able to write a framework for solving PDEs using ParModelica, a number of fields need to be studied.

4.1.1 PDE Applications

To get an idea of different PDEs, the application of PDEs needs to be studied in different fields of engineering. For this the previous work on PDEs in Modelica: [15, 4] are used, since they both bring up applications. Two course books are also used [2, 10]. Furthermore, a brief study of applications at Linköping University is done, including meeting a PhD student and attending the defence of his thesis [18].

4.1.2 PDEs in Modelica

To describe PDEs in Modelica, it is reasonable to study previous and ongoing work on PDEs in Modelica. For this [15, 4] are used as reference. A meeting was held with PhD student Jan Šilar, who authored a paper on simplified PDE extension\(^1\) and another paper comparing with Levon Saldamli’s PhD thesis\(^2\) together with Professor Peter Fritzson, who authored [4] and


supervised [15].

4.1.3 Numerical Solvers Suitable for PDEs

To write a solver suitable for simulating PDEs, the field of numerical solvers needs to be studied to get an idea of a suitable solver for PDEs. For this a lecturer in numerical methods at Linköping University, PhD Fredrik Berntsson, was contacted for a meeting.

4.1.4 Algorithmic Modelica and ParModelica

To get an idea of how to write a numerical solver in algorithmic Modelica and ParModelica, both topics need to be studied. As Modelica reference [4] is used together with the online resources Modelica by example\(^3\) and the Modelica specification\(^4\). As ParModelica references [5, 8] are used. Mahder Gebremedhin, now PhD student, was also assigned as technical supervisor.

The main points to study here:

- How to represent a user defined function and call it from the solver, both in the context of what is possible from the language point of view, and how to get good performance
- How to represent types needed by the solver
- How to optimise scalability
- How to optimise performance

4.2 Design

Considering the discretised field from Section 3.2.2 and Figure 3.6 the state, a suitable abstraction of fields would be to put them in a multidimensional array, maybe inside a record to keep it together with the size of the domain it belongs to.

The input for a solver in Sections 3.2.3 to 3.2.5 may not take only one state/field, but multiple states/fields. For this reason, arrays of fields are considered. Also the solver needs a way to calculate the state derivative at a given state, together with other possible input. Since the user should be able to simulate different models, the user should be able to provide the state derivative function.

The design should also take the results from Sections 4.1 and 5.1 into account.

4.3 Implementation

The implementation should be done using ParModelica and algorithmic Modelica. It should be possible to compare the results between two different solvers, and from the same solver implemented in ParModelica and algorithmic Modelica. It should also be possible to evaluate the scalability of a parallel solver. Results from Sections 4.2 and 5.2 should be taken into account during the implementation.

4.4 First Use Case — Heat in Plane

The first use case is to simulate heat conduction in a plane. The plane has an initial square patterned temperature according to Figure 4.1. The motivation for an initial state with discontinuities is to stress the solvers, rather than to be realistic.

Since PDEs can have different boundary conditions, the model should be simulated using different boundary conditions. This use case will also be used for comparing the solvers with each other. The simulations will be done on a Fermi M2050 GPU with a total of 448 threads available. As a comparison, the serial solvers are used on the host CPU, an Intel Xeon E5520 2.27GHz.

Heat equation from [12]:

$$\frac{\partial T}{\partial t} = \kappa \nabla^2 T + \left( \frac{\kappa h}{\lambda} \right)$$ (4.1)
Figure 4.2: Rod where heat is added at the left side.

Boundary conditions to test:
\[
\frac{\partial T}{\partial x} = 0 \text{perfect insulation}
\]
\[
\frac{\partial T}{\partial x} = a \times (T - T_{ext}) \text{poor insulation}
\]
\[
\frac{\partial T}{\partial x} = b \text{energy added/removed at constant rate}
\]
\[
T = c \text{constant temperature at boundary}
\]

The boundaries may be combined in same and different models.

### 4.5 Second Use Case — Heat in Rod

The second use case is to simulate heat conduction in a rod. Heat is added at one end, depending on the temperature at the other end. This model is used as a three dimensional use case, and to evaluate scalability of a parallel solver.
Chapter 5

Results

In this chapter the results of prestudy, design, and implementation are presented. Moreover, two use cases are presented together with test results. The results are discussed in Chapter 6.

5.1 Prestudy

The following section presents the results from prestudies of: PDE applications, previous work on PDE representation in Modelica, numerical solvers suitable for PDEs, and how algorithmic Modelica and ParModelica can be used to implement a PDE-solver.

5.1.1 PDE Applications

Examples from the domain of PDEs:

- Fluid Mechanics
- Solid Mechanics
- Thermal Physics
- Field and Wave Electromagnetics
- Heat Equations
- Image Processing

Types of fields:

- Scalar fields
- Vector fields
- Complex scalar/vector fields

Operations within a PDE:

- Derivatives with respect to time
- Derivatives with respect to one or more spatial variable
• Linear and non-linear operations on fields and spatial derivatives of fields
• Chained operations where spatial derivatives are applied to a field, followed by a linear or non-linear operation, finally followed by more spatial derivatives
• Scalar fields and vector fields
• Operations transforming a scalar field to a vector field (gradient)
• Operations transforming a vector field to a scalar field (divergence)
• Operations transforming a vector field to a new vector field (curl/rotation)
• Operations between two vector fields resulting in a scalar field (scalar product)
• Operations between two vector fields resulting in a new vector field (cross product, addition, subtraction)

Coordinate systems for PDEs:
• Cartesian coordinate system (1D, 2D, 3D)
• Polar/circular coordinate system (2D)
• Spherical and cylinder coordinate systems (3D)
• Curvilinear coordinates (1D, 2D, 3D)

5.1.2 PDEs in Modelica

Available in Modelica 3.3:
• spatialDistribution, allowing modelling of variable speed transport[4]

Suggested extension [4, 15]:
• field variables
• indomain construct

This would allow modelling a heat equation on a plane similar to:

```model HeatInPlane
  parameter Real c;
  parameter Real q;
  parameter Real h;
  field Real T(domain=omega);
  equation
    c*der(T) = pder(T,D.x2) + pder(T,D.y2) indomain omega.interior;
    c*pder(T,D.x) = q+h*(T_ext-T) indomain omega.left;
    T = 50 indomain omega.right;
    pder(T,D.y) = 0 indomain omega.top;
    pder(T,D.y) = 0 indomain omega.bottom;
```

5.1.3 Numerical Solvers Suitable for PDEs

As solver a Runge-Kutta solver with adaptable step size[1] is chosen. For comparison, a simpler Predictor-Corrector\(^1\) method is chosen.

5.1.4 Algorithmic Modelica and ParModelica

In an algorithmic context, the following concepts from Modelica is of interest:

- Functions
- Records
- Arrays
- Enumeration types
- Type definitions
- Partial function, if implemented by the user they can be passed as argument to an other function

While it is possible to define arrays of arrays, this should be seen as a multidimensional array with rectangular/box-shape. It is not possible to construct arrays containing arrays of different sizes. The same applies if an array of records containing arrays is constructed.

ParModelica adds the concept of:

- parkernel function, called from non parallel context
- parfor loop, called from non parallel context
- parallel function, called from a parkernel function or parfor loop
- parlocal memory, static size set at compile time and used within a parkernel function or parallel function
- parglobal memory, allocated in a non parallel context and passed to a parkernel function during executions

ParModelica does, however, add limitations:

- Does not support records
- Does not support partial functions as argument to other functions
- Arrays of arrays should be considered as multidimensional arrays, so it is not possible to define an array on the host containing a number of parglobal arrays to be passed to a parkernel function
- Passing parglobal arrays to an ordinary Modelica function does not work\(^2\)

\(^2\)Bug not fixed during the time of this thesis.
• The number of possible dimensions for a multidimensional array is fixed to three, but increased to four during this work
• Arrays cannot be defined within a parallel context
• Calling a parkernel function add overhead, so as much as possible should be done within one call\(^3\)
• Calling a parkernel function multiple times can be fragile\(^4\)
• Unable to declare and use parlocal arrays\(^5\)

5.2 Design

In this section, the design with requirements from Section 4.2 and parts of the limitations from Section 5.1 is presented.

5.2.1 User Defined State Derivative Function

Given the three types of functions in ParModelica, there are three candidates for how to define a user defined function:

• Ordinary Modelica function
  – Can only be called from an ordinary function
  – Can only call ordinary functions\(^6\) and parkernel functions
  – Can only pass parglobal arrays to parkernel functions
  – Will be called many times by the solver
  – Would make many calls to parkernel functions
  – Can operate on one field at the time, creating new intermediate fields by operating from other fields

• parkernel function
  – Can only be called from an ordinary function
  – Can call parallel functions
  – Harder to create intermediate fields
  – Work from different workgroups is synchronised after each call to a parkernel function

• parallel function
  – Can be called from parkernel function
  – Can be called from parallel function
  – Cannot synchronise between workgroups
  – Harder to create intermediate fields
  – Solver needs to synchronise calls
  – Several calls will be done at different points over the fields

\(^3\)Discussed in Section 6.2.4
\(^4\)Need to be investigated further.
\(^5\)Bug not fixed during the time of this thesis.
\(^6\)Ordinary functions cannot take parglobal arrays due to bug
parallel function is the only viable alternative for user provided function. Due to limitation in ParModelica, it cannot be a partial function, so it needs to have the right name and be placed at right place in the filesystem before compilation.

5.2.2 Representation of Types for Solver

Fields are can only be represented as multidimensional arrays with fixed number of dimensions. Three dimensional fields and multiple fields → four dimensional arrays where the size of one or more dimension may be one.

No viable way to abstract domains and coordinates. Domain information can be represented using a package with the right name and location before compilation. Coordinates can be represented using three different scalars.

At some points, the value of several fields need to be merged point by point while minimising memory access. This is for example the case when calculating the next step in the a solver. The first one, as shown in Figure 5.2, an array of fields will be merged into one field. The second function, as shown in Figure 5.3, is operating on arrays of fields. Depending on the way the user defined function for state derivative is defined, this concept is implemented in different ways.

5.2.3 How to Optimise Scalability

As seen in Figure 5.2 and Figure 5.3, there are quite some computations to do over the whole field. The upper limit of parallelism here is set by the size of the field. When multiple fields are operated on at once, all those computations could be performed in parallel too.

How to Optimise Performance

To optimise performance, first an efficient algorithm should be used. This have already been chosen in Section 5.1.3. Next the algorithm need to be implemented in an efficient way. For this both computations and memory access need to be considered.
Figure 5.3: Merge field arrays

Figure 5.4: How a stencil is applied in a two dimensional domain. The offset is used to calculate the target point in a region of points.
From memory access point of view, the access pattern when computing spatial derivatives should be considered. As seen in Figure 5.4 the value in the destination field will depend on the surrounding at a point. Hence, points next to each other will have overlapping points they depend on. If the memory hierarchy can load data into local memory, that is faster than the global memory, one may benefit from loading all data a workgroup depends on into local memory before performing the reduction operation. This is illustrated in Figure 5.5. If the memory hierarchy has an efficient cache memory, one may instead access memory close to each other at the same time. This way the data needed will remain in cache memory for consecutive accesses.

5.3 Implementation

This section will describe the implementation of the solvers and how they are used.
5.3.1 User Defined State Derivative and Settings

The user should provide a function for computing the state derivative. For a solver using ParModelica this should be a parallel function and named ParDerState. This function should be within the PDESolver hierarchy, in the subpackage Model. A serial solver using algorithmic Modelica will instead use the function DerState within same package. The function gets the current state, user provided variables, external fields, a time to compute the state derivative at, and the discrete coordinates as three scalars. The function will then return up to three scalars for up to three different fields. Here the function interface together with a sample model is given:

```plaintext
within PDESolver.Model;

parallel function ParDerState "Calculate the state derivative"
    import Functions = PDESolver.ParFunctions;
    import PDESolver.ParFunctions.Pder;
    import PDESolver.Types;

    input Types.Field[:] state "Array of state fields";
    input Real var[:];
    input Types.Field ext[:];
    input Real t "Time to calculate the state derivative at";
    input Integer i,j,k "Discrete coordinate within field";
    output Real value1;
    output Real value2;
    output Real value3;

protected
    // User defined
    Real d2Tdx2, d2Tdy2;
    Real c = var[1];

algorithm
    // User defined
    nDer := 0; // Perfect insulation
    d2Tdx2 := Pder.Pder2Neumann(f=state, fi=1, i=i, j=j, k=k, dim=1, nder=nDer);
    d2Tdy2 := Pder.Pder2Neumann(f=state, fi=1, i=i, j=j, k=k, dim=2, nder=nDer);
    value1 := c*(d2Tdx2 + d2Tdy2)*ext[1,i,j,k];
end ParDerState;
```

For describing the domain and how the field is discretised, the user should provide a Settings package for the model. Here it is also possible to add static parameters for the model.

```plaintext
within PDESolver.Model;

package Settings
    // Parameters used by the solver
    constant Types.FieldIndex n = {80,40,1};
    constant Types.Coordinate first = {0,0,0};
```

7 Ideally it should have been “output Real value[size(state,1)]” instead
constant Types.Coordinate last = {2,1,0};
constant Integer stateFields = 1 "T";
// Parameters used in the model
constant Integer boxSize = integer(n[2]/2);
constant Integer myBoundary = 3;
end Settings;

5.3.2 Types Used by the Solver

There are two types used by the solver field type, implemented as a four dimensional array, and an enumeration type to select solver. For the field type the dimensions are taken from the user provided settings.

within PDESolver;

type Field = Real[Model.Settings.n[1], Model.Settings.n[2], Model.Settings.n[3]]
  (each start=0, each fixed=true);

within PDESolver.Solver;

type SolverId =
  enumeration(SerialPECE, SerialRK32, ParallelPECE, ParallelRK32);

5.3.3 Solvers

The user interface for simulating a PDE is the Solve function. It will take a state array, external fields, user provided variables, the current time, time to step forward to, and a SolverId as compulsory arguments. It is also possible to provide arguments for initial intermediate step size, maximum error during one step, maximum intermediate step size. The arrays provided are host variables and for the parallel solvers they will be copied to parglobal variables before calling the solver. The result is then copied back and returned as next state.

within PDESolver.Solver;

function Solve
  input Types.Field state[:]; "Current state"
  input Types.Field ext[:]; "External field"
  input Real var[:]; "External variables"
  input Real t0 "Time at ‘state’"
  input Real t1 "Time at ‘next’"
  input SolverId solverId;
  input Real dt = (t1-t0)*2 "Initial intermediate step length."
  input Real eMax = 0.1 "Max error"
  input Real hMax = dt "Max intermediate step length"
  input Integer th1=1, th2=1, th3=1;
  output Types.Field next[size(state,1)] "New state";
The solvers need different intermediate fields. Since a parkernel function in ParModelica cannot have internal arrays, output variables are used as intermediate fields.

For merging fields within the parallel solvers, ordinary for-loops are used. Each thread will calculate the initial index, step size, and final value for the loops. The full source code for the solvers can be found in Appendix A.1.

5.4 First Use Case — Heat in Plane

5.4.1 Poor Insulation and Constant Temperature

The boundary conditions here are the same as in [4], with poor insulation at the left side and constant temperature at the right side. The top and bottom have perfect insulation.

```plaintext
if Functions.AtBoundary(i,j,k) then
  if Functions.AtFirst(i=i,j=j,k=k,dim=1) then // Left side
    d2Tdx2 := Pder.Pder2Dirichlet(f=state, fi=1,
      i=i, j=j, k=k,
      dim=1, boundary=50);
  elseif Functions.AtLast(i=i,j=j,k=k,dim=1) then // Right side
    nDer := q + h*(T_ext - state[1,i,j,1]);
    d2Tdx2 := Pder.Pder2Neumann(f=state, fi=1,
      i=i, j=j, k=k,
      dim=1, nder=nDer);
  end if;
  d2Tdy2 := Pder.Pder2Neumann(f=state, fi=1,
    i=i, j=j, k=k,
    dim=2, nder=0);
else
  d2Tdx2 := Pder.Pder2Inner(f=state, fi=1, i=i, j=j, k=k, dim=1);
  d2Tdy2 := Pder.Pder2Inner(f=state, fi=1, i=i, j=j, k=k, dim=2);
end if;
```

```
if Functions.AtBoundary(i,j,k) then
  if Functions.AtFirst(i=i,j=j,k=k,dim=1) then // Left side
    d2Tdx2 := Pder.Pder2Dirichlet(f=state, fi=1,
      i=i, j=j, k=k,
      dim=1, boundary=50);
  elseif Functions.AtLast(i=i,j=j,k=k,dim=1) then // Right side
    nDer := q + h*(T_ext - state[1,i,j,1]);
    d2Tdx2 := Pder.Pder2Neumann(f=state, fi=1,
      i=i, j=j, k=k,
      dim=1, nder=nDer);
  end if;
  d2Tdy2 := Pder.Pder2Neumann(f=state, fi=1,
    i=i, j=j, k=k,
    dim=2, nder=0);
else
  d2Tdx2 := Pder.Pder2Inner(f=state, fi=1, i=i, j=j, k=k, dim=1);
  d2Tdy2 := Pder.Pder2Inner(f=state, fi=1, i=i, j=j, k=k, dim=2);
end if;
```

5.4.2 Perfect Insulation

All boundaries have perfect insulation.

```plaintext
nDer := 0; // Perfect insulation
d2Tdx2 := Pder.Pder2Neumann(f=state, fi=1,
  i=i, j=j, k=k, dim=1, nder=nDer);
d2Tdy2 := Pder.Pder2Neumann(f=state, fi=1,
  i=i, j=j, k=k,
...
5.4. First Use Case — Heat in Plane

Figure 5.6: Plane with poor insulation to the left and constant heat to the right at $t = 0.5$

\[
dim=2, \text{nder}=\text{nDer};
\]
\[
value1 := c \times (d2Tdx2 + d2Tdy2);
\]

5.4.3 Heat Added or Removed at Constant Rate Depending on Location

Energy is added or removed at constant rate along the boundary, depending on location.

\[
nDer := 100 \times (\text{BoxZeroOne}(i,j) - 0.5);
\]
\[
d2Tdx2 := \text{Pder}\text{.Pder2Neumann}(\text{f}=\text{state}, \text{fi}=1, i=i, j=j, k=k, \text{dim}=1, \text{nder}=\text{nDer});
\]
\[
d2Tdy2 := \text{Pder}\text{.Pder2Neumann}(\text{f}=\text{state}, \text{fi}=1, i=i, j=j, k=k, \text{dim}=2, \text{nder}=\text{nDer});
\]
\[
value1 := c \times (d2Tdx2 + d2Tdy2);
\]

5.4.4 Poor Insulation with Different Temperature Outside Depending on Location along the Boundary

Heat is added or removed at constant rate depending on location, and state at the location.

\[
nDer := 100 \times \text{BoxZeroOne}(i,j) - \text{state}[1,i,j,1];
\]
\[
d2Tdx2 := \text{Pder}\text{.Pder2Neumann}(\text{f}=\text{state}, \text{fi}=1,
Figure 5.7: Plane with perfect insulation at all boundaries at \( t = 0.5 \)

Figure 5.8: Energy is added or removed at constant rate along the boundary, depending on location. \( t = 0.5 \)
5.4. First Use Case — Heat in Plane

Figure 5.9: Heat is added or removed at constant rate depending on location, and state at the location. $t = 0.5$

$$i=i, \ j=j, \ k=k,$$
$$\text{dim}=1, \ \text{nder} = nDer);$$

$$\text{d2Tdy2} := \text{Pder.Pder2Neumann}(f=\text{state}, \ \text{fi}=1,$$
$$\ \text{i}=i, \ \text{j}=j, \ \text{k}=k,$$
$$\ \text{dim}=2, \ \text{nder} = nDer);$$

$$\text{value1} := c*(\text{d2Tdx2} + \text{d2Tdy2});$$

5.4.5 Constant Temperature Depending on Location

Boundaries have constant temperature depending on location.

$$\text{boundaryT} := 100*\text{BoxZeroOne}(\text{i}, \ \text{j});$$
$$\text{d2Tdx2} := \text{Pder.Pder2Dirichlet}(f=\text{state}, \ \text{fi}=1,$$
$$\ \text{i}=i, \ \text{j}=j, \ \text{k}=k,$$
$$\ \text{dim}=1, \ \text{boundary}=\text{boundaryT});$$
$$\text{d2Tdy2} := \text{Pder.Pder2Dirichlet}(f=\text{state}, \ \text{fi}=1,$$
$$\ \text{i}=i, \ \text{j}=j, \ \text{k}=k,$$
$$\ \text{dim}=2, \ \text{boundary}=\text{boundaryT});$$

$$\text{value1} := c*(\text{d2Tdx2} + \text{d2Tdy2});$$

5.4.6 Constant Temperature Depending on Location Combined with Periodic Boundary

Top and bottom boundaries have constant temperature depending on location, while the sides have periodic boundary.
Figure 5.10: Boundary have constant temperature depending on location. \( t = 0.5 \)

```plaintext
boundaryT := 100*BoxZeroOne(i,j);
d2Tdx2 := Pder.Pder2Periodic(f=state, fi=1,
i=i, j=j, k=k,
dim=1);
d2Tdy2 := Pder.Pder2Dirichlet(f=state, fi=1,
i=i, j=j, k=k,
dim=2, boundary=boundaryT);
value1 := c*(d2Tdx2 + d2Tdy2);
```
Figure 5.11: Top and bottom boundary have constant temperature depending on location, while the sides have periodic boundary. $t = 0.5$
Figure 5.12: Speedup of heat in rod simulation
Chapter 6

Discussion

In this chapter the results and methods used are discussed.

6.1 Results

In this section the results of prestudy, design, implementation, and use cases are discussed.

6.1.1 Prestudy

The prestudy of PDEs and numerical solvers was fairly successful. So was the prestudy of earlier work on PDEs in Modelica. The prestudy of algorithmic Modelica and ParModelica went quite fine when reading about the concepts. When trying to use ParModelica it did, however, not work. At that time it had not been used for awhile. Algorithmic Modelica was studied more in general instead, while parts supported and not by ParModelica was less considered.

6.1.2 Design

Without a working ParModelica implementation, the design was continued using algorithmic Modelica instead, hitting a few bugs and limitations on the way. The design had to be iterated a couple of times. Once a working ParModelica was available, it did not support the features used in the initial design. There was also bugs in ParModelica, needed to be worked around. This resulted in a few more iterations.

6.1.3 Implementation

As already mentioned in Section 6.2.2, the design had to be iterated a number of times due to bugs and limitations. The final implementation is fairly
good. It does, however, not support intermediate fields while computing state derivatives. Furthermore, it can be error prone to specify three indexes for coordinates, instead of just an abstract coordinate. When one dimension is unused, or the model only have one state field, it can become a bit of extra work to specify the unused field dimension as one. The same applies if the model only have one state field. It is not desirable to limit the number of state fields due to not being able to define small arrays within ParModelica’s parkernel functions.

6.1.4 Use Cases

The use cases shows how to implement a PDE model. They could also be used for evaluating the solvers accuracy, performance, and scalability. It seems ParModelica have some issues around multiple calls to parkernel functions, both from performance point of view and stability. The stability issues have mainly occurred when using bigger models.

6.2 Method

In this section the methods used during prestudy, design, implementation, and use cases is discussed.

6.2.1 Prestudy

In general the prestudy was fairly fine. However, more care could have been taken when studying ParModelica to learn more about its limitation and implementation. This could have saved time during design and implementation.

6.2.2 Design

While the ideas around how to do the design were in general good, there were a bit of issues along the way. Those issues were mainly related to limitations and bugs mentioned in Section 5.1.

6.2.3 Implementation

The methodology for implementation suffered a bit of assumptions that algorithmic Modelica would be more like a traditional programming language, and tested thoroughly. ParModelica was then assumed to be an extension without adding restrictions. Those assumptions where a bit naive. If the design would have been targeting a more traditional programming language the implementation could have been smoother.

Since Modelica only offers pure functions with input and output parameters it can be hard to do abstractions where only a part of an array is
updated. For this it would be desirable to have a concept like procedures from Ada, where an argument can be given as \texttt{in out} to specify that it can be updated by the procedure.

### 6.2.4 Use Cases

The use cases shows how a PDE can be modelled using different boundary conditions. They could also have covered other coordinate systems.
Chapter 7

Conclusions

This thesis shows how an algorithmic solver can be implemented to utilise the explicit parallelism of ParModelica. In this chapter, the results will be discussed and other possible approaches will be discussed.

7.1 Summary

For solving PDEs, a framework can be implemented using ParModelica. To gain performance, however, care must be taken of where the user code is added. If the solver is written as a kernel function, calling a parallel function provided by the user, it is possible to get two order of magnitudes better performance.

7.2 Pros & Cons of Solver Written in Modelica

In the context of a bigger system, where part of it need a special solver, there can be advantages to write the solver in Modelica. The solver can then form a framework where the user only add a small portion of code. For this the suggested ParModelica extensions may also be used to gain better performance. Another advantage with ParModelica can be when evaluating the performance of a potential parallel solver. Then the solver can be written using ParModelica to get an idea of performance gain and bottlenecks when applying the solver to different problems.

The true power of Modelica is to solve equations. While the language does have an algorithmic subset, it is hard to compete with other general purpose programming languages.
7.3 Other (possible) Approaches

The design already brings up some approaches that could not be implemented. This section will discuss some other approaches that have been considered.

7.3.1 SimpleSIMD

With lots of the same operations all over the field, one may make the connection to SIMD hardware. One considered approach was to specify a SimpleSIMD instruction set for operating over the fields. The user would then write a little program with those instructions. This could make better usage of local memory. It would, however, be less user friendly. It may still be an approach to consider if the compiler backend is used to optimise the instructions and memory usage of a solver.

7.3.2 External Objects

Instead of using multidimensional arrays to represent fields, external objects could have been used. With this approach the utility functions would be implemented in C/OpenCL/CUDA. While ParModelica add some extra overhead for kernel calls, the results achieved suggests more powerful utility functions are required to gain desirable speedup.

7.3.3 Simulation Runtime

Ideally the simulation runtime should be used for solving PDEs, but this would require support for PDEs all the way from the frontend, through the backend, to finally call a solver in the simulation runtime during simulation. This is a long chain of work depending on each other, where it can be hard to cut out one part.

7.4 Further Work

7.4.1 Short Term

While this work tries to provide the functionality to simulate various PDEs, it is hard to predict all needed functionality without specific use cases. For this it is necessary to simulate more models. Stability and errors introduced by the discretisation and solvers also need further work.

There is also ongoing work, by PhD student Jan Šilar, to add PDE extensions to the frontend.

During this work, quite a few bugs were reported on ParModelica, and a few on OpenModelica in general. Other matters also need to be investigated further. It would be a pity not to investigate the issues further, and lamentable not to fix and admit test cases for known issues.
7.4.2 Long Term

Once we have PDE extensions in the frontend, and know an efficient way to simulate them, this needs to be integrated into all stages of the OpenModelica compiler and simulation runtime.

The visualisation of PDE-simulations in this report has all been done using Ti\textit{k}Z. While this is feasible for writing a report, it may be less convenient for the general user. For this 3D visualisation and animation of solutions should be more comprehensible.

7.5 Contributions to the Modelica Community

Apart of what can be read in this report, this work has also tested the implementation of algorithmic Modelica code in OpenModelica and served as a use case for ParModelica. This has resulted in a number of bug reports with test cases that can be used in the testsuite.
Appendix A

Source Code

A.1 Solvers

A.1.1 SolverId

Enumeration type for solvers

```plaintext
within PDESolver.Solver;

type SolverId =
    enumeration(SerialPECE,
                SerialRK32,
                ParallelPECE,
                ParallelRK32);
```

A.1.2 Solve

Interface to all solvers

```plaintext
within PDESolver.Solver;

function Solve
    input Types.Field state[]: "Current state";
    input Types.Field ext[]: "External field";
    input Real var[]: "External variables";
    input Real t0: "Time at 'state'";
    input Real t1: "Time at 'next'";
    input SolverId solverId;
    input Real dt = (t1-t0)*2: "Initial intermediate step length.";
    input Real eMax = 0.1: "Max error";
    input Real hMax = dt: "Max intermediate step length.";
    input Integer th1=1, th2=1, th3=1;
    output Types.Field next[size(state,1)]: "New state";
protected
    constant Integer stateFields = size(state,1);
algorithm
    if solverId == SolverId.SerialPECE then
        next := Serial.PECE(stateFields=stateFields,
```
A.1.3 ParallelRK32

Parallel Runge-Kutta Solver with Adaptive Stepsize

within PDESolver.Solver.Parallel;

function RungeKutta32
    import DerState = PDESolver.Model.ParDerState;
    import PDESolver.Types;
    input Integer stateFields;
    input PDESolver.Types.Field state[] "Current state";
    input PDESolver.Types.Field ext[] "External field";
    input PDESolver.Types.Field var[] "External variables";
    input Real t0 "Time at state";
    input Real t1 "Time at next";
    input Real dt = (t1-t0)*2 "Maximum intermediate step length.";
    input Real eMax;
    input Real hMax;
    input Integer th1, th2, th3;
    output PDESolver.Types.Field next[size(state,1)] "New state";
    output Real newDt;
    output Integer derStateEvaluations;
    output Integer stepsTaken;
protected
    type Fields = PDESolver.Types.Field[stateFields];
    type ExtFields = PDESolver.Types.Field[size(ext,1)];
    type ExtVar = PDESolver.Real[size(var,1)];

    parkernel function SolverKernel
        input Integer stateFields;
parglobal input Fields x0;
parglobal input ExtFields ext;
parglobal input ExtVar var;
    input Real t0;
    input Real t1;
    input Real dt;
    input Real eMax, hMax;
    input Integer th1, th2, th3;
input Integer serialReduceAt = 64;
parglobal output Fields x1;
parglobal output Real newDt;
parglobal output Integer derStateEvaluations[1];
parglobal output Integer stepsTaken[1];
// Temporary variables, but cannot be declared locally.
parglobal output Fields k1;
parglobal output Fields k2;
parglobal output Fields k3;
parglobal output Fields k4;
parglobal output Fields currentState;
parglobal output Fields tmp;
parglobal output Real errorPerThread[th1*th2*th3];

protected
import Rem = PDESolver.ParFunctions.Workarounds.Rem;
Real localError;
Real value1, value2, value3;
Real currentTime;
Real h;
Integer dStateCnt = 0;
Integer stepsCnt = 0;
Integer first1, step1, last1;
Integer first2, step2, last2;
Integer first3, step3, last3;
Boolean group111 = true;
Integer linearId;
Integer linearLast = th1*th2*th3;
Integer reduceRange;
algorithm
    first1 := oclGetLocalId(1);
    step1 := oclGetLocalSize(1);
    last1 := size(x0, 2);
    first2 := oclGetLocalId(2);
    step2 := oclGetLocalSize(2);
    last2 := size(x0, 3);
    first3 := oclGetLocalId(3);
    step3 := oclGetLocalSize(3);
    last3 := size(x0, 4);
    linearId := GetLinearId();

for i in 1:3 loop
    if oclGetGroupId(i) > 1 then
        // Cannot sync between workgroups,
        // so only use the threads from the first group.
        group111 := false;
    end if;
end for;
if group111 then
    for
f in 1:stateFields,
i in first1:step1:last1,
j in first2:step2:last2,
k in first3:step3:last3
loop
    currentState[f, i, j, k] := x0[f, i, j, k];
end for;
end if;
currentTime := t0;
h := min(dt, hMax);
while group111 and currentTime < t1 loop
    // Compute k1
    for
        i in first1:step1:last1,
j in first2:step2:last2,
k in first3:step3:last3
    loop
        (value1, value2, value3) := DerState(state=currentState, ext=ext, var=var,
t=currentTime,
i=i, j=j, k=k);
k1[1,i,j,k] := value1;
tmp[1,i,j,k] := currentState[1,i,j,k] + value1*h/2;
        if stateFields > 1 then
            k1[2,i,j,k] := value2;
tmp[2,i,j,k] := currentState[2,i,j,k] + value2*h/2;
            if stateFields > 2 then
                k1[3,i,j,k] := value3;
tmp[3,i,j,k] := currentState[3,i,j,k] + value3*h/2;
            end if;
        end if;
    end for;
oclGlobalBarrier();
    // Compute k2
    for
        i in first1:step1:last1,
j in first2:step2:last2,
k in first3:step3:last3
    loop
        (value1, value2, value3) := DerState(state=tmp, ext=ext, var=var,
t=currentTime+h/2,
i=i, j=j, k=k);
k2[1,i,j,k] := value1;
        if stateFields > 1 then
            k2[2,i,j,k] := value2;
            if stateFields > 2 then
                k2[3,i,j,k] := value3;
            end if;
        end if;
    end for;
oclGlobalBarrier();
    // Update tmp
    for
        i in first1:step1:last1,
j in first2:step2:last2,
k in first3:step3:last3
\[
\text{loop} \\
\text{tmp}[1,i,j,k] := \text{currentState}[1,i,j,k] + k2[1,i,j,k]*3*h/4; \text{\ if\ stateFields > 1 \ then} \\
\text{tmp}[2,i,j,k] := \text{currentState}[2,i,j,k] + k2[2,i,j,k]*3*h/4; \text{\ if\ stateFields > 2 \ then} \\
\text{tmp}[3,i,j,k] := \text{currentState}[3,i,j,k] + k2[3,i,j,k]*3*h/4; \text{\ end\ if;} \\
\text{end\ for;} \\
\text{oclGlobalBarrier();} \\
\text{//\ Compute\ k3} \\
\text{for} \\
i\ \text{in} \ \text{first1:step1:last1}, \\
j\ \text{in} \ \text{first2:step2:last2}, \\
k\ \text{in} \ \text{first3:step3:last3} \\
\text{loop} \\
(value1, value2, value3) := \text{DerState(state=\text{tmp}, ext=\text{ext}, var=\text{var},} \\
t=\text{currentTime+3*h/4,} \\
i=i, \ j=j, \ k=k); \\
k3[1,i,j,k] := value1; \text{\ if\ stateFields > 1 \ then} \\
k3[2,i,j,k] := value2; \text{\ if\ stateFields > 2 \ then} \\
k3[3,i,j,k] := value3; \text{\ end\ if;} \\
\text{end\ if;} \\
\text{end\ for;} \\
\text{oclGlobalBarrier();} \\
\text{//\ Update\ x1\ with\ possible\ solution} \\
\text{for} \\
i\ \text{in} \ \text{first1:step1:last1}, \\
j\ \text{in} \ \text{first2:step2:last2}, \\
k\ \text{in} \ \text{first3:step3:last3} \\
\text{loop} \\
x1[1,i,j,k] := \text{currentState}[1,i,j,k] + (k1[1,i,j,k]*2/9 + k2[1,i,j,k]/3 + k3[1,i,j,k]*4/9)*h; \text{\ if\ stateFields > 1 \ then} \\
x1[2,i,j,k] := \text{currentState}[2,i,j,k] + (k1[2,i,j,k]*2/9 + k2[2,i,j,k]/3 + k3[2,i,j,k]*4/9)*h; \text{\ if\ stateFields > 2 \ then} \\
x1[3,i,j,k] := \text{currentState}[3,i,j,k] + (k1[3,i,j,k]*2/9 + k2[3,i,j,k]/3 + k3[3,i,j,k]*4/9)*h; \text{\ end\ if;} \\
\text{end\ if;} \\
\text{//x1[1,i,j,k] := k3[1,i,j,k];} \\
\text{end\ for;} \\
\text{oclGlobalBarrier();}
// Compute k4
for i in first1:step1:last1,
    j in first2:step2:last2,
    k in first3:step3:last3
loop
  (value1, value2, value3) :=
    DerState(state=x1, ext=ext, var=var,
    t=currentTime+h, i=i, j=j, k=k);
  k4[1,i,j,k] := value1;
  if stateFields > 1 then
    k4[2,i,j,k] := value2;
    if stateFields > 2 then
      k4[3,i,j,k] := value3;
    end if;
  end if;
end for;
// Once here, four more calls to DerState have been done over the field.
dStateCnt := dStateCnt + 4;
// Estimate maximum error
localError := 0;
for i in first1:step1:last1,
    j in first2:step2:last2,
    k in first3:step3:last3
loop
  localError :=
    max(localError,
        abs(5/72*k1[1,i,j,k] - 1/12*k2[1,i,j,k]
            - 1/3*k3[1,i,j,k] + 1/8*k4[1,i,j,k])*h);
  if stateFields > 1 then
    localError :=
      max(localError,
          abs(5/72*k1[2,i,j,k] - 1/12*k2[2,i,j,k]
            - 1/3*k3[2,i,j,k] + 1/8*k4[2,i,j,k])*h);
  if stateFields > 2 then
    localError :=
      max(localError,
          abs(5/72*k1[3,i,j,k] - 1/12*k2[3,i,j,k]
            - 1/3*k3[3,i,j,k] + 1/8*k4[3,i,j,k])*h);
  end if;
end if;
end for;
// Reduce maximum error to local thread 1,1,1.
errorPerThread[linearId] := localError;
oclGlobalBarrier();
reduceRange := 1;
while reduceRange*serialReduceAt < linearLast loop
  if Rem(linearId - 1, reduceRange) == 0 and
    linearId + reduceRange <= linearLast
  then
    errorPerThread[linearId] :=
      max(errorPerThread[linearId],
          errorPerThread[linearId + reduceRange]);
  end if;
end if;
oclGlobalBarrier();

reduceRange := reduceRange * 2;
end while;
if reduceRange < linearLast then
  if linearId == 1 then
    localError := errorPerThread[1];
    for i in 1+reduceRange:reduceRange/2:linearLast loop
      localError := max(localError, errorPerThread[i]);
    end for;
    errorPerThread[1] := localError;
  end if;
  oclGlobalBarrier();
end if;
// All threads evaluate the error and update stepsize
localError := errorPerThread[1];
if localError < eMax then
  currentTime := currentTime + h;
  for f in 1:stateFields,
      i in first1:step1:last1,
      j in first2:step2:last2,
      k in first3:step3:last3 loop
    currentState[f, i, j, k] := x1[f, i, j, k];
  end for;
  stepsCnt := stepsCnt + 1;
end if;
if localError > eMax then
  h := 0.5 * h; // * sqrt(1e-3/localError);
else
  h := 2 * h; // * sqrt(1e3);
end if;
if h < hMax
  h := min(h, t1 - currentTime);
end while;
if oclGetLocalId(3) == 1 and oclGetLocalId(2) == 1 and oclGetLocalId(1) == 1 then
  // Only let one thread set the scalar outputs.
derStateEvaluations[1] := dStateCnt;
stepsTaken[1] := stepsCnt;
end if;
end SolverKernel;

par global Types.Field pExt[size(ext,1)];
par global Real pVar[size(var,1)];
par global Fields k1;
par global Fields k2;
par global Fields k3;
par global Fields k4;
par global Fields currentState;
par global Fields xNext;
parglobal Fields tmp1;
parglobal Fields tmp2;
//Real newDt;
parglobal Integer pderStateEvaluationsA[1];
parglobal Integer pstepsTakenA[1];
Integer derStateEvaluationsA[1];
Integer stepsTakenA[1];
parglobal Real errorPerThread[th1*th2*th3];
Real e[th1, th2, th3];
Real maxError;
Fields hostTmp;
Fields hostxNext;

algorithm
currentState := state;
pExt := ext;
pVar := var;
oclSetNumThreadsGlobalLocal3D({th1,th2,th3}, {th1,th2,th3});
(currentState, newDt, pderStateEvaluationsA, pstepsTakenA, 
k1, k2, k3, k4, tmp1, tmp2, 
errorPerThread) := SolverKernel(stateFields=stateFields, 
x0=currentState, ext=pExt, var=pVar, 
eMax=eMax, hMax=hMax, 
t0=t0, t1=t1, dt=dt, 
th1=th1, th2=th2, th3=th3);

stepsTakenA := pstepsTakenA;
derStateEvaluationsA := pderStateEvaluationsA;
// e := errorPerThread;
// print("\n");
// for i,j,k loop
// print(" " + String(e[i,j,k]));
// end for;
print("\n");
print("Steps taken: " + String(stepsTakenA[1]) + "\n");
print("DerState evaluations: " + String(derStateEvaluationsA[1]) + "\n");
print("Successful steps: " + String(stepsTakenA[1]*400/derStateEvaluationsA[1]) + "\n");
next := currentState;
end RungeKutta32;

A.1.4 ParallelPECE
Parallel Predictor-Corrector Solver with Adaptive Stepsize
within PDESolver.Solver.Parallel;

function PECE
import DerState = PDESolver.Model.ParDerState;
import PDESolver.Types;
input Integer stateFields;
input Types.Field state[:] "Current state";
input Types.Field ext[:] "External field";
input Real var[:] "External variables";
input Real t0 "Time at state";
input Real t1 "Time at next";
input Real dt = (t1-t0)*2 "Maximum intermediate step length.";
input Real eMax;
input Real hMax;
input Integer th1, th2, th3;
output Types.Field next[size(state,1)] "New state";
output Real newDt;
output Integer derStateEvaluations;
output Integer stepsTaken;

protected
type Fields = Types.Field[stateFields];
type ExtFields = Types.Field[size(ext,1)];
type ExtVar = Real[size(var,1)];

parkernel function SolverKernel
input Integer stateFields;
parglobal input Fields x0;
parglobal input ExtFields ext;
parglobal input ExtVar var;
input Real t0;
input Real t1;
input Real dt;
input Real eMax, hMax;
input Integer th1, th2, th3;
input Integer maxIterations = 10;
input Integer serialReduceAt = 64;
parglobal output Fields x1;
parglobal output Real newDt;
parglobal output Integer derStateEvaluations[1];
parglobal output Integer stepsTaken[1];
// Temporary variables, but cannot be declared locally.
parglobal output Fields derCurrentState;
parglobal output Fields x1Prev;
parglobal output Fields currentState;
parglobal output Real errorPerThread[th1*th2*th3];

protected
import Rem = PDESolver.ParFunctions.Workarounds.Rem;
Real localError;
Real value1, value2, value3;
Real currentTime;
Real h;
Integer dStateCnt = 0;
Integer stepsCnt = 0;
Integer first1, step1, last1;
Integer first2, step2, last2;
Integer first3, step3, last3;
Boolean group111 = true;
Integer linearId;
Integer linearLast = th1*th2*th3;
Integer iterationsDone;
Integer reduceRange;

algorithm
first1 := oclGetLocalId(1);
step1 := oclGetLocalSize(1);
last1 := size(x0, 2);
first2 := oclGetLocalId(2);
step2 := oclGetLocalSize(2);
last2 := size(x0, 3);
first3 := oclGetLocalId(3);
step3 := oclGetLocalSize(3);
last3 := size(x0, 4);
linearId := GetLinearId();
for i in 1:3 loop
  if oclGetGroupId(i) > 1 then
    // Cannot sync between workgroups,
    // so only use the threads from the first group.
    group111 := false;
  end if;
end for;
if group111 then
  for f in 1:stateFields,
      i in first1:step1:last1,
      j in first2:step2:last2,
      k in first3:step3:last3
    loop
    currentState[f, i, j, k] := x0[f, i, j, k];
  end for;
end if;
currentTime := t0;
h := min(dt, hMax);
while group111 and currentTime < t1 loop
  // Compute derCurrentState and initial x1Prev
  dStateCnt := dStateCnt + 1;
  for i in first1:step1:last1,
      j in first2:step2:last2,
      k in first3:step3:last3
    loop
      (value1, value2, value3) := DerState(state=currentState, ext=ext, var=var,
              t=currentTime,
              i=i, j=j, k=k);
      derCurrentState[1,i,j,k] := value1;
      x1Prev[1,i,j,k] := currentState[1,i,j,k] + value1*h;
      if stateFields > 1 then
        derCurrentState[2,i,j,k] := value2;
        x1Prev[2,i,j,k] := currentState[2,i,j,k] + value2*h;
      end if
      if stateFields > 2 then
        derCurrentState[3,i,j,k] := value3;
        x1Prev[3,i,j,k] := currentState[3,i,j,k] + value3*h;
      end if
    end for;
  end for;
  oclGlobalBarrier();
  // Set localError to a value big enough to enter the loop
  localError := eMax*2 + 1;
  // Iterate to find a value of x1
  iterationsDone := 0;
  while localError > eMax and iterationsDone < maxIterations loop
    // Compute x1 and localError
    localError := 0;
    dStateCnt := dStateCnt + 1;
    for i in first1:step1:last1,
j in first2:step2:last2,
k in first3:step3:last3
loop
  (value1, value2, value3) :=
    DerState(state=x1Prev, ext=ext, var=var,
        t=currentTime,
        i=i, j=j, k=k);
x1[1, i, j, k] :=
  currentState[1, i, j, k] + h/2*(derCurrentState[1, i, j, k] + value1);
localError := max(localError, abs(x1[1, i, j, k] - x1Prev[1, i, j, k]));
if stateFields > 1 then
  x1[2, i, j, k] :=
    currentState[2, i, j, k] + h/2*(derCurrentState[2, i, j, k] + value2);
  localError := max(localError, abs(x1[2, i, j, k] - x1Prev[2, i, j, k]));
  if stateFields > 2 then
    x1[3, i, j, k] :=
      currentState[3, i, j, k] + h/2*(derCurrentState[3, i, j, k] + value3);
      localError := max(localError, abs(x1[3, i, j, k] - x1Prev[3, i, j, k]));
  end if;
end if;
end for;
// Reduce maximum error to local thread 1,1,1.
errorPerThread[linearId] := localError;
occlLocalBarrier();
reduceRange := 1;
while reduceRange*serialReduceAt < linearLast loop
  if
    Rem(linearId - 1, reduceRange) == 0 and
    linearId + reduceRange <= linearLast
  then
    errorPerThread[linearId] :=
      max(errorPerThread[linearId],
          errorPerThread[linearId + reduceRange]);
  end if;
occlGlobalBarrier();
  reduceRange := reduceRange * 2;
end while;
if reduceRange < linearLast then
  if linearId == 1 then
    localError := errorPerThread[1];
    for i in 1+reduceRange:reduceRange/2:linearLast loop
      localError := max(localError, errorPerThread[i]);
    end for;
  end if;
occlGlobalBarrier();
end if;
localError := errorPerThread[1];
iterationsDone := iterationsDone + 1;
if localError <= eMax then
  currentTime := iterationsDone + h;
  stepsCnt := stepsCnt + 1;
  for
    f in 1:stateFields,
      i in first1:step1:last1,
      j in first2:step2:last2,
k in first3:step3:last3
loop
    currentState[f, i, j, k] := x1[f, i, j, k];
end for;
end if;
if iterationsDone >= maxIterations then
    h := h / 2;
elseif iterationsDone == 1 then
    h := 1.5 * h;
end if;
h := min(h, t1-currentTime);
end while;  // One intermediate step
end while;  // All steps to t1
if
    oclGetLocalId(3) == 1 and
    oclGetLocalId(2) == 1 and
    oclGetLocalId(1) == 1
then
    // Only let one thread set the scalar outputs.
    derStateEvaluations[1] := dStateCnt;
    stepsTaken[1] := stepsCnt;
end if;
end SolverKernel;
parglobal Types.Field pExt[size(ext,1)];
parglobal Real pVar[size(var,1)];
parglobal Fields derCurrentState;
parglobal Fields x1Prev;
parglobal Fields currentState;
parglobal Fields xNext;
parglobal Fields tmp1;
parglobal Fields tmp2;
//Real newDt;
parglobal Integer pderStateEvaluationsA[1];
parglobal Integer pstepsTakenA[1];
Integer derStateEvaluationsA[1];
Integer stepsTakenA[1];
parglobal Real errorPerThread[th1*th2*th3];
Real e[th1*th2*th3];
Real maxError;
Fields hostTmp;
Fields hostxNext;
algorithim
    currentState := state;
pExt := ext;
pVar := var;
oclSetNumThreadsGlobalLocal3D({th1,th2,th3}, {th1,th2,th3});
(currentState, newDt, pderStateEvaluationsA, pstepsTakenA,
derCurrentState, x1Prev, currentState,
errorPerThread) := SolverKernel(stateFields=stateFields,
x0=currentState, ext=pExt, var=pVar,
eMax=eMax, hMax=hMax,
t0=t0, t1=t1, dt=dt,
thl=thl, th2=th2, th3=th3);
stepsTakenA := pstepsTakenA;
derStateEvaluationsA := pderStateEvaluationsA;
// e := errorPerThread;
A.1. Solvers

// print("\n");
// for i loop
// print(" " + String(e[i]));
// end for;
print("\n");
next := currentState;
end PECE;

A.1.5 SerialRK32

Serial Runge-Kutta Solver with Adaptive Stepsize

within PDESolver.Solver.Serial;

function RungeKutta32
import DerState = PDESolver.Model.DerState;
import PDESolver.Types;
input Integer stateFields;
input Types.Field state[] "Current state";
input Types.Field ext[] "External field";
input Real var[] "External variables";
input Real t0 "Time at 'state'";
input Real t1 "Time at 'next'";
input Real dt = (t1-t0)*2 "Maximum intermediate step length.";
input Real eMax;
input Real hMax;
output Types.Field next[size(state,1)] "New state";
output Real newDt;
output Integer derStateEvaluations;
output Integer stepsTaken;
protected
Type Fields = Types.Field[stateFields];
Type ExtFields = Types.Field[size(ext,1)];
Type ExtVar = Real[size(var,1)];

function SolverKernel
input Integer stateFields;
input Fields x0;
input ExtFields ext;
input ExtVar var;
input Real t0;
input Real t1;
input Real dt;
input Real eMax, hMax;
output Fields x1;
output Real newDt;
output Integer derStateEvaluations;
output Integer stepsTaken;
protected
Fields k1;
Fields k2;
Fields k3;
Fields k4;
Fields currentState;
Fields tmp;
Real localError;
Real value1, value2, value3;
Real currentTime;
Real h;
Integer dStateCnt = 0;
Integer stepsCnt = 0;

algorithm
currentState := x0;
currentTime := t0;
h := min(dt, hMax);
while currentTime < t1 loop
  // Compute k1
  for i in 1:size(x0,2),
     j in 1:size(x0,3),
     k in 1:size(x0,4)
  loop
    (value1, value2, value3) :=
      DerState(state=currentState, ext=ext, var=var,
               t=currentTime,
               i=i, j=j, k=k);
    k1[1,i,j,k] := value1;
    tmp[1,i,j,k] := currentState[1,i,j,k] + value1*h/2;
    if stateFields > 1 then
      k1[2,i,j,k] := value2;
      tmp[2,i,j,k] := currentState[2,i,j,k] + value2*h/2;
      if stateFields > 2 then
        k1[3,i,j,k] := value3;
        tmp[3,i,j,k] := currentState[3,i,j,k] + value3*h/2;
      end if;
    end if;
  end for;
  // Compute k2
  for i in 1:size(x0,2),
     j in 1:size(x0,3),
     k in 1:size(x0,4)
  loop
    (value1, value2, value3) :=
      DerState(state=tmp, ext=ext, var=var,
               t=currentTime+h/2,
               i=i, j=j, k=k);
    k2[1,i,j,k] := value1;
    if stateFields > 1 then
      k2[2,i,j,k] := value2;
      if stateFields > 2 then
        k2[3,i,j,k] := value3;
      end if;
    end if;
  end for;
  // Update tmp
  for i in 1:size(x0,2),
     j in 1:size(x0,3),
     k in 1:size(x0,4)
  loop
tmp[1,i,j,k] := currentState[1,i,j,k] + k2[1,i,j,k]*3*h/4;
if stateFields > 1 then
  tmp[2,i,j,k] := currentState[2,i,j,k] + k2[2,i,j,k]*3*h/4;
  if stateFields > 2 then
    tmp[3,i,j,k] := currentState[3,i,j,k] + k2[3,i,j,k]*3*h/4;
    end if;
  end if;
end if;
end for;

// Compute k3
for i in 1:size(x0,2),
  j in 1:size(x0,3),
  k in 1:size(x0,4)
loop
  (value1, value2, value3) :=
    DerState(state=tmp, ext=ext, var=var,
      t=currentTime+3*h/4,
      i=i, j=j, k=k);
  k3[1,i,j,k] := value1;
  if stateFields > 1 then
    k3[2,i,j,k] := value2;
    if stateFields > 2 then
      k3[3,i,j,k] := value3;
    end if;
  end if;
end for;

// Update x1 with possible solution
for i in 1:size(x0,2),
  j in 1:size(x0,3),
  k in 1:size(x0,4)
loop
  x1[1,i,j,k] :=
    currentState[1,i,j,k]
    + (k1[1,i,j,k]*2/9
        + k2[1,i,j,k]/3
        + k3[1,i,j,k]*4/9)*h;
  if stateFields > 1 then
    x1[2,i,j,k] :=
      currentState[2,i,j,k]
      + (k1[2,i,j,k]*2/9
          + k2[2,i,j,k]/3
          + k3[2,i,j,k]*4/9)*h;
    if stateFields > 2 then
      x1[3,i,j,k] :=
        currentState[3,i,j,k]
        + (k1[3,i,j,k]*2/9
            + k2[3,i,j,k]/3
            + k3[3,i,j,k]*4/9)*h;
    end if;
  end if;
end for;

// Compute k4
for i in 1:size(x0,2),
  j in 1:size(x0,3),
\textbf{k in 1:size(x0,4)}

\textbf{loop}
\begin{verbatim}
    (value1, value2, value3) :=
        DerState(state=x1, ext=ext, var=var,
            t=currentTime+h,
            i=i, j=j, k=k);
    k4[1,i,j,k] := value1;
    if stateFields > 1 then
        k4[2,i,j,k] := value2;
    end if;
    if stateFields > 2 then
        k4[3,i,j,k] := value3;
    end if;
end for;
\end{verbatim}

// Once here, four more calls to DerState have been done over the field.

\textbf{dStateCnt := dStateCnt + 4;}

// Estimate maximum error

\textbf{localError := 0;}
\textbf{for i, j, k loop}
\begin{verbatim}
    localError :=
        max(localError,
            \text{abs}(5/72*k1[1,i,j,k] - 1/12*k2[1,i,j,k]
                - 1/3*k3[1,i,j,k] + 1/8*k4[1,i,j,k])*h);
    if stateFields > 1 then
        localError :=
            max(localError,
                \text{abs}(5/72*k1[2,i,j,k] - 1/12*k2[2,i,j,k]
                    - 1/3*k3[2,i,j,k] + 1/8*k4[2,i,j,k])*h);
    end if;
end for;
if localError < eMax then
    currentTime := currentTime + h;
    for f in 1:stateFields,
        i in 1:size(x0,2),
        j in 1:size(x0,3),
        k in 1:size(x0,4)
        loop
            currentState[f, i, j, k] := x1[f, i, j, k];
        end for;
    stepsCnt := stepsCnt + 1;
end if;
if localError > eMax then
    h := 0.5 * h; // * sqrt(1e-3/localError);
else
    h := 2 * h; // * sqrt(1e3);
end if;
\textbf{h := min(h, hMax);}
\textbf{h := min(h, t1 - currentTime);}
\textbf{end while;}
\textbf{derStateEvaluations := dStateCnt;
A.1. Solvers

stepsTaken := stepsCnt;

end SolverKernel;

Types.Field pExt[size(ext,1)];
Real pVar[size(var,1)];
Fields k1;
Fields k2;
Fields k3;
Fields k4;
Fields currentState;
Fields xNext;
Fields tmp1;
Fields tmp2;
//Real newDt;
Real maxError;
Fields hostTmp;
Fields hostxNext;
algorithm
currentState := state;
(currentState, newDt, derStateEvaluations, stepsTaken) :=
SolverKernel(stateFields=stateFields,
x0=currentState, ext=ext, var=var,
eMax=eMax, hMax=hMax,
t0=t0, t1=t1, dt=dt);

print("\n");
print("Steps taken: " + String(stepsTaken) + "\n");
print("DerState evaluations: " + String(derStateEvaluations) + "\n");
print("Successful steps: " + String(stepsTaken*400/derStateEvaluations) + "%\n");
next := currentState;
end RungeKutta32;

A.1.6 SerialPECE
Serial Predictor-Corrector Solver with Adaptive Stepsize
within PDESolver.Solver.Serial;

function PECE
import DerState = PDESolver.Model.DerState;
import PDESolver.Types;
input Integer stateFields;
input Types.Field state[:]: "Current state";
input Types.Field ext[:]: "External field";
input Real var[:]: "External variables";
input Real t0 "Time at 'state'";
input Real t1 "Time at 'next'";
input Real dt = (t1-t0)*2 "Maximum intermediate step length.";
input Real eMax;
input Real hMax;
output Types.Field next[size(state,1)]: "New state";
output Real newDt;
output Integer derStateEvaluations;
output Integer stepsTaken;
protected
type Fields = Types.Field[stateFields];
type ExtFields = Types.Field[size(ext,1)];
type ExtVar = Real[size(var,1)];

function SolverKernel
  input Integer stateFields;
  input Fields x0;
  input ExtFields ext;
  input ExtVar var;
  input Real t0;
  input Real t1;
  input Real dt;
  input Real eMax, hMax;
  input Integer maxIterations = 10;
  output Fields x1;
  output Real newDt;
  output Integer derStateEvaluations;
  output Integer stepsTaken;
protected
  import Rem = PDESolver.ParFunctions.Workarounds.Rem;
  Fields derCurrentState;
  Fields x1Prev;
  Fields currentState;
  Real localError;
  Real value1, value2, value3;
  Real currentTime;
  Real h;
  Integer dStateCnt = 0;
  Integer stepsCnt = 0;
  Integer iterationsDone;
  Fields derCurrentState;
  Fields x1Prev;
  Fields currentState;
algorithm
  currentState := x0;
  currentTime := t0;
  h := min(dt, hMax);
  while currentTime < t1 loop
    // Compute derCurrentState and initial x1Prev
    dStateCnt := dStateCnt + 1;
    for i in 1:size(x0,2),
        j in 1:size(x0,3),
        k in 1:size(x0,4)
      loop
        (value1, value2, value3) :=
          DerState(state=currentState, ext=ext, var=var,
                   t=currentTime,
                   i=i, j=j, k=k);
        derCurrentState[1,i,j,k] := value1;
        x1Prev[1,i,j,k] := currentState[1,i,j,k] + value1*h;
        if stateFields > 1 then
          derCurrentState[2,i,j,k] := value2;
          x1Prev[2,i,j,k] := currentState[2,i,j,k] + value2*h;
        if stateFields > 2 then
          derCurrentState[3,i,j,k] := value3;
          x1Prev[3,i,j,k] := currentState[3,i,j,k] + value3*h;
        end if;
      end if;
  end if;
end for;
// Set localError to a value big enough to enter the loop
localError := eMax*2 + 1;
// Iterate to find a value of x1
iterationsDone := 0;
while
  localError > eMax and
  iterationsDone < maxIterations
loop
  // Compute x1 and localError
  localError := 0;
  dStateCnt := dStateCnt + 1;
  for
    i in 1:size(x0,2),
    j in 1:size(x0,3),
    k in 1:size(x0,4)
  loop
    (value1, value2, value3) :=
      DerState(state=x1Prev, ext=ext, var=var,
      t=currentTime,
      i=i, j=j, k=k);
    x1[1,i,j,k] :=
      currentState[1,i,j,k] + h/2*(derCurrentState[1,i,j,k] + value1);
    localError := max(localError, abs(x1[1,i,j,k] - x1Prev[1,i,j,k]));
    if stateFields > 1 then
      x1[2,i,j,k] :=
        currentState[2,i,j,k] + h/2*(derCurrentState[2,i,j,k] + value2);
      localError := max(localError, abs(x1[2,i,j,k] - x1Prev[2,i,j,k]));
      if stateFields > 2 then
        x1[3,i,j,k] :=
          currentState[3,i,j,k] + h/2*(derCurrentState[3,i,j,k] + value3);
        localError := max(localError, abs(x1[3,i,j,k] - x1Prev[3,i,j,k]));
      end if;
    end if;
  end for;
  iterationsDone := iterationsDone + 1;
  if localError <= eMax then
    currentTime := currentTime + h;
    stepsCnt := stepsCnt + 1;
    currentState := x1;
  end if;
  if iterationsDone >= maxIterations then
    h := h / 2;
  elseif iterationsDone == 1 then
    h := 1.5 * h;
  end if;
  h := min(h, t1-currentTime);
end while; // One intermediate step
end while; // All steps to t1
derStateEvaluations := dStateCnt;
stepsTaken := stepsCnt;
end SolverKernel;

// Real newDt;
Real maxError;
algorithm
  (next, newDt, derStateEvaluations, stepsTaken) :=
SolverKernel(stateFields=stateFields,
    x0=state, ext=ext, var=var,
    eMax=eMax, hMax=hMax,
    t0=t0, t1=t1, dt=dt);

print("\n");
print("Steps taken: " + String(stepsTaken) + "\n");
print("DerState evaluations: " + String(derStateEvaluations) + "\n");
// print("Successful steps: " + String(stepsTakenA*400/derStateEvaluationsA[1])
end PECE;

A.2 Utilities

A.2.1 Types

Types

within PDESolver;

package Types
// type Dimension = enumeration(x, y, z);
type Boundery = enumeration(zero, wrap, only, extend);

constant Integer wMax = 3;

record Stencil
    Integer width[3];
    Integer offset[3];
    Real weight[wMax,wMax,wMax];
end Stencil;

type Field = Real[Model.Settings.n[1],
    Model.Settings.n[2],
    Model.Settings.n[3]]
    (each start=0, each fixed=true);

record Domain
    // Range range[3];
    // Range range1;
    // Range range2;
    // Range range3;
end Domain;

// record Range
//    Real first(start=0, fixed=true);
//    Real last(start=0, fixed=true);
//    end Range;

type Coordinate = Real[3];
type FieldIndex = Integer[3];

record FieldIndexRange
    FieldIndex first;
    FieldIndex last;
    FieldIndex step;
end FieldIndexRange;

type DomainPart = enumeration
    (BoundaryFirst, BoundaryLast, Interior, Domain);
end Types;

A.2.2 Stencils

Useful stencils and functions operating on stencils

within PDESolver;

package Stencils "Useful stencils"

constant Types.Stencil Null (offset={0,0,0},
    width={0,0,0},
    weight=fill(0.0,3,3,3))

"Null stencil. Used as default value.");

custom
    Types.Stencil PderCenterDx =
        array2stencil(x={-0.5,0,0.5}, y={1}, z={1}, offset={2,1,1})
    "Central derivative along the x-axis. Divide by h before usage.");

custom
    Types.Stencil PderForwardDx =
        array2stencil(x={-1,1}, y={1}, z={1}, offset={1,1,1})
    "Forward derivative along the x-axis. Divide by h before usage.");

custom
    Types.Stencil Pder2CenterDx2 =
        array2stencil(x={1,-2,1}, y={1}, z={1}, offset={2,1,1})
    "Second order central derivative along the x-axis.
    Divide by h^2 before usage.");

function array2stencil
    input Real x[:];
    input Real y[:];
    input Real z[:];
    input Integer offset[3];
    output Types.Stencil result;
protected
    constant Integer wMax = Types.Stencil.wMax;
algorithm
    result.width := {size(x,1),size(y,1),size(z,1)};
    result.offset := offset;
    for
        i in 1:result.width[1],
        j in 1:result.width[2],
        k in 1:result.width[3]
    loop
        result.weight[i,j,k] := x[i]*y[j]*z[k];
        end for;
    end array2stencil;

function ijk2jki "Rotate a stencil one step in the coordinate system."
    input Types.Stencil src;
    output Types.Stencil dst (weight=fill(0.0,
        size(src.weight, 2),
        size(src.weight, 3),
        size(src.weight, 1)));
algorithm
    dst.offset[2] := src.offset[1];
dst.offset[1] := src.offset[3];
for i,j,k loop
  dst.weight[j,k,i] := src.weight[i,j,k];
  end for;
end ijk2jki;

function mirror_x
  input Types.Stencil src;
  output Types.Stencil dst;
algorithm
end mirror_x;

function Mult
  input Types.Stencil src;
  input Real factor;
  output Types.Stencil dst;
algorithm
  dst.width := src.width;
  dst.offset := src.offset;
  dst.weight := factor * src.weight;
end Mult;
end Stencils;

A.2.3 ijk2cord
Convert an array index into coordinates for the model.

within PDESolver.Functions;

function ijk2cord
  input Types.Domain domain;
  input Types.FieldIndex index;
  output PDESolver.Types.Coordinate coordinate;
protected
  import PDESolver.Model.Settings;
  function i2c
    input Integer dim;
    input Integer idx;
    output Real c;
  protected
    Real first = Settings.first[dim];
    Real last = Settings.last[dim];
    Integer n = Settings.n[dim];
  algorithm
    if idx == 1 or n[dim] == 1 then
      c := first;
    elseif idx == n[dim] then
      c := last;
    else
      c := first + (last - first)*(idx-1)/(domain.n[dim]-1);
    end if;
  end i2c;
algorithm
  for i loop
    coordinate[i] := i2c(dim=i, idx=index[i]);
  end for;
end ijk2cord;
A.2.4 CoordinateFunction

Partial function to describe a field at certain time. Passed as argument to ApplyCoordinateFunction Appendix A.2.5

```plaintext
within PDESolver.Functions;
partial function CoordinateFunction
  input Types.Domain domain;
  input Types.Coordinate c;
  input Real arg[:];
  output Real result;
end CoordinateFunction;
```

A.2.5 ApplyCoordinateFunction

Function to create a field based on the values of a function.

```plaintext
within PDESolver.Functions;

function ApplyCoordinateFunction
  import PDESolver.Types;
  input CoordinateFunction func;
  input Real arg[:]=fill(0,0);
  output Types.Field f;
  protected
    Types.FieldIndexRange r;
  algorithm
    r := GetIterationRange(part=Types.DomainPart.Domain);
    for i in r.first[1]:r.step[1]:r.last[1] loop
      for j in r.first[2]:r.step[2]:r.last[2] loop
        for k in r.first[3]:r.step[3]:r.last[3] loop
          f[i,j,k] := func(domain, c=ijk2cord(domain, {i,j,k}), arg=arg);
        end for;
      end for;
    end for;
end ApplyCoordinateFunction;
```

A.3 Models

A.3.1 HeatInPlaneDerState

Heat in Plane State Derivative

```plaintext
within PDESolver.Model;

parallel function ParDerState "Calculate the state derivative"
  import Functions = PDESolver.ParFunctions;
  import PDESolver.ParFunctions.Pder;
  import PDESolver.Types;

  input Types.Field[1] state "Temperture";
  input Real var[:];
  input Types.Field ext[:];
  input Real t "Time to calculate the state derivative at";
  input Integer i,j,k "Discrete coordinate within field";
```
output Real value1;
output Real value2;
output Real value3;
protected
// import Model.Settings;
parallel function BoxZeroOne
  input Integer i, j;
  output Integer result;
protected
  Integer idx;
algorithm
  idx := 
    Functions.Workarounds.Div(i-1, Settings.boxSize) + 
    Functions.Workarounds.Div(j-1, Settings.boxSize);
  result := 
    Functions.Workarounds.Rem(idx, 2);
end BoxZeroOne;

Real d2Tdx2, d2Tdy2;
constant Real c = 1;
constant Real q = 1;
constant Real h = 1;
constant Real T_ext = 20;
Real nDer;
Real boundaryT;
algorithm
  if Functions.AtBoundary(i, j, k) then
    if Settings.myBoundary == 1 then
      nDer := 0; // Perfect insulation
      d2Tdx2 := Pder.Pder2Neumann(f=state, fi=1,
        i=i, j=j, k=k,
        dim=1, nder=nDer);
      d2Tdy2 := Pder.Pder2Neumann(f=state, fi=1,
        i=i, j=j, k=k,
        dim=2, nder=nDer);
    elseif Settings.myBoundary == 2 then
      // Heat is added or removed at constant rate depending on // location.
      nDer := 100*(BoxZeroOne(i,j) - 0.5);
      d2Tdx2 := Pder.Pder2Neumann(f=state, fi=1,
        i=i, j=j, k=k,
        dim=1, nder=nDer);
      d2Tdy2 := Pder.Pder2Neumann(f=state, fi=1,
        i=i, j=j, k=k,
        dim=2, nder=nDer);
    elseif Settings.myBoundary == 3 then
      // Heat is added or removed at constant rate depending on // location, and state at the location.
      nDer := 100*BoxZeroOne(i,j) - state[1,i,j,1];
      d2Tdx2 := Pder.Pder2Neumann(f=state, fi=1,
        i=i, j=j, k=k,
        dim=1, nder=nDer);
      d2Tdy2 := Pder.Pder2Neumann(f=state, fi=1,
        i=i, j=j, k=k,
        dim=2, nder=nDer);
    elseif Settings.myBoundary == 4 then
      // Boundary have constant heat depending on location.

A.3. Models

boundaryT := 100*BoxZeroOne(i,j);
d2Tdx2 := Pder.Pder2Dirichlet(f=state, fi=1, i=i, j=j, k=k, dim=1, boundary=boundaryT);
d2Tdy2 := Pder.Pder2Dirichlet(f=state, fi=1, i=i, j=j, k=k, dim=2, boundary=boundaryT);

elseif Settings.myBoundary == 5 then
  // Top and bottom boundary have constant heat depending on location, while the sides have periodic boundary.
  boundaryT := 100*BoxZeroOne(i,j);
d2Tdx2 := Pder.Pder2Periodic(f=state, fi=1, i=i, j=j, k=k, dim=1);
d2Tdy2 := Pder.Pder2Dirichlet(f=state, fi=1, i=i, j=j, k=k, dim=2, boundary=boundaryT);
else
  // Set the boundaries according to the example in Peter Fritzson’s book.
  if Functions.AtFirst(i=i,j=j,k=k,dim=1) then // Left side
    d2Tdx2 := Pder.Pder2Dirichlet(f=state, fi=1, i=i, j=j, k=k, dim=1, boundary=50);
  elseif Functions.AtLast(i=i,j=j,k=k,dim=1) then // Right side
    nDer := q + h*(T_ext - state[1,i,j,1]);
    d2Tdx2 := Pder.Pder2Neumann(f=state, fi=1, i=i, j=j, k=k, dim=1, nder=nDer);
  end if;
d2Tdy2 := Pder.Pder2Neumann(f=state, fi=1, i=i, j=j, k=k, dim=2, nder=0);
end if;
d2Tdx2 := Pder.Pder2Inner(f=state, fi=1, i=i, j=j, k=k, dim=1);
d2Tdy2 := Pder.Pder2Inner(f=state, fi=1, i=i, j=j, k=k, dim=2);
endif;
value1 := c*(d2Tdx2 + d2Tdy2);
end ParDerState;

A.3.2 HeatInPlaneSettings

Heat in Plane Settings

within PDESolver.Model;

package Settings

// Parameters used by the solver
constant Types.FieldIndex n = {80,40,1};
constant Types.Coordinate first = {0,0,0};
constant Types.Coordinate last = {2,1,0};
constant Integer stateFields = 1 "T";
// Parameters used in the model
constant Integer boxSize = integer(n[2]/2);
constant Integer myBoundary = 3;
end Settings;

A.3.3 HeatInRodDerState
Heat in Rod State Derivative

within PDESolver.Model;

parallel function ParDerState "Calculate the state derivative"
  import Functions = PDESolver.ParFunctions;
  import PDESolver.ParFunctions.Pder;
  import PDESolver.Types;

  input Types.Field[1] state "Temperture";
  input Real var[:];
  input Types.Field ext[:];
  input Real t "Time to calculate the state derivative at";
  input Integer i,j,k;
  output Real value1;
  output Real value2;
  output Real value3;
protected
  import Model.Settings;
  Real d2Tdx2, d2Tdy2, d2Td2z2;
  constant Integer nx = Settings.n[1];
  constant Integer ny = Settings.n[2];
  constant Integer nz = Settings.n[3];
  // Iron
  constant Real lambda = 82 "Thermal conductivity";
  constant Real cv = 449 "Heat capacity";
  constant Real ra = 7.87 "Dencity";
  constant Real kappa = 1; //lambda/(cv*ra) "Thermal conduction capacity";
  constant Real lambdaExt = 10;
  constant Real cvExt = 500;
  constant Real raExt = 4;
  constant Real kappaExt = lambdaExt/(cvExt*raExt);
  constant Real TExt = 250;
  constant Real Tleft = 500;
  constant Real maxPower = 80000;
  constant Real Pcontrol = 10000;
  constant Real TTarget = 300;
  Real nDer;
  Real power;
algorithm
  if Functions.AtBoundary(i,j,k) then
    // if i == 1 then
    // d2Tdx2 := Pder.Pder2Dirichlet(f=state, fi=1, i=i, j=j, k=k, dim=1, bound=myBoundary);
    // d2Tdy2 := Pder.Pder2Dirichlet(f=state, fi=1, i=i, j=j, k=k, dim=2, bound=myBoundary);
    // d2Td2z2 := Pder.Pder2Dirichlet(f=state, fi=1, i=i, j=j, k=k, dim=3, bound=myBoundary);
    // else
    // if i == 1 then
    // nDer := kappaExt*(Tleft - state[1,i,j,k]);
    // else
    // end if;
    nDer := (TExt - state[1,i,j,k]);
    d2Tdx2 := Pder.Pder2Neumann(f=state, fi=1, i=i, j=j, k=k, dim=1, nder=nDer);
d2Tdy2 := Pder.Pder2Neumann(f=state, fi=1, i=i, j=j, k=k, dim=2, nder=0.1*nDer);
d2Tdz2 := Pder.Pder2Neumann(f=state, fi=1, i=i, j=j, k=k, dim=3, nder=0.1*nDer);
  // end if;
else
  d2Tdx2 := Pder.Pder2Inner(f=state, fi=1, i=i, j=j, k=k, dim=1);
d2Tdy2 := Pder.Pder2Inner(f=state, fi=1, i=i, j=j, k=k, dim=2);
d2Tdz2 := Pder.Pder2Inner(f=state, fi=1, i=i, j=j, k=k, dim=3);
end if;
value1 := kappa*(d2Tdx2 + d2Tdy2 + d2Tdz2);
  // if 1 < i and i < 5 and j==3 and k==3 then
if i < 5 then
  power := min(maxPower,
              Pcontrol*(TTarget - state[1,nx,integer(ny/2),integer(nz/2)]));
  power := max(power,0);
  value1 := value1 + kappa*power/lambda;
end if;
end ParDerState;

A.3.4 HeatInRodSettings
Heat in Rod Settings
within PDESolver.Model;

package Settings
  constant Types.FieldIndex n = {20,5,5};
  constant Types.Coordinate first = {0,0,0};
  constant Types.Coordinate last = {10,1,1};
  constant Integer stateFields = 1 "T";
  // constant Integer externalFields = 2
  // "Heat and conductivity";
  // constant Integer externalVariables = 1
  // "(linearDensity, tension)";
  constant Integer maxStencilWidth = 3;
end Settings;
Bibliography


77


På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtidiga ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämd som upphovsmann i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet – or its possible replacement – for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

©Gustaf Thorslund