Parallel Simulation of PDE-based Modelica Models using ParModelica

Gustaf Thorslund[1] Mahder Gebremedhin[2] Peter Fritzson[2] Adrian Pop[2]

[1]ThorslundTech AB, Sweden, gustaf@thorslundtech.se
[2]Dept. of Computer and Information Science, Linköping University, Sweden,
{mahder.gebremedhin,peter.fritzson,adrian.pop}@liu.se

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 article 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.

Keywords: OpenModelica, ParModelica, PDE, parallel computing, GPU, GPGPU

1 Introduction

To understand the behavior 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 (ODE) or Differential Algebraic Equation (DAE). If, however, the equations contain derivatives with respect to more than one variable, they describe a Partial Differential Equation (PDE).

Modelica[1] is an object oriented language[2] for modeling complex physical systems using equations. The model can then be simulated using a numerical solver. However, Modelica does not currently support modeling partial differential equations. There are suggested extensions for PDEs in (Fritzson, 2014; Saldamli, 2006).

OpenModelica[3] is an open source[4] implementation of the Modelica language, and an active research area.

Given that a PDE can describe a model in several dimensions, the required computations can grow exponentially with the size of the model. This should make it suitable for parallel computing.

1.1 ParModelica

ParModelica (Gebremedhin et al., 2012), implement a suggested extension for explicit parallelism in the algorithmic subset of Modelica. Similar to CUDA and OpenCL, it adds the concept of parallel computation device, device memory, and functions to be called on the device and within the device.

1.2 Previous Research on PDEs in Modelica

An extensive work on PDEs within Modelica has been done (Saldamli, 2006), 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.

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.

$$\rho_l \frac{\partial^2 \xi(x,t)}{\partial t^2} = F \frac{\partial^2 \xi(x,t)}{\partial x^2} + f_y(x) \tag{1}$$

$$\frac{\partial T}{\partial t} = \kappa \nabla^2 T + \left(\frac{\kappa h}{\lambda}\right) = \kappa \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) + \left(\frac{\kappa h}{\lambda}\right) \tag{2}$$

Equation (1) describes the vibration of a string with constant tension and (2) describes heat conduction, both equations are from (Nordling and Österman, 2006).

[1]http://www.modelica.org/ accessed May 2016

[2]The Modelica language is an open standard and can be downloaded for free. There is also a book (Fritzson, 2014) available with many examples of how to use the language.

[3]http://www.openmodelica.org/ accessed May 2016

[4]http://opensource.org/ accessed May 2016

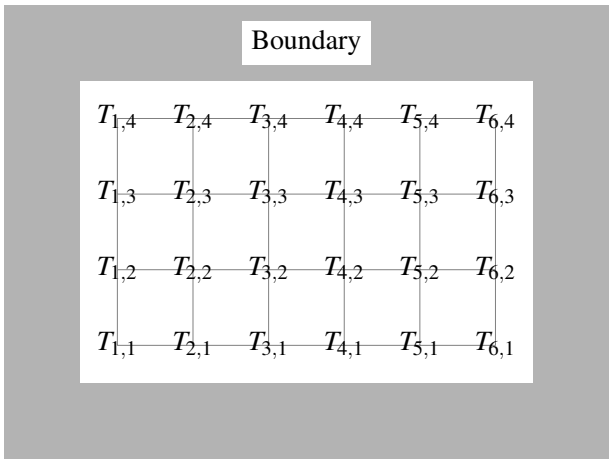

Figure 1. Method of lines applied to the heat conduction equation over a plane.

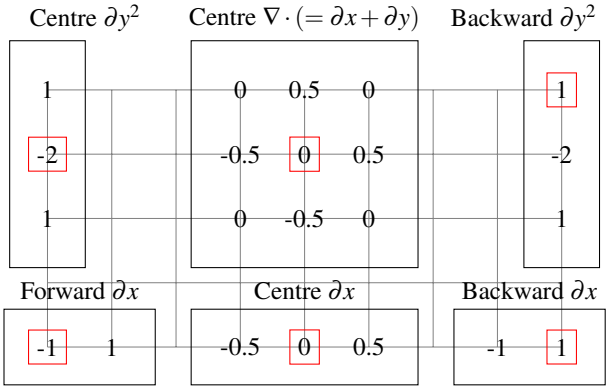

Figure 2. Example of stencils used for calculating spatial derivatives. The red box symbolizes the destination, while the numbers are the weights to use when summing up the neighboring values. They are all approximations, and some can be derived in different ways, resulting in different weights.

1.4 Explicit Form

In control theory and modeling it is often desirable to put an equation into explicit state form, see (Fritzson, 2014; Glad and Ljung, 1989; Ljung and Glad, 2004). 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:

$$\dot{\vec{x}}(t) = \vec{f}(\vec{x}(t),\vec{u}(t)) \tag{3a}$$
$$\vec{y}(t) = \vec{g}(\vec{x}(t),\vec{u}(t)) \tag{3b}$$

In case f and g are linear, matrix notation can be used instead:

$$\dot{\vec{x}}(t) = A\vec{x}(t) + B\vec{u}(t) \tag{4a}$$
$$\vec{y}(t) = C\vec{x}(t) + D\vec{u}(t) \tag{4b}$$

This article will only use the general form in (3). Models where an explicit form cannot be derived will require solver methods not covered here.

2 Numerics

To give a better understanding of the implementation, this section covers the algorithms involved in simulating mathematical models. For further reading, see: (Eldén and Wittmeyer-Koch, 1996; Fritzson, 2014; Ljung and Glad, 2004), or another book covering numerical analysis or applications of numerical analysis.

2.1 Discretisation

To be able to solve a PDE over space and time, one approach is to discretized the PDE over space and this way get a system of ODEs. If we take the heat conduction equation from (Nordling and Österman, 2006), with ∇^2 expanded to two dimensions, and calculate it at $n_x \times n_y$ discrete points in space we get:

$$\begin{aligned}\frac{\partial T_{i,j}}{\partial t} &= \kappa_{i,j}\nabla^2_{i,j}T + \left(\frac{\kappa h}{\lambda}\right)_{i,j} \\ &= \kappa_{i,j}\left(\frac{\partial^2 T_{i,j}}{\partial x^2} + \frac{\partial^2 T_{i,j}}{\partial y^2}\right) + \left(\frac{\kappa h}{\lambda}\right)_{i,j}\end{aligned} \tag{5}$$

Figure 1 shows how T has been discretised over a grid with 6×4 points. The derivatives in (2) can be approximated with:

$$\frac{\partial^2 T}{\partial x^2} = \frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2} \tag{6a}$$
$$\frac{\partial^2 T}{\partial y^2} = \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y^2} \tag{6b}$$

Those approximations can be derived using Taylor series, see for example (Eldén and Wittmeyer-Koch, 1996; Åström, 2015). As seen in (6), the discretisation (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 depending only on points at one side. The weights to used to approximate the spatial derivatives at a given point is commonly referred to as stencils. Different types of stencils are illustrated in Figure 2.

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. When running on a

General Purpose Graphic Processing Unit (GPGPU) each thread can have its own point.

2.2 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 (Bogacki and Shampine, 1989), parameters for calculating both a third and second order approximation using four computations of k are suggested:

$$k_1 = f(x_n, x_n) \tag{7a}$$
$$k_2 = f(t_n + \frac{1}{2}h, x_n + \frac{1}{2}hk_1) \tag{7b}$$
$$k_3 = f(t_n + \frac{3}{4}h, x_n + \frac{3}{4}hk_2) \tag{7c}$$
$$x_{n+1}^{(3)} = x_n + (\frac{2}{9}k_1 + \frac{1}{3}k_2 + \frac{4}{9}k_3)h \tag{7d}$$
$$k_4 = f(t_n + h, x_{n+1}) \tag{7e}$$
$$x_{n+1}^{(2)} = x_n + (\frac{7}{24}k_1 + \frac{1}{4}k_2 + \frac{1}{3}k_3 + \frac{1}{8}k_4)h \tag{7f}$$

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 General-Purpose Computing on Graphics Processing Units (GPGPU)

A Graphic Processing Unit (GPU) can be used as a computation device attached to a host, Figure 3. Within a GPU there are multiple Computation Unit (CU). The CUs 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.

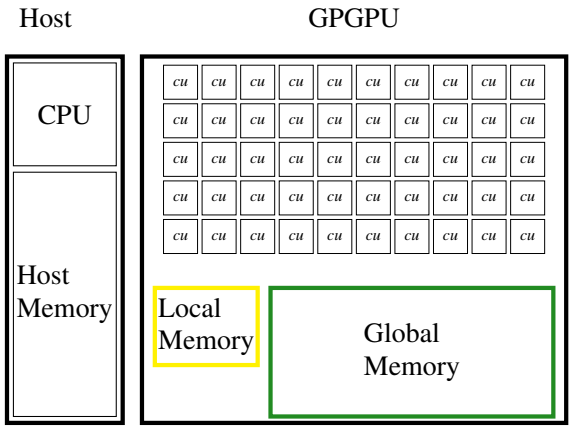

Figure 3. Computer equipped with a GPU. The GPGPU has a number of Computation Units, local and global memory.

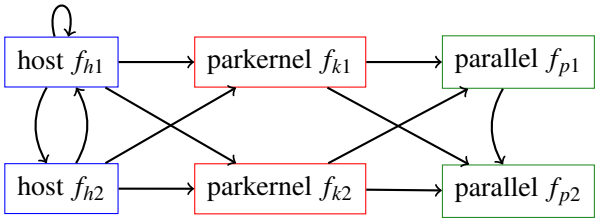

Figure 4. How functions can call each other in ParModelica

4 PDEs in Modelica

PDEs, in general, are not supported in Modelica. Starting with Modelica 3.3 there is support for spatialDistribution, allowing modelling of variable speed transport (Fritzson, 2014). The suggested extension in (Fritzson, 2014; Saldamli, 2006) are:

- field variables
- indomain construct

Those extensions would allow modelling a heat equation on a plane as:

```
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 omage.interior;
  T = 50          indomain omega.left;
  c*pder(T,D.x) = q+h*(T_ext-T)
    indomain omega.right;
  pder(T,D.y) = 0 indomain omega.top;
  pder(T,D.y) = 0 indomain omega.bottom;
end HeatInPlane;
```

5 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

Modelica functions can be called recursive. The *parallel* functions introduced by ParModelica can, however, only be called from a parallel context. This is from within a *parkernel* function, *parfor* loop or an other *parallel* function. Furthermore, the *parallel* functions cannot be called recursively. How functions can call each other is illustrated in Figure 4.
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

6 Solver Framework

This section gives an overview of the framework for solving PDEs.

6.1 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:

```
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.

```
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 "I";
```

```
  // Parameters used in the model
  constant Integer boxSize = integer(n[2]/2);
  constant Integer myBoundary = 3;
end Settings;
```

6.2 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);
```

6.3 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 lengh.";
  input Integer th1=1, th2=1, th3=1;
  output Types.Field next[size(state,1)]
    "New state";
protected
  // ...
algorithm
  // ...
end Solve;
```

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.

7 Use Case — Heat in Plane

In this section two use cases with heat conduction in a plane and different boundary conditions are presented, together with results and a discussion about result.

7.1 Poor Insulation and Constant Temperature

The boundary conditions here are similar to those in (Fritzson, 2014), with constant temperature at one side, poor insulation at opposite side and perfect insulation at the remaining two sides. Figure 5 shows how the temperature falls from the side with constant temperature to the side with poor insulation, while the sides with perfect insulation does not have an impact on the temperature.

```
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);
  else
    d2Tdy2 :=
      Pder.Pder2Neumann(f=state, fi=1,
                        i=i, j=j, k=k,
                        dim=2, nder=0);
  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);
end if;
value1 := c*(d2Tdx2 + d2Tdy2);
```

7.2 Constant Temperature Depending on Location

In this example boundaries have constant temperature depending on location around the plate. The result after a 500ms simulation can be seen in Figure 6. The tempera-

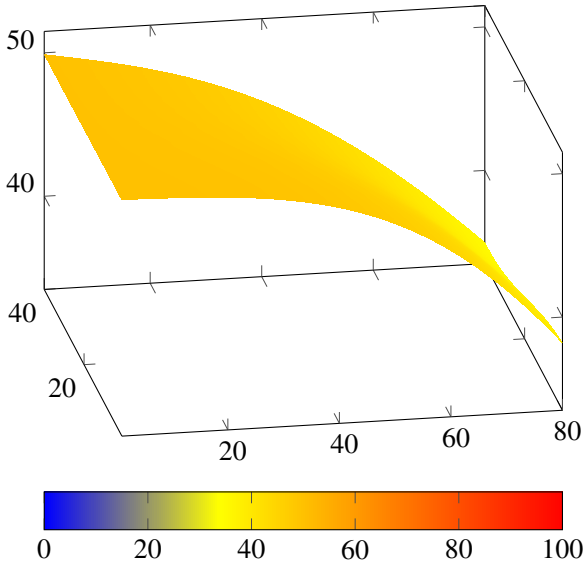

Figure 5. Plane with constant heat at the left side and poor insulation at the right side ($t = 0.5$).

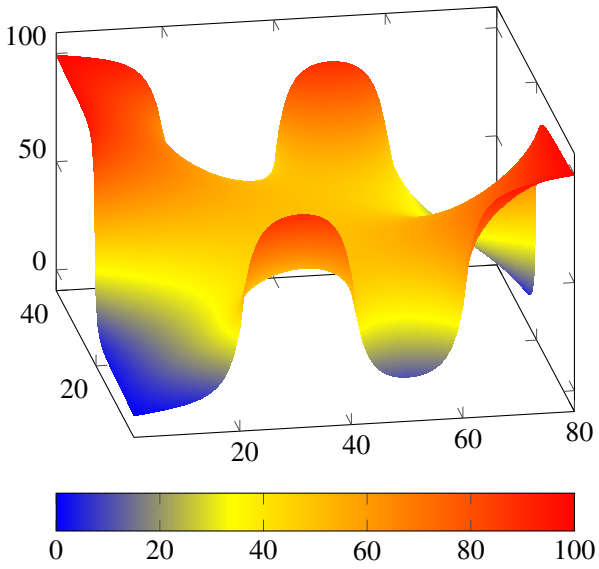

Figure 6. Boundary have constant temperature depending on location ($t = 0.5$).

ture have been forced towards 0 or 100 depending on the boundary conditions.

```
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);
value1 := c*(d2Tdx2 + d2Tdy2);
```

8 Performance Measurement

Performance measurement where done on a Fermi M2050 GPU with a total of 448 CUDA cores available. The same simulation was started using different number of threads and the speedup compared to just using one thread can be seen in Figure 7. In parallel computation there will always be a sequential part limiting the maximum speedup. This is due to the sequential part will take the same amount of time no matter how fast the parallel part may run.

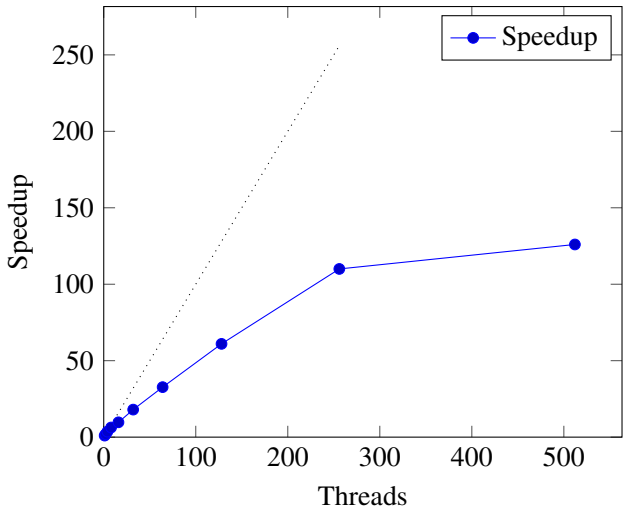

Figure 7. Speedup when simulating same model using different amount of threads for the simulation on a GPU with 448 CUDA cores.

9 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 appying 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.

10 Conclusions

In this article we show how an algorithmic solver can be implemented to utilise the explicit parallelism of 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.

While this work tries to provide the functionality to solve 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. This was presented during the OpenModelica workshop 2016, (Šilar, 2016). Once OpenModelica have PDE extensions in the frontend, and there is an efficient way to simulate PDEs, this needs to be integrated into all stages of the OpenModelica compiler and simulation runtime.

This work is still a research prototype and not stable enough to be included in the OpenModelica release. The work we present was initiated by (Thorslund, 2015).

References

P. Bogacki and L.F. Shampine. A 3(2) pair of Runge - Kutta formulas. *Appl. Math. Lett.*, 2(4):321–325, 1989.

Lars Eldén and Linde Wittmeyer-Koch. *Numerisk analys — en intruduktion*. Studentlitteratur, third edition, 1996.

Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3. A cyber-physical approach*. Wiley, second edition, 2014. ISBN 9781118859124.

Mahder Gebremedhin, Afshin Hemmati Moghadam, Peter Fritzson, and Kristian Stavåker. A data-parallel algorithmic modelica extension for efficient execution on multi-core platforms. In *Proceedings of the 9th International MODELICA Conference; September 3-5; 2012;*, pages 393–404, Munich, Germany, September 2012.

Torkel Glad and Lennart Ljung. *Reglerteknik Grundläggande teori*. Studentlitteratur, second edition, 1989.

Lennart Ljung and Torkel Glad. *Modellbygge och simulering*. Studentlitteratur, second edition, 2004.

Carl Nordling and Jonny Österman. *Physics Handbook*. Studentlitteratur, 2006. ISBN 978-91-44-04453-8.

Levon Saldamli. *PDEModelica – A High-Level Language for Modeling with Partial Differential Equations*. PhD thesis, Linköping University, PELAB - Programming Environment Laboratory, The Institute of Technology, 2006.

Gustaf Thorslund. Simulating partial differential equations using the explicit parallelism of ParModelica. Master's thesis, Linköping University, Software and Systems, Faculty of Science & Engineering, 2015.

Freddie Åström. *Variational Tensor-Based Models for Image Diffusion in Non-Linear Domains*. PhD thesis, Department of Electrical Engineering, Linköping University, 2015.

Jan Šilar. Partial Differential Equations in Modelica. OpenModelica2016-talk12-JanSilar-PartialDifferentialEquationsinModelica.pdf, 2016. URL `https://openmodelica.org/events/openmodelica-workshop/openmodelica-program-2016`.