Automatic Parallelization using Pipelining for Equation-Based Simulation Languages

by

Håkan Lundvall

Submitted to Linköping Institute of Technology at Linköping University in partial fulfilment of the requirements for the degree of Licentiate of Engineering

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

Linköping 2008
Automatic Parallelization using Pipelining for Equation-Based Simulation Languages

by

Håkan Lundvall

September 2008
Thesis No. 1381
ISSN 0280-7971

ABSTRACT

During the most recent decades modern equation-based object-oriented modeling and simulation languages, such as Modelica, have become available. This has made it easier to build complex and more detailed models for use in simulation. To be able to simulate such large and complex systems it is sometimes not enough to rely on the ability of a compiler to optimize the simulation code and reduce the size of the underlying set of equations to speed up the simulation on a single processor. Instead we must look for ways to utilize the increasing number of processing units available in modern computers. However to gain any increased performance from a parallel computer the simulation program must be expressed in a way that exposes the potential parallelism to the computer. Doing this manually is not a simple task and most modelers are not experts in parallel computing. Therefore it is very appealing to let the compiler parallelize the simulation code automatically. This thesis investigates techniques of using automatic translation of models in typical equation based languages, such as Modelica, into parallel simulation code that enable high utilization of available processors in a parallel computer.

The two main ideas investigated here are the following: first, to apply parallelization simultaneously to both the system equations and the numerical solver, and secondly, to use software pipelining to further reduce the time processors are kept waiting for the results of other processors. Prototype implementations of the investigated techniques have been developed as a part of the OpenModelica open source compiler for Modelica. The prototype has been used to evaluate the parallelization techniques by measuring the execution time of test models on a few parallel architectures and to compare the results to sequential code as well as to the results achieved in earlier work. A measured speedup of 6.1 on eight processors on a shared memory machine has been reached. It still remains to evaluate the methods for a wider range of test models and parallel architectures.

This work has been supported by the Swedish Foundation for Strategic Research (SSF) in the ECSEL graduate school, by Vinnova in the Safe and Secure Modeling and Simulation Project, and by the Swedish Research Council (VR).
Acknowledgements

I have learned a great deal during these years working towards this thesis and I have many people to thank for this. First, I would like to express my gratitude towards my supervisor Peter Fritzson for giving me the opportunity to work on this research and for the guidance and encouragement he has provided. I also want to thank my co-supervisors Christoph Kessler and Dag Fritzson for their support.

Even though I live two hours drive from the colleagues at PELAB it has always been a pleasant drive knowing that interesting discussion and good friends await when I arrive. Many thanks to all the employees at PELAB for that. And I would especially like to thank Bodil for all the administrative help which has made the life easier for all the rest of us.

Those days when I have not drove all the way to Linköping, Sogeti Sverige AB has provided a nice working atmosphere here in Örebro, as well as a computer to work with.

My time has been divided evenly between the PhD studies and the work as a consultant at Sogeti and I wish to thank Atlas Copco Rock Drills AB for keeping me busy with assignments through out these years knowing that they can only get half of my attention.

Finally I would like to thank my lovely wife Jenny and my wonderful daughters Emma and Lina for always being there and believing in me.

Örebro, August 13, 2008
Håkan Lundvall
# Table of Contents

## Part 1  Introduction

### Chapter 1  Background and Introduction

1.1  Modeling and Simulation  
1.2  Related Work on Task Scheduling and Clustering  
1.3  Research Problem  
1.4  Contributions  
1.5  Thesis Structure  
1.6  List of Papers  

## Chapter 2  Implementation – The OpenModelica Parallelizing Backend

2.1  Background  
2.2  Compiler Phases  
2.3  Modules of the OpenModelica Compiler  
2.3.1  DAELow  
2.3.2  TaskGraph  
2.3.3  TaskGraphExt  
2.3.4  ModPar/Schedule  
2.3.5  ModPar/Codegen  
2.3.6  ModPar/TaskGraph  
2.4  Simulation Runtime  
2.4.1  MPI-Based Runtime System  
2.4.2  Pthreads-Based Runtime System  
2.5  Additional Load Balancing and Memory Layout Ideas  

## Bibliography  

## Part II  Papers

### Paper A  Event Handling in the OpenModelica Compiler and Runtime System  

1  Introduction  
2  Conversion of a Modelica Model to a Hybrid DAE  
3  Simulation of Hybrid Models Based on Solving Hybrid DAEs  
4  The Form of the Hybrid DAE System  
4.1  Summary of Notation  
4.2  Continuous-Time Behavior  
4.3  Discrete-Time Behavior  
4.4  The Complete Hybrid DAE  
5  Hybrid DAE Solution Algorithm  
6  Varying Structure of the Active Part of the Hybrid DAE  
7  Finding Consistent Initial Values at Start or Restart  
8  Detecting Events during Continuous-time Simulation  
9  Crossing Functions
<table>
<thead>
<tr>
<th>10</th>
<th>Introducing the Numerical Solver with Event-Handling</th>
<th>42</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Code Generation</td>
<td>44</td>
</tr>
<tr>
<td>12</td>
<td>Example</td>
<td>45</td>
</tr>
<tr>
<td>13</td>
<td>Measurements and Evaluation</td>
<td>47</td>
</tr>
<tr>
<td>13.1</td>
<td>Bouncing Ball</td>
<td>47</td>
</tr>
<tr>
<td>13.2</td>
<td>Full Wave Rectifier</td>
<td>48</td>
</tr>
<tr>
<td>14</td>
<td>Conclusions</td>
<td>50</td>
</tr>
<tr>
<td>15</td>
<td>Acknowledgements</td>
<td>50</td>
</tr>
<tr>
<td>16</td>
<td>References</td>
<td>50</td>
</tr>
</tbody>
</table>

**Paper B**  Automatic Parallelization of Object Oriented Models Across Method and System  

<table>
<thead>
<tr>
<th>1</th>
<th>Background – Introduction to Mathematical Modeling and Modelica</th>
<th>55</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>The OpenModelica Open Source Implementation</td>
<td>57</td>
</tr>
<tr>
<td>3</td>
<td>Approaches to Integrate Parallelism and Mathematical Models</td>
<td>57</td>
</tr>
<tr>
<td>3.1</td>
<td>Automatic Parallelization of Mathematical Models</td>
<td>58</td>
</tr>
<tr>
<td>3.2</td>
<td>Coarse-Grained Explicit Parallelization Using Computational Components</td>
<td>59</td>
</tr>
<tr>
<td>3.3</td>
<td>Explicit Parallel Programming</td>
<td>59</td>
</tr>
<tr>
<td>4</td>
<td>Combining Parallelization at Several Levels</td>
<td>59</td>
</tr>
<tr>
<td>5</td>
<td>Pipelining the Task Graph</td>
<td>62</td>
</tr>
<tr>
<td>6</td>
<td>Sorting Equations for Short Access Distance</td>
<td>62</td>
</tr>
<tr>
<td>7</td>
<td>Scheduling</td>
<td>63</td>
</tr>
<tr>
<td>8</td>
<td>Measurements</td>
<td>64</td>
</tr>
<tr>
<td>9</td>
<td>Conclusion</td>
<td>65</td>
</tr>
<tr>
<td>10</td>
<td>Future work</td>
<td>65</td>
</tr>
<tr>
<td>11</td>
<td>Acknowledgements</td>
<td>66</td>
</tr>
<tr>
<td>12</td>
<td>References</td>
<td>66</td>
</tr>
</tbody>
</table>

**Paper C**  Automatic Parallelization of Models using Pipeline Extraction from Combined RHS and Inlined Solvers  

<table>
<thead>
<tr>
<th>1</th>
<th>Introduction</th>
<th>71</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Background on Mathematical Modeling and Modelica</td>
<td>72</td>
</tr>
<tr>
<td>2.1</td>
<td>The OpenModelica Open Source Environment</td>
<td>74</td>
</tr>
<tr>
<td>3</td>
<td>Approaches to Exploit Parallelism in Mathematical Models</td>
<td>75</td>
</tr>
<tr>
<td>3.1</td>
<td>Automatic Fine-Grained Parallelization of Mathematical Models</td>
<td>75</td>
</tr>
<tr>
<td>3.2</td>
<td>Coarse-Grained Explicit Parallelization Using Computational Components</td>
<td>75</td>
</tr>
<tr>
<td>3.3</td>
<td>Explicit Parallel Programming</td>
<td>76</td>
</tr>
<tr>
<td>4</td>
<td>Combining Parallelization from Several Levels of Abstraction</td>
<td>76</td>
</tr>
<tr>
<td>5</td>
<td>Pipelining the Task Graph</td>
<td>77</td>
</tr>
<tr>
<td>6</td>
<td>Sorting Equations for Short Access Distance</td>
<td>77</td>
</tr>
<tr>
<td>7</td>
<td>Scheduling</td>
<td>79</td>
</tr>
<tr>
<td>8</td>
<td>Measurements</td>
<td>81</td>
</tr>
</tbody>
</table>
9 Conclusions 85
10 Future work 85
11 Acknowledgements 85
12 References 86
Part 1

Introduction
Chapter 1

Background and Introduction

In this chapter we first give a short background and introduction to the research area investigated. We also state the research problem and list the contributions of this work. At the end of the chapter we present the structure of this thesis and list the papers presenting the results of the research.

1.1 Modeling and Simulation

In many areas modeling and simulation becomes more and more important. In product development, for example, modeling and simulation is used to allow for shorter development time and more robust systems. Instead of embarking on a time consuming and expensive prototype construction, a model of the system is built and experiments are performed on this model. The model can be a physical model where some details of the real system has been left out, but usually the model is a mathematical one stored in a computer and the experiments carried out on the model are computer simulations. Running the simulations on a model instead of building the system can help us to answer about the system which would not otherwise be possible due to reasons such as:

- The experiments would be too expensive to perform on a real system.
- The investigated variable may not be accessible in the real system.
- Performing some experiments on the real system may be too dangerous.
- It may take a too long time to perform the experiment on the real system.

In order to perform a simulation of a model on a computer we must have an executable program that performs all the calculations needed. When computer simulations first saw the light of day these programs were hand written in relatively low level languages like C or Fortran. Although hand coded simulation programs are still used and developed, more and more models are written in high level equation-based modeling languages such as Modelica[4], gPROMS[9], and VHDL-AMS[8]. One of the great benefits of these equation-based languages is that the mathematical models behind the simulations are usually given in equations and a lot of manual work of transforming the equations into assignment statements can be
avoided. The same submodel can also be used in several different contexts where the dependent variables, i.e., the variables to solve for, are not necessarily the same. This greatly reduces development time and improves reusability. The information about which variables are dependent variables and which are independent variables in the equations is called the causality of the system.

As the models grow larger and more accurate simulation results are desired, more computational power is needed. Although computers have become faster, recently most of the increased computational power of new computers can only be utilized with extensive usage of parallelism. This puts demands on the modeling languages used to allow the simulation programs to utilize parallelism either by introducing language constructs to allow the modeler to express parallelism explicitly in the models, but even better by automatically parallelizing the same models previously used in simulations on single CPU machines.

### 1.2 Related Work on Task Scheduling and Clustering

When a mathematical model is translated into executable simulation code a task graph is generated which describes all dependencies between the tasks which puts restrictions on the order in which the tasks can be executed. Therefore it is important to have efficient algorithms that can schedule the tasks on a parallel computer. Much research has been done in this area. In this section we summarize some of the previous work on task scheduling and clustering.

A classification of scheduling algorithms can be found in [13], which classifies the algorithms in a hierarchical manner. At the top level the algorithms are divided into local and global algorithms, where the local algorithms schedule the tasks on one processor whereas the global algorithms consider the case of multiple processors. The global algorithms are further divided into static and dynamic algorithms. In this work we explore static scheduling which means the scheduling takes place at compile time.

Static scheduling can be done optimally or sub-optimally and sub-optimal scheduling is further divided into heuristic and approximate scheduling. Both the optimal and the approximate sub-optimal scheduling are divided into four different kinds: enumerative, graph-theory, mathematical programming, and queuing theory. The work in this thesis falls under the approximate scheduling category.

There exist algorithms for automatically scheduling entire taskgraphs on a fixed number of processors, such as the Modified Critical Path (MCP) scheduling algorithm [14] and the Earliest Ready Task (ERT) algorithm [15].

The problem with scheduling algorithms which solves the entire scheduling problem, including assigning each task to a processor, is the computational complexity of such algorithms. Therefore, a clustering algorithm is used to make the job easier for the scheduler. A cluster is a set of tasks that run on the same processor. After the clustering face, a smaller number of clusters can be assigned to the processors and the scheduler still need to determine the execution order within the processor, i.e., local scheduling.
Existing clustering algorithms include the Internalization algorithm \cite{16} and the Dominant Sequence Clustering (DSC) algorithm \cite{17}. The Internalization algorithm works by traversing all edges of the task graph and check whether assigning the two tasks to the same processor would lead to an increased total execution time. If this is not the case the edge is internalized and the two tasks are put in the same cluster.

1.3 Research Problem

In this section we discuss the research problem of this work.

In the research leading to this thesis the aim has been to investigate if other techniques to automatically translate models stated in equation-based languages into parallel code can lead to more efficient code compared to previously tested approaches. A second objective of this work has been to reduce the complexity of the translation algorithm in order to speed up the translation process.

The research methodology used in this work is the traditional system oriented computer science research method. In order to validate our hypotheses, a prototype implementation is built. By running simulations of test models on parallel computers we measure the amount of time needed to complete the simulations. We can compare these measurements to test runs on a single processor and thereby calculate the gained speed-up. In order to compare the results to earlier work the test model used is chosen such that test data are available from the earlier work to which we compare our results. This work shares essentially the same objectives as the work presented in \cite{7} by Peter Aronsson, but we continue a bit further and develop some new methods.

Parallelizing the simulation of a mathematical model can be done at several levels, see Paper B, page 59. The calculation of the simulation results involves a numerical solver which iterates over the simulation problem in order to compute the result. Parallelization can be done at the numerical method level by using a parallel version of the solver or at the system level by using a sequential solver where each call to the right hand side of the system of equations is parallelized.

One hypothesis explored in this work is that more speedup can be gained if parallelization is done at both of these levels at the same time.

The methods developed in Aronssons thesis use a graph rewriting system to automatically merge tasks into larger units to increase the granularity of the task graph. This is done on the global full system task graph. A second hypothesis explored in this work is that compilation time can be reduced by assigning the tasks to processors early, guided by information about locality present in the model, to avoid expensive work on a large task graph, and instead later merge the tasks locally in each processor, to increase granularity.
1.4 Contributions

The following are the main contributions of this work:

- Contributions to the OpenModelica compiler (specifically the event handling mechanism) which is used in several research projects. This is described in Paper A, pages 29-50.
- A new variant of topological sorting of equations to allow for simple and fast scheduling with little communication. This is described in Paper B, pages 53-66.
- Task clustering after processor assignment to enable pipelining of simulations steps without unnecessary waiting times. This is described in Paper C, pages 69-85.

1.5 Thesis Structure

This thesis has the following structure.

Part I is divided into two chapters where the first gives an introduction including the research objective, some background on the Modelica Language and a summary of the contributions of this thesis. The second chapter describes the implementation of the prototype system developed as a part of this thesis work.

Part II contains the papers which presents the research results of this work.

1.6 List of Papers

The research results are presented in three papers in Part II of the thesis. The papers are;

Paper A: Event Handling in the OpenModelica Compiler and Runtime System

Håkan Lundvall, Peter Fritzson, and Bernhard Bachmann

Paper B Automatic Parallelization of Object Oriented Models Across Method and System

Håkan Lundvall and Peter Fritzson


Paper C Automatic Parallelization of Models using Pipeline Extraction from Combined RHS and Inlined Solvers

Håkan Lundvall and Peter Fritzson

Edited version of a paper submitted to the 14th Workshop on Compilers for Parallel Computing, IBM Research Center, Zurich, Switzerland, January 7-9, 2009
Chapter 2

Implementation – The OpenModelica Parallelizing Backend

In order to test and evaluate the methods and theories developed in this thesis work, which are described in Papers A to C, a prototype implementations of these methods have been developed as part of the OpenModelica compiler. This chapter describes the implementation of the prototype parallelizing backend for OpenModelica[6].

2.1 Background

OpenModelica is an open source project aiming at creating a complete Modelica modeling, compilation and simulation environment. The current version includes the following parts:

- A Modelica compiler (omc).
- An interactive shell (OMShell) which gives a command interface to the compiler.
- The OpenModelica Notebook (OMNotebook), a Mathematica style electronic notebook for Modelica where Modelica models can be written and simulated within the document.
- A Modelica Eclipse plugin (MDT – Modelica Development Tooling)
- The OpenModelica Development Environment (OMDev).

The OpenModelica project was started at PELAB in 1998 and is now run within The Open Source Modelica Consortium [11].

In earlier work on automatic parallelization of Modelica code, the ModPar[7] system was developed and integrated into the OpenModelica compiler.
Some code from the ModPar system has been reused also in this implementation to save implementation time. The ModPar system is divided into a number of different modules, some written in MetaModelica and some in C++. A description of the modules is presented in Section 2.3.

### 2.2 Compiler Phases

In this section we provide a short overview of the different compiler phases involved when compiling a Modelica model, see also Figure 2-1.

First a Modelica parser reads the Modelica source code and produces an abstract syntax tree (AST) that represents the Modelica model. In the next step a translator reorganizes the model into a flat list of equations and variables. This involves, among other things, handling inheritance, import statements and redeclarations.

After the equations have been flattened the equations are topologically sorted according to the causal dependencies between the equations. This step involves symbolic manipulations such as algebraic simplifications, symbolic index reduction, and, removal of trivial equations. Once the causality has been established, the dependent variable in each equation, not part of a strong component, is solved for and the equations are converted to assignment statements. Equations that are part of a strong component must be solved simultaneously. Such systems of simultaneous equations are either solved symbolically at this stage, if possible, or left to a numerical solver. We now have an execution order in which the right hand side of the equations can be evaluated.

In the last step C code is generated for all the assignment statements and a suitable C code representation (e.g. an RHS function) of the strongly connected equations is generated, which is then linked to a numerical solver.
Figure 2-1. OpenModelica compiler translation phases.

The newly developed version of ModPar only influences the last few steps; equation sorting and code generation. An extra step has been added to the topological sorting to keep equations with dependencies closer together, and the code generation step is exchanged for a set of modules which generates a task graph and performs scheduling and generation of parallel code. In section 2.3 we give a more detailed explanation of the modules involved.

2.3 Modules of the OpenModelica Compiler

The OpenModelica compiler is divided into a number of modules, see Figure 2-2. A description of the entire compiler and its modules can be found in the OpenModelica system documentation [10]. Most of the modules are implemented in MetaModelica [12]. The ModPar system adds a number of modules to the compiler. A list of the modules implementing the ModPar system is shown in table 2-1. For each module we have a section “Added Functionality” which summarizes the contributions to the implementation compared to the old ModPar version.
Figure 2-2. The most important modules of the OpenModelica Compiler (omc) with module connections and data flow (from the OpenModelica System Documentation). There are more than 40 modules in the compiler.

When using the ModPar system the SimCodeGen module is exchanged for the ModPar system which takes over the task of generating simulation code. Some changes have also been made to the DAELow module which handles the topological sorting of the equations.

<table>
<thead>
<tr>
<th>Module name</th>
<th>Implementation language</th>
<th>Specific to ModPar</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAELow</td>
<td>MetaModelica</td>
<td>No</td>
</tr>
<tr>
<td>TaskGraph</td>
<td>MetaModelica</td>
<td>Yes</td>
</tr>
<tr>
<td>TaskGraphExt</td>
<td>MetaModelica/C++</td>
<td>Yes</td>
</tr>
<tr>
<td>ModPar/Schedule</td>
<td>C++</td>
<td>Yes</td>
</tr>
<tr>
<td>ModPar/Codegen</td>
<td>C++</td>
<td>Yes</td>
</tr>
<tr>
<td>ModPar/TaskGraph</td>
<td>C++</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 2-1. List of modules implementing the ModPar system.

2.3.1 DAELow

This module takes care of sorting of the equations according to data dependency and building the block lower triangular form of the system. Input to the module is
the flattened set of equations. The module provides functions for matching of variables to solve for, to access equations and finding out the causality between the equations.

### 2.3.1.1 Main Functions

Below follows explanations of the most important functions of the DAELow module:

```plaintext
public function lower
  input DAE.DAElist lst;
  input Boolean add_dummy;
  output DAELOW outDAELow;
end lower;
```

This function takes a list of discrete algebraic equations (DAE), which is the flattened model, and translates it into a different form, called DAELow defined in this module. The DAELow representation splits the DAE into equations and variables and further divides variables into known and unknown variables and the equations into simple and non-simple equations.

The input variable `add_dummy` adds a state variable and an associated equation which is independent of all other variables. This is used if the model contains no state variables since the numeric solver requires at least one state variable.

```plaintext
public function incidenceMatrix
  input DAELOW inDAELow;
  output IncidenceMatrix outIncidenceMatrix;
end incidenceMatrix;
```

This function calculates the incidence matrix, i.e., which variables are present in each equation. This matrix has rows representing each equation and columns representing each unknown variable. A non-zero entry means that the (column) variable is referenced in the (row) equation. See also Paper C, page 77 and the following page.
public function matchingAlgorithm
    input DAELow inDAELow;
    input IncidenceMatrix inIncidenceMatrix;
    input IncidenceMatrixT inIncidenceMatrixT;
    input MatchingOptions inMatchingOptions;
    output Integer[:,:] outAssignments1;
    output Integer[:,:] outAssignments2;
    output DAELow outDAELow3;
    output IncidenceMatrix outIncidenceMatrix4;
    output IncidenceMatrixT outIncidenceMatrixT5;
end matchingAlgorithm;

This function performs the matching algorithm, which is the first part of sorting the equations into BLT (Block Lower Triangular) form. The BLT form is discussed in Paper C on page 77. The matching algorithm finds a variable that is solved in each equation. But to also find out which equations form a block of equations, the second algorithm of the BLT sorting is run, finding strong components. This function returns the updated DAE which, in case of index reduction, has added equations and variables, and the incidence matrix. The variable outAssignments is returned as a vector of variable indices, as well as its inverse, i.e., which equation a variable is solved in as a vector of equation indices.

The variable inMatchingOptions contain options given to the algorithm.

- Whether index reduction should be used or not.
- Whether the equation system is allowed to be underconstrained or not, which is used when generating code for initial equations.

public function strongComponents
    input IncidenceMatrix inIncidenceMatrix1;
    input IncidenceMatrixT inIncidenceMatrixT2;
    input Integer[:,:] inAssignments1;
    input Integer[:,:] inAssignments2;
    output list<list<Integer>> components;
end strongComponents;

This is the second part of the BLT sorting. It takes the variable assignments and the incidence matrix as input and identifies strong components, i.e., subsystems of equations. The result is stored in the output result variable, a list of integer lists called components. The integers stored in the innermost list represent the equation numbers and are references to specific equations of the model. The result is a topological sorting of the dependencies between the equations, i.e., the equations can be solved in the order given by the outermost list. If the innermost list contains more than one equation, it constitutes a strongly connected component and the involved equations must be solved simultaneously.
public function sortBlocksForShortAccessDistance
    input list<list<Integer>> comps;
    input Integer[:] matching;
    input IncidenceMatrix incidenceMatrix;
    output list<list<Integer>> outComps;
end sortBlocksForShortAccessDistance;

This function takes an already topologically sorted list of equations and rearranges the equations to bring equations with direct dependencies closer together. Pseudo code for this function can be found in Paper B, after page 62.

2.3.1.2 Added Functionality

The main difference compared to the sequential version of the compiler is the sorting step introduced in the function sortBlocksForShortAccessDistance. This can greatly reduce the need for communication between the processors when we schedule the tasks based on the order of the equations given by this sorting step.

2.3.2 TaskGraph

This module replaces the SimCodegen module when generating parallel code. In the standard OpenModelica compiler the SimCodegen module traverses the sorted equations as they come from the DAELow module and generates C code for the equations. When generating parallel code the TaskGraph module traverses the equations in the same way with the difference that instead of C code, a task graph is generated and extra tasks implementing a Runge-Kutta solver are interleaved in the generated task graph, i.e., a kind of inlining of the solver.

2.3.2.1 Main Functions

The buildTaskgraph function is the main function and only public function of the TaskGraph module. Its job is to generate the task graph.

public function buildTaskgraph
    input DAELow.DAELow inDAELow1;
    input Integer[:] inAssignments2;
    input Integer[:] inAssignments3;
    input list<list<Integer>> inComponets;
end buildTaskgraph;

It takes the sorted list of equations and the matching of equations to unknown variables as input. For each equation the variable matched to the equation by the matching algorithm is solved for so that the equation can be reorganized into an assignment statement. Then tasks are added for the right hand side of the assignment and one task for the result being the root of this generated subtree.
When generating tasks for the right hand side expression, references to unknown variables are looked up in the already generated part of the task graph. Tasks for calculating all such references are already in the generated part of the task graph since the equations are traversed in data dependency order. All tasks are marked with the position in the topological sorting of the equation from which they were generated.

The function does not have a return value. Instead the task graph is stored in the internal state of the C++ modules. Normally MetaModelica functions cannot have internal state, but calling C++ functions makes this possible.

2.3.2.2 Added Functionality

This module now also generates tasks for the solver itself and not just for computing the right hand side of the simulated system. This means some extra variables are introduced to store intermediate results used by the solver tasks and the tasks calculating the right hand side are duplicated for each solver stage.

2.3.3 TaskGraphExt

This module constitutes the interface between the MetaModelica part and the C++ part of the backend. It exposes functions for building the task graph and generating simulation code. Since this module and the rest of the ModPar system is implemented as a C++ library it can unlike MetaModelica modules, hold an internal state. The generated task graph is stored in this internal state of the C++ library and is accessible from all the C++ modules described below.

2.3.4 ModPar/Schedule

This module implements the scheduling algorithm. Input to the module is the generated task graph and the number of processors. The result is one task list for each processor. The first thing the scheduler does is to split all the tasks into one set of tasks per processor to generate code for. The position of the equation in the topological sorting is used for this.

In the next stage a topological sorting of the tasks within a processor must be established. This is what is discussed in Paper C. The order in which the tasks are used when generating the C code is established using a priority queue in which all tasks associated to a processor is stored. The queue is sorted on two properties: 1) task set index and 2) level. The task set index refers to which task set within a given processor the task has been assigned to and the level refers to the length of the path from the task to the ‘end task’. The ‘end task’ being a task with inbound edges from all the tasks representing the unknown variables, thus being the last task in any topological sorting. The task sets are discussed in Paper C. In short all tasks within one processor are divided into three sets executed in order and where all inbound
communication to a processor is gathered to the boundary between the first set \((P_{a,i})\) and the second set \((P_{b,i})\) and where all outgoing communication is gathered to the boundary between the second set and the third set \((P_{c,i})\).

To calculate these properties the task graph is traversed with three depth first traversals. The first one has a visitor function which sets the level attribute of each visited node to the maximum level of all child nodes incremented by one. The second one is a reverse depth first search which divides all nodes into three categories:

1. The node has at least one incoming edge from a task scheduled on a higher ranked processor or at least one parent that was already assigned to this category. This means that this node is dependent on results from the previous simulation step.
2. The node has at least one incoming edge from a task scheduled on a lower ranked processor or at least one parent was already put in this category. This means that some inbound communication from another processor must have preceded this task.
3. The node does not fit into either of the two categories above.

In the third traversal the tasks are clustered into the three sets \((P_{a,i}, P_{b,i}, \text{and} P_{c,i})\). The visitor function of the third traversal works in the following way:

- If the currently visited node is in category 3 and there is at least one child in category 1 or 2, then the node is inserted in \(P_{a,i}\).
- If the currently visited node is in category 1 or 2 and there are at least one child assigned to different processor, then the node is inserted in \(P_{b,i}\).
- Otherwise the currently visited node is inserted in \(P_{c,i}\).

### 2.3.4.1 Main Functions

The following are the main functions of the ModePar/Schedule module.

\[
\text{Schedule(TaskGraph*, VertexID start, VertexID end, int nproc);} \\
\text{get_tasklist(int proc);} \\
\text{getAssignedProcessors(VertexID v);} \\
\text{isAssignedTo(VertexID v, int proc);} \\
\]

Initializes the scheduler and performs the scheduling algorithm described above.

Returns the sorted task list for a given processor.

Returns the set of tasks assigned to a given processor.

Checks whether a given node in the task graph is assigned to a processor.
2.3.4.2 Added Functionality

This module is almost completely rewritten since a totally different scheduling approach is used compared to the original ModPar system. The task merging code is replaced by a new scheduler that performs processor assignment by column index in the BLT matrix as given by the DAELow module and local scheduling of the tasks for each processor. The job of increasing the granularity of the code is moved from the global perspective into the local scheduling for each processor.

2.3.5 ModPar/Codegen

This module generates parallel C code from the task graph along with the schedule provided by the modpar/Schedule module.

The schedule provides one task queue for each processor to generate code for. One function is generated for each task queue and put in separate files so that each function is put in a separate object file together with the data array for the simulation variables. Before generating the simulation functions the tasks are traversed to collect data about all the communication needs. For each pair of processors between which messages need to be communicated, a set is kept where all variables communicated are stored. Before beginning to generate the tasks in $P_{c,i}$ for processor a given processor $i$ these communication sets are used to generate message receive code for all messages with processor $i$ as destination. All messages between $P_{a,j}$ and $P_{b,i}$, for two processors $i$ and $j$, are merged into one single message. The same happens with the message send code after the tasks of $P_{b,i}$ are generated.

2.3.5.1 Main Functions

The following are the main functions in the ModPar/Codegen module.

```c
void initialize(TaskGraph*, Schedule *, ...);
```

This function initializes the code generator. It takes the global task graph and the schedule as input parameters. Some arguments to the function have been omitted for brevity.

```c
void generateCode(char * init_code);
```

This function uses the schedule and the task graph given in the initialization and generates C code.

2.3.5.2 Added Functionality

In the version of ModPar that existed before this work started, there was only a runtime for MPI available. In that version the tasks reaching the code generator was merged into larger but fewer tasks. In a merged task all result variables of one large task was collected into one message, but there could be more than one message
between one pair of processors. In the new version the original small tasks are kept, instead they are clustered into three sets. This means that the message passing code has been rewritten.

The simulation code for the different processors is now also generated into separate functions in order to be usable by the pthreads runtime.

2.3.6 ModPar/TaskGraph

This module implements the task graph and provides primitives for accessing properties of the task graph nodes. The task graph is implemented using the Boost graph Library [3].

2.3.6.1 Main Functions

The following are the main functions in the ModPar/TaskGraph module.

```c
EdgeID add_edge(VertexID parent, VertexID child, 
                 TaskGraph *tg, string *result=0, 
                 int prio=0);
```

This function adds a vertex between two nodes in the task graph. The name of the result variable communicated from the parent is given in the result argument.

In case of more than one incoming edge to a node there is a priority given in order to sort out which order to use the incoming data in the task.

Other graph manipulating functions are available through the Boost graph library.

2.3.6.2 Added Functionality

The nodes of the task graph have been equipped with a new attribute which stores information about which variable the task is involved in evaluating the result for. This information is used by the scheduler to assign the task to the appropriate processor. Otherwise this module has been kept largely the same.

2.4 Simulation Runtime

When the simulation code has been generated it is linked together with a simulation runtime. The runtime is written in C++. Two different runtime systems have been developed. One for distributed memory using MPI [1] and one for shared memory using pthreads [2].
2.4.1 MPI-Based Runtime System

In this section we give a short overview of the MPI-based runtime system.

When running a parallel program using MPI all nodes run the same executable and the MPI system takes care of starting up the program on each node that should be used. Since the same code is run on all processors a switch statement takes care of calling the correct simulation code for each processor.

Each task of the task graph generates a result, but if a task only represents a sub-expression of one equation then there is no simulation variable from the model to store the result in, so a temporary variable is used. In the MPI-based runtime these temporary variables are allocated as a global array on the heap. This is due to the fact that in the previous version of ModPar it was possible that such a variable would be used in a message to another processor. In the new version, however, all tasks originating from the same equation always get assigned to the same processor.

Messages to other processors are implemented using non-blocking MPI send and receive. Before sending all the variables involved in one message are copied to a send buffer, since they are not necessarily placed consecutively in memory.

On the receiving end there is a corresponding receive buffer from which the values are copied to their final destination.

The function containing the simulation code is called from within a loop which first updates the simulation time variable and after the simulation code is called stores the new values of all simulation variables in a result file. One result file is generated for each processor. In this way the time it takes to store the simulation results are also distributed over all the processors. The files must be concatenated afterwards to get the complete simulation result.

2.4.2 Pthreads-Based Runtime System

In the pthreads runtime the entire simulation is running in the same process but it is divided into several threads. Since all threads share the same memory it is possible to just keep one array of simulation variables that all threads reference. For the communication we just have to make sure that the task that produces the value for a variable that is used in a task on another thread has completed before the receiving task starts. To accomplish we use a map indexed by the communication id consisting of the sending and receiving thread number. In this map we store how many simulation steps the sending thread has completed. The receiving thread performs a busy waiting loop until the sending thread has passed the point were the needed values are available and has increased the counter in the communication map.

However, it turns out that the simulation goes a lot faster if we separate the simulation variables into one array per thread, see Paper C. Thus, in the latest version of the prototype each thread has its own copy of the simulation variable array and the communication mechanism is extended to copy the relevant variables between the copies.
The temporary variables used to store the results of subexpressions are generated as local variables. In this way the C compiler is able to eliminate them during optimization. The resulting executable looks the same as if the entire equation was generated as one single C expression instead of a series of assignment statements using only one operator per expression.

### 2.5 Additional Load Balancing and Memory Layout Ideas

Some ideas have not yet found their way into the current ModPar prototype.

In Paper C we mention the possibility of load balancing when the execution cost of all tasks are not known. The idea is to make an initial guess of assigning the tasks evenly over two processors and then run a small test run, adjust the initial guess based on the test run and then iterate until the tasks are balanced. Then we split each half in two and do the same thing for four processors. If this is done inside the compiler a large part of the compilation work can be reused for each test compilation so the compilation time should not suffer that much. This function is not implemented, but we made some small test manually, which indicates that a better load balancing can be achieved compared to just splitting the tasks evenly over the processors.

In the current version of the prototype the simulation variables are laid out in memory in the order they appear in the model. It would be better if we made the memory layout of the variables according to the equation sorting. This would keep all variables used by one processor close in memory. We should also make sure that variables that are communicated between processes should be laid out adjacent to each other so that they need not be copied to a communication buffer before transmission to other processors.
Bibliography

During the most recent decades modern equation-based object-oriented modeling and simulation languages, such as Modelica, have become available. This has made it easier to build complex and more detailed models for use in simulation. To be able to simulate such large and complex systems it is sometimes not enough to rely on the ability of a compiler to optimize the simulation code and reduce the size of the underlying set of equations to speed up the simulation on a single processor. Instead we must look for ways to utilize the increasing number of processing units available in modern computers. However to gain any increased performance from a parallel computer the simulation program must be expressed in a way that exposes the potential parallelism to the computer. Doing this manually is not a simple task and most modelers are not experts in parallel computing. Therefore it is very appealing to let the compiler parallelize the simulation code automatically. This thesis investigates techniques of using automatic translation of models in typical equation based languages, such as Modelica, into parallel simulation code that enable high utilization of available processors in a parallel computer.

The two main ideas investigated here are the following: first, to apply parallelization simultaneously to both the system equations and the numerical solver, and secondly, to use software pipelining to further reduce the time processors are kept waiting for the results of other processors. Prototype implementations of the investigated techniques have been developed as a part of the OpenModelica open source compiler for Modelica. The prototype has been used to evaluate the parallelization techniques by measuring the execution time of test models on a few parallel architectures and to compare the results to sequential code as well as to the results achieved in earlier work. A measured speedup of 6.1 on eight processors on a shared memory machine has been reached. It still remains to evaluate the methods for a wider range of test models and parallel architectures.

Nyckelord
Automatic parallelization, Equation-Based Languages, Modelica
No 17  Vojin Plavsic: Interleaved Processing of Non-Numerical Data Stored on a Cyclic Memory. (Available at: FOA, Box 1165, S-581 11 Linköping, Sweden. FOA Report B30062E)
No 73  Ola Strömfl: A Structure Editor for Documents and Programs, 1986.
No 177  Peter Åberg: Design of a Multiple View Presentation and Interaction Manager, 1989.
No 180  Henrik Eriksson: A Study in Domain-Oriented Tool Support for Knowledge Acquisition, 1989.
No 380  Johan Ringström: Compiler Generation for Parallel Languages from Denotational Specifications, 1993.


Dan Fristedt: Metoder i användning - mot förbättring av systemutveckling genom situationell metodkunskap och metodanalys, 1995.


Magnus Werner: Multidatabase Integration using Polymorphic Queries and Views, 1996.

Mikael Lind: Affärsprocessinriktad förändringsanalys - utveckling och tillämpning av synsätt och metod, 1996.


Kristina Larsen: Förutsättningar och begränsningar för arbete på distans - erfarenheter från fyra svenska företag, 1996.


Niclas Andersson: Compilation of Mathematical Models to Parallel Code, 1996.


Jörgen Lindström: Chefer av användningen av kommunikationsteknik, 1996.


Fredrik Eklund: Declarative Error Diagnosis of GAPLog Programs, 1997.
Mats Gustafsson: Bringing Role-Based Access Control to Distributed Systems, 1997.
Bengt E W Andersson: Samverkande informationssystem mellan aktörer i offentliga åtaganden - En teori om aktörsarenor i utbyte av information, 1998.
Charlotte Björkegren: Learning for the next project - Bearers and barriers in knowledge Transfer within an organisation, 1999.
Svein Bergman: Managerial communication in telework, 2000.


Jean Paul Meynard: Control of industrial robots through high-level task programming, 2000.


Magnus Kald: The role of management control systems in strategic business units, 2000.


Per Oscarsson: Informationssäkerhet i verksamheter - begrepp och modeller som stöd för förståelse av informationssäkerhet och dess hantering, 2002.


Jon Edvardsson: Contributions to Program- and Specification-based Test Data Generation, 2002


Fredrik Elg: Ett dynamiskt perspektiv på individuella skillnader av heuristisk kompetens, intelligens, mentala modeller, mål och konflikter i kontroll av mikrovärlden Moro, 2002.


| No 1293 | Jiri Trnka: Prerequisites for data sharing in emergency management, 2007. |
| No 1356 | Erik Kuiper: Mobility and Routing in a Delay-tolerant Network of Unmanned Aerial Vehicles, 2008. |
| No 1361 | Martin Karresand: Completing the Picture - Fragments and Back Again, 2008. |
| No 1363 | Per Nyblom: Dynamic Abstraction for Interleaved Task Planning and Execution, 2008. |