Linköping Studies in Science and Technology

Licentiate Thesis No. 1716

# Automatic and Explicit Parallelization Approaches for Mathematical Simulation Models

by

**Mahder Gebremedhin**

National Graduate School of Computer Science

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

Linköping 2015

This is a Swedish Licentiate's Thesis

Swedish postgraduate education leads to a doctor's degree and/or a licentiate's degree.
A doctor's degree comprises 240 ECTS credits (4 year of full-time studies).
A licentiate's degree comprises 120 ECTS credits.

ISBN 978-91-7519-048-8
ISSN 0280–7971
Printed by LiU Tryck 2015

URL: http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-117338

## Abstract

The move from single-core processor systems to multi-core and many-processor systems comes with the requirement of implementing computations in a way that can utilize these multiple computational units efficiently. This task of writing efficient parallel algorithms will not be possible without improving programming languages and compilers to provide the supporting mechanisms. Computer aided mathematical modeling and simulation is one of the most computationally intensive areas of computer science. Even simplified models of physical systems can impose a considerable computational load on the processors at hand. Being able to take advantage of the potential computational power provided by multi-core systems is vital in this area of application. This thesis tries to address how to take advantage of the potential computational power provided by these modern processors in order to improve the performance of simulations, especially for models in the Modelica modeling language compiled and simulated using the OpenModelica compiler and run-time environment.

Two approaches of utilizing the computational power provided by modern multi-core architectures for simulation of Mathematical models are presented in this thesis: *automatic* and *explicit* parallelization respectively. The *automatic* approach presents the process of extracting and utilizing potential parallelism from equation systems in an automatic way without any need for extra effort from the modelers/programmers. This thesis explains new and improved methods together with improvements made to the OpenModelica compiler and a new accompanying task systems library for efficient representation, clustering, scheduling, profiling, and executing complex equation/-task systems with heavy dependencies. The *explicit* parallelization approach allows utilizing parallelism with the help of the modeler or programmer. New programming constructs have been introduced to the Modelica language in order to enable modelers to express parallelized algorithms. The OpenModelica compiler has been improved accordingly to recognize and utilize the information from these new algorithmic constructs and to generate parallel code for enhanced computational performance, portable to a range of parallel architectures through the OpenCL standard.

*This work has been supported by Vinnova in the RTSIM and ITEA2 OPEN-PROD and MODRIO projects, the Swedish Strategic Research Foundation (SSF) in the Proviking EDOP project, by the National Graduate School of Computer Science (CUGS), and by the ELLIIT project. The Open Source Modelica Consortium supports the OpenModelica project.*

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

**Acknowledgements**

First and foremost I would like to thank my main supervisor Professor Peter Fritzson for believing in me and giving me the chance to work with him. Without his trust and support I wouldn't have been able to do any of this. I would really like to thank him for providing me with an environment that is based on guidance and support rather than strict supervision.

I would also like to thank all my colleagues at PELAB for the nice environment they have created. All the coffee breaks and interesting discussions. Special thanks goes to all the people working tirelessly on improving the OpenModelica compiler. Without them this thesis work would not have been possible. Thanks for all the bug fixes, suggestions and pointers which have made my work possible. I would like to thank my co-supervisors Professor Christoph Kessler and Assistant Professor Adrian Pop for all their help and support.

I would also like to thank all the administrative staff. Especially Anne Moe for her patience and support, from explaining even the simplest things and going through all the details all the way to this thesis. My deepest thanks. I would also like to thank Åsa Kärrman and Eva Pelayo Danils for all the help with various matters over the years.

Finally I would like to thank my family. Even if far away, you have given me the will and determination to go through the years. If I didn't know that you are all behind me I wouldn't have been able to go forward. Thank you everyone.

Mahder Gebremedhin, April 2015

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Motivation

Build faster processors. This used to be the way to get computations done faster in the past. You get the latest fastest processor in the market and that was all you needed to do. Lately, however, with the power requirements of yet faster single processors becoming highly uneconomical the trend is instead towards building many smaller processors and then distribute our heavy computations between them. The move from single-core and single-processor systems to multi-core and many-processors systems comes with the extra requirement of implementing computations in a way that can utilize these multiple computational units efficiently. This task of writing efficient parallel algorithms will not be possible without improving programming languages and compilers to provide the mechanisms to do so. In recent years substantial research effort is being spent on providing such mechanisms. This thesis work is one of the efforts. In this work we investigate how the available potential parallelism in Mathematical models can be used for efficient parallel computation.

Computer aided mathematical modeling and simulation is one of the most computationally intensive areas of computer science. Even simplified models of physical systems can impose a considerable computational load on the processors at hand. Being able to take advantage of the potential computation power provided by modern multi-core and many-processor systems is vital in this application area.

Equation-based Object Oriented languages like Modelica provide a very convenient way of modeling real world cyber-physical systems. Object orientation gives these languages the power to hierarchically model physical systems. This allows reuse and flexible modification of existing models and enables users to provide increasingly complex models by building on existing components or libraries. Complex models in turn require a lot of computational power to be conveniently usable. This is why parallelization in modeling and simulation is an area that needs extensive investigation. Simulation of ever more complex models will become too time consuming without efficient automatic and explicit methods of harnessing the power of multi-core processors.

In this thesis work we have studied the problem of utilizing parallelism in the context of the Modelica equation-based object-oriented language and the OpenModelica model compiler and run-time environment. Multiple parallelization approaches have been studied in the past by the Programming Environments Laboratory (PELAB) here at Linköping University where the OpenModelica compiler is being actively developed. Most of these past parallelization approaches were concerned with automatic extraction of parallelism from Modelica models. There have been different prototype implementations that tried to provide automatic parallelization of simulations on multi-core CPUs as well as GPUs. Some of them were capable of simulating full physical models with no restrictions while others had certain restrictions on the system e.g. restrictions on the Modelica source code, restrictions on the solvers that can be used, etc. Unfortunately some of these implementations were rather obsolete by the time these thesis work started due to lack of maintenance or just simply because they were not relevant anymore due to continuous changes to the OpenModelica compiler and recent improvements in parallel programming arena. Other recent parallelization attempts are operational but differ in several ways from the work presented in the thesis. More information on these parallelization implementations and methods is given in the related work Sections 3.3 and 4.3.

This thesis work presents two different but inter-operable approaches to parallelization of Modelica models evaluated on implementations in the OpenModelica compiler. Many of these results are valid for EOO languages and environments in general. The two approaches are:

- Automatic parallelization of equation-based models
- Explicit parallelization of algorithmic models

The first parallelization approach is a task-graph based implementation concerned with automatically extracting and utilizing parallelism from complex equation systems. This is a very compelling approach due to the fact that it can handle existing models and libraries without any modification. The method and implementation mainly consists of two different parts: a dependency analysis and parallelization extraction phase and a run-time task system handling and parallelization phase. The dependency analysis phase is rather specific to the compiler and simulation environment at hand, in this case OpenModelica. The runtime task-system handling and parallelization part, on the other hand, is implemented as an independent C++ library and can be used in any other simulation environment as long as the dependency information is readily available.

The explicit parallelization approach is more language and compiler specific. This approach introduces new explicit parallel programming constructs, for example parallel for-loops, to the Modelica language, implemented as extensions of the OpenModelica compiler. Using these extensions, users can write explicitly parallel algorithmic Modelica code to run on multi-core GPUs, CPUs, accelerators and so on. Even though this approach requires users to write their algorithmic Modelica code in specific ways, the effort is usually worthwhile since they can achieve much higher performance improvements for suitable algorithms. Moreover explicit parallel programming means that users are expected to have some knowledge of parallel programming. This might be an issue for modeling language users who are usually experts in fields other than computer science. However, with the increasing prevalence of multi-core processors, some knowledge of parallel programming is bound to be a necessity for anyone working with any programming language. The explicit parallel programming extensions are not yet standard Modelica and are currently only available for users of the OpenModelica compiler. To our knowledge there is no other Modelica tool that provides similar features at the moment of this writing.

## 1.2 Main Contributions

The main contribution of these thesis work are:

- Design and implementation of new automatic parallelization support for the OpenModelica compiler.
- Design and implementation of a highly flexible, efficient and customizable task system handling library with several clustering and scheduling options.
- Design, implementation, and evaluation of the explicit ParModelica algorithmic extensions to allow the Modelica language to take advantage of modern multi-core and multi-processor architectures.

## 1.3 Limitations

There were some technical limitations that affected the implementations done in this thesis work. One of the biggest limitations for the explicit parallelization approach is that the compilers and tools used to compile the generated OpenCL code are very restrictive. OpenCL is based on the C99 standard (ISO/IEC 9899:1999) [28] with many restrictions. For example only a few headers from the standard C library can be used in an OpenCL program. The OpenModelica compiler runtime requires many complex operations to be fully operational. This means that it makes quite heavy use of the C header files as well as other utility libraries.

In order to make sure that generated parallel OpenCL code is compilable while maintaining inter-operability with the normal sequential runtime environment has required many compromises. However, recently there have been some C++ construct extensions of the core OpenCL language provided by some hardware vendors e.g. the OpenCL Static C++ Kernel Language Extension from AMD [1]. These extensions can be used to improve the current implementation and can several issues. However, being vendor specific means that they are only available on certain architectures and are not really fully portable.

## 1.4 Thesis Structure

The thesis starts by providing a quick common background information on modeling and simulation as well as on parallel programming in general in Chapter 2. The rest of the thesis consists of two main topics: chapters dedicated to automatic and explicit parallelization approaches, respectively.

The first part, Chapter 3, presents the methods and approaches used to extract and implement automatic parallelization in equation based task systems. A brief explanation of dependency analysis and extraction of parallelism from highly connected equation systems is presented. Then the features and implementation of Task System Library used for parallelization of the resulting task systems are explained in detail. These include the clustering algorithms, schedulers, profiling and cost estimations methods and so on. This part of the thesis is partly based on the following two papers:

- Mahder Gebremedhin and Peter Fritzson
  *Automatic Task Based Analysis and Parallelization in the Context of Equation Based Languages*
  Proceedings of the 6th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, (EOOLT'2014), Berlin, Germany, October 9, 2014.

- Martin Sjőlund, Mahder Gebremedhin and Peter Fritzson
  *Parallelizing Equation-Based Models for Simulation on Multi-Core Platforms by Utilizing Model Structure*
  17th International Workshop on Compilers for Parallel Computing (CPC 2013), Lyon, France, July 3-5, 2013, 2013.

The second part, Chapter 4, presents and explains the ParModelica algorithmic language extensions. The design of these constructs is inspired by OpenCL and is implemented as an extension of the Modelica language supported by the OpenModelica compiler. The extensions and the available mechanisms for runtime support of this explicit parallelization approach are explained in this part of the thesis and are based partly on these two papers:

- Mahder Gebremedhin, Afshin Hemmati Moghadam, Kristian Stavåker and Peter Fritzson

*A Data-Parallel Algorithmic Modelica Extension for Efficient Execution on Multi-Core Platforms*
Proceedings of the 9th International Modelica Conference (Modelica 2012), Munich, Germany. 2012.

- Afshin Hemmati Moghadam, Mahder Gebremedhin, Kristian Stavåker and Peter Fritzson
  *Simulation and benchmarking of Modelica models on multi-core architectures with explicit parallel algorithmic language extensions*
  Fourth Swedish Workshop on Multi-Core Computing MCC-2011, 2011.

Corresponding introductions, background information and previous work are presented in each part of the thesis.

Other publications by the author not used in this thesis work but are related to modeling and parallelization are:

- Alachew Shitahun, Vitalij Ruge, Mahder Gebremedhin, Bernhard Bachmann, Lars Eriksson, Joel Andersson, Moritz Diehl and Peter Fritzson
  *Model-Based Dynamic Optimization with OpenModelica and CasADi*
  IFAC-AAC 2013, 2013.

- Bachmann, Bernhard, Lennart Ochel, Vitalij Ruge, Mahder Gebremedhin, Peter Fritzson, Vaheed Nezhadali, Lars Eriksson, and Martin Sivertsson.
  *Parallel multiple-shooting and collocation optimization with OpenModelica.*
  In Proceedings of the 9th International Modelica Conference (Modelica 2012), Munich, Germany. 2012

# Chapter 2

# Background

## 2.1 Modelica

Modelica [46] is a non-proprietary, object-oriented, equation based, multi-domain modeling language for component-oriented modeling of complex physical systems containing, e.g., mechanical, electrical, electronic, hydraulic, thermal, control, electric power or process oriented subcomponents. Its development is managed by the non-profit Modelica Association [47]. The Modelica Association also overlooks the development of the open source Modelica Standard Library which contains components and example models from different application domains.

### 2.1.1 Modelica for Mathematical Modeling

Modelica is quite well suited for modeling of complex cyber-physical systems. This is not surprising since the language was designed and continuously improved for that specific purpose. Like many other complex object oriented languages Modelica has classes, support for advanced features like inheritances (extends), operator overloading, generic programing (redeclarations) and so on. However, what makes Modelica especially well suited for modeling is its ability to capture physical systems in an intuitive way.

Modelica is an Object Oriented language. This means that it can be used to create models in a hierarchical manner by combining and extending different model components. `Connector` classes, connectors and `connect` equations enable modelers to create relationships and interactions between these components in an intuitive way.

Probably the most important feature that makes of Modelica so powerful is its support for acausal modeling. Modelica allows equations in addition to assignment statements which are common in most programming languages. Assignments always have their left-hand sides as outputs, i.e., variables assigned to. Equations, on the other hand, do not specify which variables are inputs and which are outputs. It is the specific Modelica compiler's job to sort equations in data-flow order and generate the causal structure for the system. This makes for a very flexible modeling environment where models can express physical systems in a natural way. A brief overview of this symbolic manipulation process is presented in Section 3.4.

In addition, having different specialized classes such as `models`, `records`, `blocks`, `connectors`, etc., gives Modelica the ability to represent physical components in a way that resembles their real world attributes. For example `record` classes are used to represent a collection of data about a physical component e.g. A `record` class for a point in space contains the x,y, and z co-ordinates of the point. Model classes are used to represent components with dynamic behaviors. A `model` class for a rocket can contain the variables that represent its dynamic behavior, e.g., its current mass, velocity, acceleration, etc together with the equations that govern the behavior of the rocket. Connector classes and connect equations enable modelers to create relationships and interactions between components in an intuitive way.

There are many other Modelica features that are interesting for physical modeling. However we will not go in to a detailed explanation here since these features are all well documented. Very detailed explanations and examples can be found in [18], [42] among many others.

## 2.1.2 Modelica for Scientific Computing

A rather overlooked application area of Modelica is its usability in general scientific computations. The language, as it is right now, is quite well suited to be used in heavy scientific computations and not just in modeling areas. Most scientific computation algorithms involve linear algebra operations on large amounts of data usually organized as vectors, matrices, and higher dimensional arrays. Modelica provides a very powerful array representation and related features that can make writing these complicated algorithms more convenient. It is of course not possible to cover all the features Modelica provides for convenient *algorithmic* code implementation here. However, this section presents a few selected features that can give a general idea of what Modelica has to offer for the scientific computation community.

Modelica arrays can be declared with unknown sizes, specified by colons as dimension sizes, as shown in Listing 2.1. Here the size of the array `x` which is the input to the function `unknowInputSizeArray` is not specified. The actual size is of this input array is determined at call time by the argument passed to it, as demonstrated by the calls in the function `callMultiple`. The output array y of the function is also flexible. Its size is determined by the size of the first dimension of the input array via the call *size(x,1)*.

```
function unknowInputSizeArray
  input Real x[:];
  output Real y[size(x,1)];
...
```

```
end unknowInputSizeArray;

function callMultiple
...
protected
  Real[2] x2,y2;
  Real[3] x3,y3;
algorithm
  x2 := ones(2);
  y2 := unknowInputSizeArray(x2);
  x3 := ones(3);
  y3 := unknowInputSizeArray(x3);
...
end callMultiple;
```

Listing 2.1: Unknow size arrays

This kind dynamic adjustment of array sizes makes functions more generic and reusable. For example without the ability to work with unknown size arrays we would need one function for each size to perform the same operation.

Indexing and Slicing: there are many of ways to index Modelica arrays. Indexing an array can result in a scalar or an array of selected elements of the original array. The latter is what we call array slicing. Simple examples of array slicing operations are shown in Listing 2.2.

```
function SliceTest
  input Real x[10,10];
...
protected
  Real y[10] := x[1,:]; // y is the first row of Matrix x
  Real z[10] := x[:,1]; // z is the first column of Matrix x
  Real a[3,10] = x[1:3, :] // indexing with range. The first 3 rows
      of x
  Real b[3,10] = x[{1, 3, 5}, :] // indexing with array. The 1st,
      3rd and 5th rows of x

  y[2:10] := y[1:9]; // shifting elements of y 1 index to the left.
  for i in 1:9 loop // equivalent for loop implementation.
    y[i+1] := y[i];
  end loop;
end SliceTest;
```

Listing 2.2: Array slicing

Slicing allows algorithms to be more concise and provides the opportunity to take advantage of code structure to generate a more efficient code. More complex usages of slicing than shown in this examples enable modelers/programmers to manipulate arrays in a convenient way.

Modelica overloads the normal built-in arithmetic operators (+,-,*,/,ˆ) for vector, matrix and array arithmetical operations. Depending on the type of operands involved in the arithmetic operations these operators are resolved to the specific mathematical operation. For example

$$C = A * B$$

is resolved as matrix multiplication if A and B are matrices and the multiplication dimensions match ($m$x$n*n$x$k$) and results in matrix C($m$x$k$). This kind of resolving of operators is done for more combinations of operands. The full list can be found in [45].

In addition to overloading the common arithmetic operators, Modelica also provides a set of *element-wise* operators for arrays. These operators operate on an element by element basis on arrays of matching dimensions and dimension sizes. For example

$$C = A. * B$$

is an element by element multiplication of A($m$x$n$) and B($m$x$n$) which results in the matrix C($m$x$n$). The full list and semantic rules of the element-wise operators can be found in [45].

Yet another interesting example of using Modelica for computations is the use of range expressions and reduction operations for concise and readable representation of algorithms. Consider that we want to compute the sum of the first n odd numbers starting from 1. That is

$$S = \sum_{i=1}^{n} (2 * i - 1) \tag{2.1}$$

Using for ranges expressions and the built-in sum reduction operator we can write this in Modelica as

```
S = sum(2i-1 for i in 1:n)
```

Listing 2.3: Array slicing

In addition to these array and range related features the object oriented features of Modelica can help to further simplify scientific computation algorithms. For example records (which, in some extents are, the Modelica equivalent of C++ classes) with operator overloading can be used to manipulate structured data sets in a very convenient manner. A good example of this usage is the Complex numbers library from the Modelica Standard Library.

There are many more simple and advanced features that make Modelica very suitable for algorithms of scientific computations. Yet it seems like Modelica has been rather overlooked by the scientific computation community so far. This might be due to two main reasons:

- Modelica is originally intended for physical system modeling. Hence the focus of the user community is more on what it has to offer for modeling. Being such a powerful language for modeling has somehow over shadowed its convenience and power in the other areas. Even modelers who use Modelica frequently seem to use other languages e.g. Matlab when they have the need to do some sort of complex scientific computation.

- With regard to scientific computations Modelica has a crippling lack of library support. The only substantial library for scientific computations that is available is the *Modelica.Math* library which mostly provides interfaces to external LAPACK routines. This lack of library support might be the main reason why even frequent Modelica users prefer other languages and tools for their computations.

. This brings us to the second possible reason which is that

The explicit data-parallel programming extensions presented in these thesis (Chapter 4) provide an even further improvement on how Modelica can handle complex heavy computations on modern multi-core and multi-processor architectures.

## 2.2 Modelica Standard Library (MSL)

The Modelica Standard Library [48] provides model components in many domains that are based on standardized interface definitions. It is available freely and usually is bundled with many Modelica tool distributions. The library is quite extensively used and well tested. Selected models from this library are used to test the performance of the implementations presented in this thesis. MSL version 3.2 is used in this work.

## 2.3 OpenModelica

OpenModelica [1] is an open-source Modelica-based modeling and simulation environment intended for industrial and academic usage. Its long-term development is supported by a nonprofit organization The Open Source Modelica Consortium (OSMC) [10]. The Programming Environments Laboratory (PELAB) at Linköping University, together with OSMC, is developing the OpenModelica Compiler (OMC) for the Modelica language (including the MetaModelica [51], ParModelica [19] and Optimica [4] extensions) and the accompanying simulation environment.

The research prototypes developed in this thesis work are all done in the OpenModelica Compiler. The implementations and additions to the compiler presented here are have been developed within several phases of the compiler. This is a direct consequence of the specific parallelization paradigm and programmability intended in each investigation. A rough depiction of the OpenModelica compiler's compilation phases is shown in Figure 2.1.

The automatic task parallelization approach presented in Chapter 3 required some modifications to the Back-end and Code Generator stages of the compiler. However, most of the work for this approach was put into the runtime support of parallel execution by providing the rather independent Task Systems Library presented in Section 3.5.

On the other hand, the explicit parallel programming extensions presented in Section 4 required modification to almost all phases of the compiler starting from the parser all the way to the code generation and runtime support. Obviously adding new constructs to a language means that the compiler will have to recognize the new constructs, make sure any syntactic and semantic rules for these constructs are obeyed, generate appropriate code for the usage, and finally provide runtime support for any new features required. The compiler has been modified for these changes.

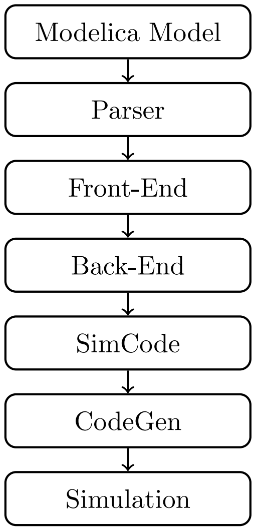


Figure 2.1: OpenModelica Compiler Compilation Phases

## 2.4 Parallel Programming

Parallel programming is concerned with the simultaneous or parallel use of multiple computational units or resources to solve a given computational problem. A given computational problem can be broken down into smaller less computationally intensive problems and computed on different processing units with a system wide control of problem structures and coordination.

There are many different paradigms and flavors of parallel programming in existence today. Especially in recent years, with the advent of widespread availability of multi-core and multi-processor architectures, researchers are in a rush to provide and utilize even more efficient and powerful paradigms and implementations.

Of course there is no universally best solution to all the computational problems that exist in the physical world. Different kinds of applications require different approaches and implementations to take full advantage of the computing power of the available resources. Moreover, different processor architectures are suited for different paradigms and approaches.

User preferences and programmability are other important characteristics that are influencing the development of these parallel programming paradigms. Some approaches are intended for advanced users with a good knowledge of the problems of parallel programming who are looking to take the last drop of performance out of the computational resources available to them. Others approaches are intended for less experienced users looking for a quick and efficient way of improving the performance of their computations.

Within the scope of this thesis, we categorize parallel programming approaches in two ways. The first categorization is concerned with the programmability of the approach from a user's perspective. How will users be able to take advantage of the potential parallelization? Do they need to write their programs in a specific way? Will they have to modify existing code to take advantage of the method? The second categorization is concerned with the type of threading model or the types computations the approach is suitable for. Some parallelization paradigms are geared towards performing the same operations on a large amounts of shared data sets while others are intended for performing possibly different tasks on possibly distributed data sets.

### 2.4.1 Programmability

With regards to programmability, we can classify parallelization approaches as *automatic* parallelization and *explicit* parallelization. These approaches are briefly explained in the next sections.

#### 2.4.1.1 Automatic Parallelization

Automatic parallelization is the process of automatically detecting, optimizing, and parallelizing a computation. This parallelization method involves enhancements to compilers while the language stays the same. It imposes no or a quite small amount of work from the user's perspective. Users would not have to write their model or code in any different way than they would with no parallelization in mind. The compiler has full responsibility of finding and utilizing any potential parallelism from the user's model or algorithm.

Improving existing compilers for supporting automatic parallelization requires a considerable effort. However, it is naturally the most preferred way for end-users since it enables them to use models and algorithms without having to learn the details of complicated parallel programming languages. This is especially useful for communities like Modelica where most users working with the language/compiler are experts in a different field than Computer Science.

Another advantage of automatic parallelization approaches is that they allow the parallelization of existing code. The possibility of parallelized execution of existing libraries and implementation without any need for changes is quite appealing. Of course some sort of consideration might need to be made when writing code in order to assist the compiler with better extraction of parallelism.

#### 2.4.1.2 Explicit Parallelization

Explicit parallelization, unlike automatic parallelization, is based on users explicitly stating where, when, and how their code should be parallelized. Explicit parallelization requires modifications to the compiler as well as to the language itself, i.e., if it doesn't have support for explicit parallelization yet, which currently is the case for the standard Modelica language.

To utilize this kind parallelization users have to write programs that uses these constructs explicitly where parallelism is needed. This means that users need to have some knowledge and expertise about how to write efficient parallel code.

Despite the fact that users have to spend extra effort in developing their programs to utilize explicit parallelism, this can result in huge performance improvements for many kinds of algorithms. Humans usually have a better understanding of their algorithms than the compilers. By implementing their programs or models in an optimized explicit way, they can achieve higher performance gains than the compiler would have done automatically.

### 2.4.2 Threading Model

With regard to threading models, we can classify parallelization approaches into data parallel and task parallel methods. These are briefly presented in the next sections.

There is no clear-cut distinction between these two models of parallel computation. Computations are rather loosely attributed to each parallelization model based on how close they resemble the corresponding pure model.

#### 2.4.2.1 Data Parallelism

Data parallelism involves multiple computational units simultaneously performing the same or very similar operations to different items/parts of a given data set. Ideally this would involve every processing unit involved in the computation performing exactly the same operation on different data elements. A good example of data parallelism would be a simple element-wise addition of two vectors where each processing unit performs addition on the corresponding single elements from each the two arrays. Of course not all processing units can perform exactly the same operation on different data for all physical problems. There are many cases where a few selected units might be doing additional operations or fewer operations based on the specifics of the problem at hand.

Data parallel programing usually involves operations on data structured into arrays and different processing units operating on these arrays. Since these operations are mostly similar there are not as many parallel control structures (barriers, synchronizations) needed as task parallelism.

Most parallel architectures are designed with heavy data parallelism requirements in mind. This is especially true in recent years with the ever-increasing power and complexity of multi-core CPU and GPU architectures.

#### 2.4.2.2 Task Parallelism

Task parallelism lies at the other end of the spectrum compared to data parallelism. In an ideal task parallel operation each involved computational unit performs a potentially completely different operation on the same or different data compared to other units. This is in contrast to data parallelism where all units perform the same operation on different parts of the data set.

Some task parallel algorithms can be considered as a special case of data parallelism. In the element wise addition example given for data parallelism there are usually different operand pair values for each addition operation on corresponding array elements. Now consider a given task parallel application. Assume that our data is an array of tasks. Each computing unit takes on task from these array and operates on it. Just like the values of the array members for the element-wise addition, the members of the task array have different values. From an abstract point of view it is possible to consider this “task parallel” algorithm as “data parallel” with data consisting of tasks.

### 2.4.3 Combined Parallelism: Programmability with Threading Model

The two classifications of parallelization models explained above can be combined and used to take advantage of different algorithms. For example, it is possible and common to have compilers extract data parallelism automatically or to have users write explicit task parallel programs.

In this work the explicit parallel programming extensions were designed mainly for data parallel operations. This is a direct consequence of the programming model they mimic, which is OpenCL. However, users can of course use some of these extensions to implement their task parallel algorithms.

On the other hand, the automatic parallelization methods and implementation presented here are, currently, limited to extracting task parallelism

| | Automatic | Explicit |
|---|---|---|
| Data Parallel | | ▨ |
| Task Parallel | ▨ | ▨ |

Table 2.1: Supported Parallelism

from complex equation systems. This does not mean that parallelization is always done as task parallelism. It simply means that right now the compiler doesn't look for and does not extract possible data parallel operations from a given algorithm in a Modelica model. It is only concerned about finding dependencies at an equation level which can affect the dependency relationship for parallelization purposes. How the extracted parallelism is utilized is a different matter. Depending on the kind of scheduler and executor this task parallelism can be converted to a data parallel approach with tasks as data. The Level Scheduler implementation presented in Section 3.5.5.1 is a good example. Here clustered tasks within each level are represented as arrays of tasks and a simple data parallel iteration loop is used to execute this task array.

Table 2.1 shows what kinds of parallelism can be used with or are extracted by OpenModelica compiler for Modelica models at the moment of this writing only based on this work alone. To summarize:

- Users can write explicitly data parallel or task parallel algorithms.
- The compiler can currently extract task parallelism automatically from equation systems.

It is rather straight-forward to implement the missing parallelization which is automatically extracting data parallelism. For example it should be rather easy to locate arithmetic expressions like element-wise multiplication of two arrays which can benefit from data parallelism. Once these operations have been extracted the runtime functionality already available for the explicit data parallel implementation can be used to perform the rest of the work.

# Chapter 3

# Automatic Parallelization

## 3.1 Introduction

This chapter presents our task-graph based automatic parallelization design and implementation for handling complex task systems with heavy dependencies. Methods for analyzing dependencies, representing them in a convenient way and processing the resulting task graph representations are presented. We present a library based task system representation, clustering, profiling and scheduling approach to simplify the otherwise tedious and complicated process of parallelizing complex task systems. The implementation offers a flexible and robust task system handling library to manipulate and parallelize these complex task systems on shared memory multi-core and multi-processor systems.

A brief background information on some fundamental scheduling problems, the algorithms used for dealing with these problems, and 3rd party tools used in the implementation is first provided. Then a simple approach of extracting dependency information from equation systems and representing them in a convenient graph based system is presented. The core ParModelica Task System library is presented next, in Section 3.5, with explanations of the different clustering, scheduling and profiling algorithms and options.

## 3.2 Background

### 3.2.1 The Bin Packing Problem

The Bin Packing problem is a classical optimization problem that deals with partitioning a set of items into as few bins as possible while making sure that the total sum of some selected attribute of all items packed into the same bin does not exceed a specified value. The problem can be formally defined as:

Given a set of items $\boldsymbol{C} = \{c_1, c_2 \ldots, c_n\}$ where $0 < c_i \leq 1, \forall c_i \in \boldsymbol{C}$ and a set of *bins* $\boldsymbol{B} = \{b_1, b_2 \ldots, b_m\}$ with *capacity* of 1

Find a mapping $\boldsymbol{C} \rightarrow \boldsymbol{B}$ so that the number of non-empty bins is minimized.

The Bin Packing problem is interesting in clustering applications where there are cost considerations of tasks. We are often interested in collecting or merging tasks into clusters based on some proximity criteria while making sure that the overall cost of the resulting cluster does not exceed a given cost limit.

Bin packing is an NP-hard problem [30] [5]. However there are constant factor approximate heuristics for the problem. Three of these algorithms are presented briefly below.

### 3.2.2 Next fit, First Fit, and First Fit Decreasing Heuristics

The Next Fit(NF), First Fit(FF) and First Fit Decreasing(FFD) algorithms provide approximate solutions for the Bin Packing problem. Consider a given the set of items $\boldsymbol{C} = \{c_1, c_2 \ldots, c_n\}$ where $0 < c_i \leq 1, \forall c_i \in \boldsymbol{C}$ and a set of *bins* $\boldsymbol{B} = \{b_1, b_2 \ldots, b_m\}$ each with a *capacity* of 1. Assume that all bins are initially empty:

- NF: start by adding the first item, $i_1$, to the current bin $b_i$ (initially $b_1$). Then consider the rest of the items one after another and add them to the current bin if it has enough capacity. If at any point the item being considered doesn't fit in to the current bin, open the next bin, $b_{i+1}$, and repeat the same process until there are no more items left.

- FF: start by adding the first item to the current bin $b_i$ (initially $b_1$). Iterate over all remaining items adding them to the current bin if it has capacity. If there are no more items left that can fit to the current capacity of the current bin then open the next bin, $b_{i+1}$, and repeat the same process until there are no more items left.

- FFD: Sort the items of set $\boldsymbol{C}$ in descending order into set $\boldsymbol{S}$ and apply FF to $\boldsymbol{S}$.

The First Fit Decreasing is the best-possible approximation algorithm for Bin Packing [31] [30] [5]. FFD is the algorithm used as part of some of the clustering algorithms implemented in this thesis work. An explanation of the packing problem in the context of task systems for mathematical equation systems and adaptation of the FFD algorithm is presented in Section 3.5.3.

### 3.2.3 k-way Linear Partitioning Problem

The k-way Linear Partitioning problem is a complementary problem to Bin Packing. The general problem is to try and partition a given set of items into a specified number of disjoint sets while a selected objective is minimized. Formally:

> Given a set of items $\boldsymbol{C} = \{c_1, c_2 \ldots, c_n\}$, find a mapping of $\boldsymbol{C}$ in to $k$ disjoint sets $B_1, B_2, \ldots, B_k$ where $(B_1 \cup B_2 \cup \cdots \cup B_k) = \boldsymbol{C}$ and $B_i \cap B_j = \emptyset$ for $1 \leq i, j \leq k$, so that a given objective function $F(B^k)$ is minimized.

This linear partitioning problem is very similar to the Minimum Makespan Scheduling problem on identical processors. The general Makespan Scheduling problem can be defined as

> Given a set of $k$ machines $M = \{m_1, m_2, \ldots, m_k\}$ and $n$ jobs $J = \{j_1, j_2, \ldots, j_n\}$ where job $j$ takes $t_{ij}$ time units to execute to completion on machine $i$. If $J_i$ is the set of jobs scheduled on machine $i$ and the total load on machine $i$ is $L_i = \sum_{j \in J_i}$. The problem is to schedule the jobs so that the maximum load, $L_{max} = \max_{i \in M}$ is minimized.

If all machines are identical, i.e., $t_{ij} = t_j$ for all $i \in M$ and all $j \in J$, then the Minimal Makespan Scheduling problem is equivalent to a k-way linear partitioning where the objective function is the maximum load over the set of machines.

There are a number of approximate algorithms that aim to generate a schedule that is within some specified worst case bounds [22], [38], [23], [14]. One group of heuristics for generating approximate schedules is the List Scheduling based algorithms.

### 3.2.4 List and Sorted List Scheduling

The List scheduling class of heuristics has many variations depending on how priorities are assigned to the jobs. Jobs that are ready for execution are sorted into what is referred to as *ready list* according to some specific criteria of *priority.* These candidate jobs are assigned to a specific machine when it is available.

One of the simplest List Scheduling algorithm is Graham's List Scheduling [20]. Given a set of jobs and machines the algorithm considers the jobs in the original order and adds them to the *ready list.* Then it removes the first job from the *ready list* and assigns it to the machine with the least load at the moment. The load of a machine being the total run-time cost of all jobs assigned to it so far. Repeat the process until there are no more jobs left. Although rather simple, the notable advantage of this algorithm is that it is an *online* algorithm. Even if jobs arrive one after another and there is no knowledge about what jobs may arrive next or when they will arrive, the algorithm can still be applied.

This simple algorithm can be improved by introducing some ordering or priority to the jobs. The Sorted List Scheduling algorithm provides a better approximation [21] by first sorting the job list in decreasing order of cost. Once sorted the simple List Scheduling algorithm is applied.

The Sorted List Scheduling is used as part of the clustering implementations presented in this work with minor adaptations. Specifically as part of the MLC clustering algorithm. This is discussed later in Section 3.5.3.

### 3.2.5 Boost Graph

The Boost Graph Library (BGL, *boost::graph*) [9] is a generic library that provides advanced data structures and algorithm for conveniently implementing graph computations. It is an open source library which is developed by Boost [10] and is part of the Boost library suite which contains collections of different industry standard C++ libraries and tools. It is available under the Boost Software License [11] which encourages both commercial and non-commercial use.

The task-graph based parallelization presented in this chapter makes heavy use of the Boost Graph library. The Task Systems Library presented in Section 3.5 basically extends the Boost graph library and provides extra clustering, scheduling, and execution mechanisms for the graphs. The library, being built on top of Boost, tries to resemble and mimic the structures of the graph library as much as possible to keep the implementation as generic as possible and allow potential inter-operation with other Boost graph algorithms and implementations.

The OpenModelica simulation environment is dependent on a couple of Boost libraries other than the graph library. Hence the whole Boost library suite is already distributed together with the OpenModelica source code as part of the OMDev suite on Windows. On Unix systems users need to get Boost separately but it is rather straight-forward to setup and use.

### 3.2.6 Intel Threading Building Blocks

The Threading Building Blocks (TBB) [26] is a C++ template library developed by Intel. The library provides multiple data structures and algorithms that significantly simplify parallel programming in C++ compared to other native threading packages like POSIX Threads [8] or OpenMP [7]. It provides a flexible, generic and efficient implementation that has simplified the work in this thesis. It is available under the GPLv2 [17] license.

The ParModelica Task Systems library uses TBB for two different purposes. The primary use of TBB is for its flow-graph (*tbb::flow*) sub-library which is available in TBB 4.0 and later. The flow-graph library provides support for representing dependencies between tasks as messages passed between nodes. This flow-graph library is at the heart of the Flow Graph Based Scheduler presented in Section 3.5.5.2. In addition to its use for the flow-graph implementation, TBB is used throughout the Task System library to simplify and improve flexibility of other parallelization needs. It is not really mandatory to use TBB for these reasons. All these extra uses of TBB can be excluded and replaced by other native implementations (e.g. using POSIX Threads or boost:threads) if the dependency on TBB would not be desired any longer.

## 3.3 Related Work

Substantial previous research have has been done on automatic parallelization related to the OpenModelica compiler and the Modelica language in general. Perhaps the closest research work compared to the work described in this thesis is the *modpar* parallelization design and implementation [6]. The *modpar* implementation is based on task graphs and graph re-writing rules to merge tasks. It also used the Boost Graph library and implemented some similar clustering/merging algorithms. However there are two main differences between *modpar* and the work presented in this thesis:

- *modpar* was targeted towards distributed memory parallel architectures while the current implementation is for shared memory architectures. This makes modpar more general than the current implementation since it can also run on shared-memory implementations using shared-memory based message passing implementations.

- *modpar* built an initially very fine-grained task-graph, essentially one task for each expression node, which was merged into a more coarse-grained task using clustering algorithms. Building such a fine-grained task graph turned out to be very memory and computationally demanding. The approach was not scalable to large models. The approach taken in this thesis work is slightly less fine-grained. Tasks and dependencies are extracted at the equation level or blocks of equations level, see Section 3.4.1. This decreases the complexity and memory requirements of the resulting task system without losing significant parallelism potential.

- Clustering and Scheduling were done at compile time with *modpar*. In our work both scheduling and clustering stages are done at simulation time. This opens up opportunities for dynamic cost estimation which can improve task graph clustering outputs considerably and allows to perform dynamic rescheduling that can adaptively fit to the behavior of the system throughout simulation.

The modpar design and implementation was later extended to support pipelined parallelism at the solver level [41]. That kind of solver parallelism could also be potentially used together with our current approach

A recent automatic parallelization effort called HPCOM [53] at Dresden University of Technology also develops a similar task graph based approach.

Like modpar this approach performs all operations other than execution at compile time as part of the OpenModelica compiler. However HPCOM has tried to improve the estimation by implementing mechanisms to utilize profiling informations from previous simulations to create a more efficient scheduling outputs. While this should improve performance significantly, it is still not be possible to do the adaptive scheduling at runtime mentioned above .

A previous work by the author and colleagues also investigated a parallelization approach that utilized model structures and decoupling of equation systems with the help of transmission line modeling. This is presented in more detail in Section 3.4.3.

One more distinction of the current implementation compared to some of those mentioned above is that it is implemented a standalone flexible task system handling implementation that is not tied to a specific environment. It only expects very small interfacing consistencies from the OpenModelica compiler. This means that it is not affected by most of the changes to the compiler and can be maintained and improved separately. This might simplify the work of the developers who want to use the implementation as well as OpenModelica compiler developers who don't want to bother themselves with the parallelization issues in their daily work. On the other hand, there are also disadvantages with such a separate implementation choice. It might open the way to the implementation being overlooked with regard to changes to the rest of the OpenModelica environment. It is necessary to make sure that the implementation can efficiently handle and adapt to changes that affect the behavior of simulations. For example improvements to the OpenModelica back-end, e.g. new tearing algorithms, can affect the complexity of the resulting systems. The scheduling and clustering implementations of the library need to keep track of these changes and perform corresponding adjustments to benefit from them.

## 3.4 Mathematical Modeling

### 3.4.1 Equation Systems and Dependency Analysis

The OpenModelica compiler front-end accepts an object oriented acausal Modelica model as input and translates it to a flat Modelica model after performing a number of compilation phases like syntactic checks, instantiation, typing and type matching, etc. The compiler back-end takes this flat acausal model representation, sorts the equations, performs a number of optimizations, and creates a low-level model representation that is suitable for generating simulation code in some low level language (C, C++, C#, Java, etc.). The methods and algorithms of these transformations are explained in detail in [13] [18].

The most relevant part of this low level representation, for the sake of parallelization, is that it contains all the equations of the system divided into

| | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ |
|---|---|---|---|---|---|---|---|
| $f_1$ | ▨ | ▨ | | | | | |
| $f_2$ | | | ▨ | | | | |
| $f_3$ | ▨ | | ▨ | ▨ | | | |
| $f_4$ | | | ▨ | | ▨ | | |
| $f_5$ | ▨ | | | ▨ | ▨ | | |
| $f_6$ | | | | | | ▨ | |
| $f_7$ | | | | | | ▨ | ▨ |

Table 3.1: Original Incidence Matrix

sub-systems of equations (so-called strongly-connected components, SCCs). Most of these of sub-systems are simple assignment equations while others might be blocks or sub-systems of SCC equations that involve linear or nonlinear systems. This sorting and causalization of equations results in a system which can be represented as an incidence matrix of equations and variables in block lower triangular (BLT) form.

Consider the system of acausal equations shown in Equation 3.1. This system of equations can be represented with a structural incidence matrix where equations are represented by rows, variables are represented by columns and there is an entry with value 1 at location [i,j] if the i$^{th}$ variable appears in the j$^{th}$ equation. The incidence matrix for the system is shown in Table 3.1.

$$\begin{aligned} f_1(x_1, x_2, t) &= 0 \\ f_2(x_3, t) &= 0 \\ f_3(x_1, x_3, x_4, t) &= 0 \\ f_4(x_3, x_5, t) &= 0 \\ f_5(x_1, x_4, x_5, t) &= 0 \\ f_6(x_6, t) &= 0 \\ f_7(x_6, x_7, t) &= 0 \end{aligned} \tag{3.1}$$

Note that these equations are in implicit form and it is not yet known which equation can be used to solve for which variable. In order to be able to solve the system the compiler has to sort the equations and match them with the variables. The sorting process should result in a system of strongly connected components (SCCs) representing subsystems of equations. In simple cases such SCCs only contain a single-assignment equations for which it is explicitly known which variable to compute for in each equation.

One possible way to achieve this causalization and sorting of the system is by manipulating the incidence matrix. The idea is to convert the given incidence matrix in to a Lower Triangular Matrix (LT) if possible or to a Block Lower Triangular Matrix (BLT) otherwise. If the incidence matrix of the system is in LT form it can be guaranteed that every variable can be solved by the equation at the same level as itself, i.e., all SCCs contain only a single equation. In addition we make sure that no variable will be used before it is evaluated or computed.

An alternative is to represent the system as a bipartite graph and apply Tarjan's algorithm. The bipartite graph for the system in Equation 3.1 is shown in Figure 3.1. The algorithm starts by coloring all edges between equations and variables as black. It Initializes counters $i$ to 1 and $j$ to N where N is the number of equations in the system. Then applies these two rules recursively.

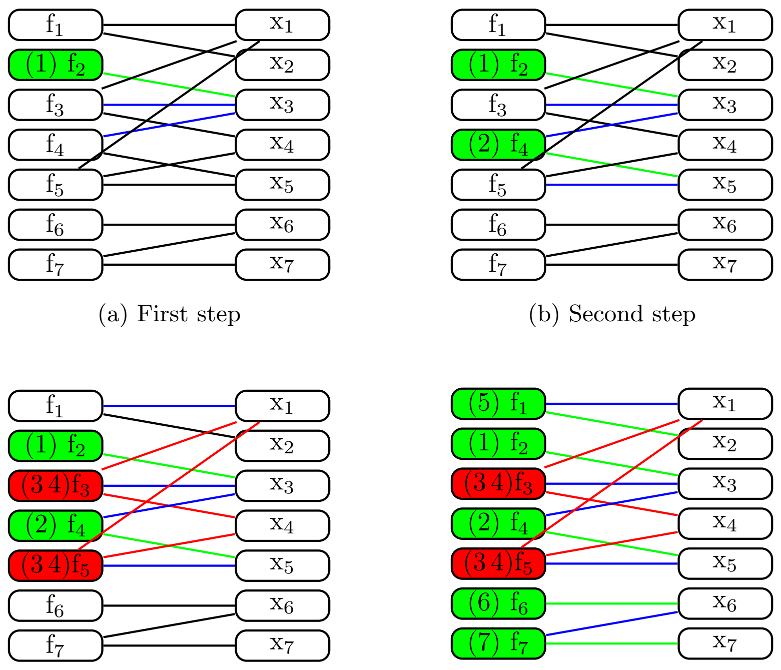
- If an equation has only one black line attached to it, color that edge green, follow it to the variable it connects to and color all remaining edges of that variable blue. Number the equation $i$ and increment $i$.
- If a variable has only one black line attached to it, color that edge green, follow it to the equation it connects to and color all remaining edges of that equation blue. Number the equation $j$ and decrement $j$.

If the algorithm terminates successfully it will produce a causalized system of blocks containing subsystems of inter-dependent equations, also called algebraic loops, representing strongly connected components (SCCs) in the graph. The associated incidence matrix is said to be of Block Lower Triangular (BLT) form. In the trivial case where each subsystem consists of a single equation the system will be completely causalized with an incidence matrix in the Lower Triangular (LT) form. There are methods of eliminating or at least reducing the size of SCCs loops in order to solve the whole system more efficiently. The sorting and matching process as well as the methods for reducing SCCs are discussed in detail in [13].

Figure 3.1 shows some of the iterations of applying this algorithm to the system of Equation 3.1. Figure 3.1c depicts the strongly connected subsystem of equations problem explained above shown with the edges marked red. The final mostly causalized system is shown in Figure 3.1d.

Now, if the equations from the final stage of the matching process are extracted and each variable is solved for in the corresponding matched equation the system shown in Equation 3.2 will be produced. The corresponding incidence matrix is shown in Table 3.2. Note that the matrix is in BLT form and not completely LT.

$$\begin{aligned}
x_3 &:= g_2(t)\\
x_5 &:= g_4(x_3, t)\\
g_3(x_1, x_3, x_4, t) &= 0\\
g_5(x_1, x_4, x_5, t) &= 0\\
x_2 &:= g_1(x_1, t)\\
x_6 &:= g_6(t)\\
x_7 &:= g_7(x_6, t)
\end{aligned} \tag{3.2}$$



(a) First step

(b) Second step

(c) Algebraic loop.

(d) Causalized graph

Figure 3.1: Matching of Equations

## 3.4.2 Strongly Connected Components

Five of the equations in the system are causalized to single assignment statements of the form:

$$x := f(\vec{v}, t) \tag{3.3}$$

where t represents time and v is the vector of variables involved in the computation of x. Equations $g_3$ and $g_5$, however, are strongly connected, i.e., mutually dependent on each other, thus forming a subsystem of equations which needs to be solved simultaneously for the variables $x_1$ and $x_4$. Although rather simple here, such blocks of equations, forming linear or nonlinear systems, can consist of tens or hundreds of equations.

Solving such simultaneous subsystem of equations is usually the most computationally expensive part of a simulation since these subsystems usually involve multiple assignments, complex linear algebra operations, some kind of iterative method for solution, etc. These complex blocks of equations often give rise to potential data parallelism within the block since most linear algebra operations can be parallelized. However this thesis work is

|  | $x_3$ | $x_5$ | $x_1$ | $x_4$ | $x_2$ | $x_6$ | $x_7$ |
|---|---|---|---|---|---|---|---|
| $g_2$ | ■ |  |  |  |  |  |  |
| $g_4$ | ▨ | ■ |  |  |  |  |  |
| $g_3$ | ▨ |  | ■ | ■ |  |  |  |
| $g_5$ |  | ▨ | ■ | ■ |  |  |  |
| $g_1$ |  |  | ▨ |  | ■ |  |  |
| $g_6$ |  |  |  |  |  | ■ |  |
| $g_7$ |  |  |  |  |  | ▨ | ■ |

Table 3.2: Incidence Matrix of causalized system

|  | $x_3$ | $x_5$ | $\{x_1,x_4\}$ | $x_2$ | $x_6$ | $x_7$ |
|---|---|---|---|---|---|---|
| $g_2$ | ■ |  |  |  |  |  |
| $g_4$ | ▨ | ■ |  |  |  |  |
| $g_{35}$ | ▨ | ▨ | ■ |  |  |  |
| $g_1$ |  |  | ▨ | ■ |  |  |
| $g_6$ |  |  |  |  | ■ |  |
| $g_7$ |  |  |  |  | ▨ | ■ |

Table 3.3: Incidence Matrix in Lower Triangular form

not concerned with this potential and treats such complex blocks as atomic units of computation that need to be executed non-preemptively as a whole by a single processing unit. The system of equations formed by such blocks of equations can be represented as

$$\vec{x} := f(\vec{v}, t) \tag{3.4}$$

where $\boldsymbol{x}$ is now the vector of variables being updated by the system of equations. In the above example the equations $g_3$ and $g_5$ are treated as one single vector equation of the form:

$$\{x_1, x_4\} := g_{35}(x_3, x_5, t) \tag{3.5}$$

resulting in the incidence matrix of Lower Triangular form shown in Table 3.3.

### 3.4.3 Decoupled Systems and TLM for Coarse-grained Parallelization

From the Incidence matrix in Table 3.3 it can be observed that the system contains two sets of connected components. The sets $\{g_6,g_7\}$ and $\{g_2,g_4,g_{35},g_1\}$ form two completely independent subsystems or partitions. Having these kinds of equation systems with multiple decoupled system gives a potential for parallelism in each time step of a simulation. Since the two sets have no equations with dependencies outside their set they can potentially be computed simultaneously for each time step. This provides the opportunity to utilize a coarse-grained automatic parallelization mechanism to improve the simulation performance of models with such multiple systems.

If a system has many such decoupled subsystems of equations then a load balancing method needs to be available in order to make sure that the parallelized simulation can distribute work approximately equally to each processing unit involved in computation. However before any load balancing is performed there need to be some kind of cost/load estimation mechanism which can be either static or dynamic.

The simplest cost estimation mechanism would be a static cost estimation based on the size of the subsystem, i.e., the size of a partition is defined to be its cost. The load balancing can be done by trying to move and merge these partitions with each other to end up with sets of partitions with equal size or cost. This, obviously, is not the most effective method since not all equations have the same computational cost. Different expressions in different equations result in different computation load. Things are further complicated by the presence of linear and non-linear systems, loops, function calls, etc. More effective static and dynamic cost estimation methods are discussed later in Section 3.5.4.

Once the partitions have been balanced then code for parallel execution can be generated in a rather straight-forward fashion. Generate each system separately, use as many processing units as the number of partitions if there are no restrictions on the number of units. If the number of processing units is specified or restricted then merge the partitions further to match the number of processing units. Then synchronize all the processing units at the end of each time step. Of course things are not so straight-forward for an actual implementation and a number of factors should be taken into account. The most important one of which is runtime thread safety, i.e, there should be mechanisms ensuring that shared data structures are manipulated in a manner that guarantees safe and correct execution.

The coarse-grained parallelization approach discussed above should work sufficiently well if the system in question contains many partitions and has a rather uniform cost for each individual equation. Unfortunately, in practice, most physical models are rather highly connected. This is not surprising since modelers and engineers are usually interested in modeling a specific aspect of a system that contains variables which are dependent on each other and ignore aspects of the system that does not affect or is not affected by what they want to observe.

The decoupling of systems can be further improved by a modeling technique called Transmission Line Modeling (TLM). We have previously investigated this approach [52] and implemented a parallelization that is based on balancing these completely independent systems and executing them in parallel. Although we have found quite satisfactory results with this approach it has limited genera usability for two reasons. First many application models are heavily connected. Modelers and engineers are usually interested in modeling a specific aspect of a system which contains variables that are dependent on each other and ignore aspects of the system that does not affect or is not affected by what they want to observe. This limits the amount of decoupling we can utilize.

The second reason is that improving the system decoupling with the TLM approach requires modifying the original Modelica model which is also something most modelers and engineers are not excited about. They like their models to resemble the natural system they are modeling and TLM may add some obscurity to the model. However, other models who are using TLM argue that TLM improves the model structure by making existing physical delays or decoupling more explicit. In addition there are quite large number of Modelica models and libraries around already in use which can benefit with some parallelization without the need for them to be rewritten or revised.

For these reasons we have decided to improve the previous implementation to analyze not just connected components but the whole system and extract more parallelism. To this end we have designed and implemented a task graph based approach to better represent the system and enable a more convenient analysis. This approach uses graph structures and algorithms to represent the complete equation system as a task graph and processes it further to extract parallelize and improve performance overall.

An alternative approach of extracting parallelism by further analyzing and manipulating the BLT incidence matrix is proposed by Casella [12].

### 3.4.4 From Equation Systems to Task Graphs

The parallelization process starts by converting the incidence matrix into an adjacency list representing a directed acyclic graph (DAG). A directed acyclic graph DAG, where each node represents an equation block and each directed edge from block $i$ to block $j$ represents a dependency of block $j$ to the variable defined (assigned to) in block $i$, can be built using the information provided in the incidence matrix. Assuming the incidence matrix (IM) has N entries the graph can be constructed as shown in Listing 3.1. The resulting DAG after applying this algorithm to the lower triangular incidence matrix shown in Table 3.3 is shown in Figure 3.2.

From now on we will refer to equation blocks (both simple and complex) as tasks and treat both kinds of blocks as atomic tasks. This, naturally, gives rise to a task system with tasks of variable length (computation times) and width (the number of variables involved). Clearly some sort of scheduling is needed in order to balance this task system and execute it efficiently.

```
v0 = add_node()
for i in 1:N-1 loop
  vi = add_node()
  for j in i-1:0 loop
    if IM[i,j] != 0
      add_edge(vj,vi)
    end if
  end for
end for
```

Listing 3.1: Incidence Matrix to Adjacency List

### 3.4.5 Data Dependency

Each edge in the task graph from one task/equation to another represents some kind of data dependency between these tasks. A directed edge from equation A to equation B means that equation B will have to be evaluated after equation A has been fully evaluated. These data dependency edges are created by analyzing which variables are used and/or updated in each equation and finding the intersections between the variable sets of the two equations.

Before we can discuss the data dependencies to equation systems resulting from a mathematical modeling environment like Modelica we need to explain the types of data dependencies that can appear in task systems.

Generally there are three types of data dependencies between different tasks: *flow dependency*, *anti dependency* and *output dependency*. Theses are described below. For all explanations assume we have three tasks $A$, $B$ and $C$ which appear respectively in sequence in the original sequential flow.

#### Flow Dependency

Flow dependency or true data dependency is the dependency created as a result of task $B$ using data produced as a result of task $A$. This is a *read after write* dependency and has to be strictly obeyed, i.e, the corresponding task need to be executed in the original order. Consider

$$\begin{aligned} A: &\quad a = b + c \\ B: &\quad d = sin(a) \end{aligned}$$

Here the second statement uses the value $a$ which is produced by the first statement to compute the value of d.

#### Anti Dependency

Anti dependency is the dependency created as a result of task B producing a new value of data that was used in A. This is a *write after read* dependency.

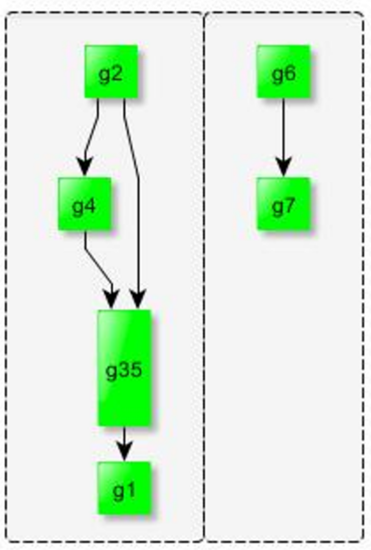


Figure 3.2: Resulting task graph of equation system

Consider

$$\begin{aligned} A: &\quad a = b + c \\ B: &\quad b = sin(d) \\ C: &\quad e = b - c \end{aligned}$$

Here the second statement produces a new value for the variable $b$ which has already been used by the first statement to compute the value of $a$. If some parallelization mechanism manages to execute task $B$ before $A$ then the result of $A$ will be wrong. Furthermore, if they were to execute simultaneously on different processing units the result will be undefined since we are trying to read and write to the same variable $b$ at the same time.

It is possible to resolve these dependencies by *renaming.* For example we can introduce a new variable $t$ to hold the value of $b$ as output of $B$ and rename all subsequent uses of $b$ to $t$ as shown below.

$$\begin{aligned} A: &\quad a = b + c \\ B: &\quad t = sin(d) \\ C: &\quad e = t - c \end{aligned}$$

### Output Dependency

An output dependency is the dependency created as a result of task some task $B$ producing a new value of data that was also produced by some other task $A$. This is a *write after write* dependency. Consider

$$\begin{aligned} A: &\quad a = b + c \\ B: &\quad a = sin(d) \\ C: &\quad e = a - c \end{aligned}$$

Here the second statement produces a value for the variable $a$. However, the first statement also produces a value for the same variable. Unlike *anti dependency* modifying the execution order these tasks will not invalidate a correct result. However, if they were to run simultaneously with different processing units then there might be undefined behavior since the two units will be trying to write the same variable $b$ from at the same time.

A *renaming* operation can also resolve these dependencies completely. For example we can introduce a new variable $t$ to hold the value of $a$ as output of $B$ and rename all subsequent uses of $a$ to $t$ resulting in the same resolved structure shown for *anti dependency.*

### Data Dependencies in Mathematical Equation Systems

Now that the different kinds of dependencies found in task systems are explained, the question is how would systems of equations resulting from mathematical modeling environments would appear in regard to these dependencies. Task graphs created from theses equation systems have some a few characteristics that are interesting for dependency analysis.

Modeling environments and languages have certain restrictions on the structure of the equation system that is to be solved. This restrictions can simplify the job dependency analysis for parallelization by eliminating or preventing certain types dependencies.

The causalized equation systems generated from Modelica models will have the same number of equations as variables. In other words the incidence matrices for these systems need to be square matrices. The sorting and matching processes explained in 3.4.1 result in systems with some useful properties.

- Causalization/Matching: *Every variable in the equation system will be assigned to only once.* That means that the variable will be updated by only one equation (left hand side variable updated in one and only one equation). This in turn insures that we will have *no output dependencies* in the generated system. This also applies for SCC blocks of the from shown in Equation 3.5.

- Sorting: *Every variable is assigned a value before it is used.* That is it appears on the left hand side of an equation before it is used by the right hand side in any equation. Since, by the previous rule, a variable can only be assigned by only one equation there will be no cases where a variable can be written to after it is read. This in turn ensures that there will be *no anti dependencies* in the system.

These two properties can also be observed by realizing that the incidence matrix for the system is in lower triangular form where the diagonal elements specify the equation which updates the corresponding variable (black cells in Table 3.3).

The only dependency types that need to be considered are *true dependencies*. These are the kinds of dependencies that define the structure of task systems created corresponding to a mathematical equation systems.

### 3.4.6 Data Memory

Task systems represented by directed acyclic graphs usually associate some attributes to the edges of the graph in addition to just using edges to represent dependencies. The most common attribute, for edges of task systems intended for parallel execution, is communication cost. This cost is the time used for sending data from the source node of the edge to the destination node.

In distributed memory computer architectures, this communication cost is an important attribute that can affect the scheduling process as well as execution performance of the parallelized system. Therefore considerations need to made to make sure that data is communicated only when necessary. For example it might be better to accumulate data to be sent from one node (possibly originating from multiple tasks) and forward this data to another node at one time in order to reduce the overhead involved with many send operations.

In this work however we are solely interested in shared-memory multicore architectures. This essentially renders the need for communication non-existent. Data is shared by all processing units and is visible to all throughout the simulation process. Therefor edges in this task system representation have no cost attributes needed. In fact edges currently have no extra attributes apart from representing dependencies.

In the future it might be useful to improve the implementation to also handle distributed memory computer architectures as well. In that case the issues mentioned above for those systems need to be taken in to consideration.

## 3.5 The Task Systems Library

The task systems library, developed as part of this thesis work, is a generic task representation, clustering, profiling and execution library written in C++. It uses C++ templates heavily and is built on top of the Boost Graph and Intel Thread Building Blocks Libraries. Boost Graph is used to represent the underlying graph primitives for the task systems and TBB for the parallelization primitives. The library can be used for the whole task representation and parallelization process including clustering, profiling and execution. However it is also possible to only use the library for one or more of these purposes. For example it can be used to perform a specific set of clustering algorithms on the task system. The resulting system can be used for other purposes. The idea is to provide a framework that is easy to extend, for example by adding a new custom clustering algorithm or a new scheduling algorithm while the rest of the system representation stays intact. Currently the library supports only shared memory systems.

### 3.5.1 Task Parallelism and Scheduling

The task parallelization approach and the accompanying Task System library presented here is originally implemented for the OpenModelica simulation compilation and runtime system. However the theory can be applied to any system that implements an abstraction of tasks and dependencies. This can, for example, be a train scheduling problem with connections, executable code generated from compilers, a system of equations with dependencies, etc. The library is a generic C++ template library. It is designed to be as generic as possible and to provide a lot of flexibility in how clustering algorithms and schedulers are implemented and used.

This thesis work presents automatic parallelization of systems of equations in the context of code generation and execution of the OpenModelica compiler for models provided in the Modelica language.

### 3.5.2 Task Systems

The library uses an adjacency list to represent a directed acyclic graph (DAG) that models a given task system. The task system can be represented by the tuple

$$G = (\vec{V}, \vec{E}, c) \tag{3.6}$$

where **V** is the set of vertices, **E** is the set of directed edges and **c** is the execution cost of each vertex. Each vertex in the graph corresponds to one task and each directed edge represents a data dependency between the source vertex/task and the destination vertex/task.

An abstract root node/task is added to each task graph. All tasks in the system which would otherwise be root nodes, i.e., have no parent task, are added as children of this root node. This abstract node is used to conveniently manipulate the task system. The *level* of a node is defined as the longest path from the root node to the node. Since the task system is represented as a DAG with non-weighted directed edges it is possible find the level of each node with a breadth first visit.

The abstract root node is the only level 0 node and all tasks that do not depend on another task in the system become level 1 nodes with the root node as parent.

#### 3.5.2.1 Tasks and Clusters

A user defined task can be represented as a C++ class. This class must inherit from the abstract class Task provided by the library. The abstract base class provides the necessary information for the clustering and scheduling algorithms. For example a modified version of an equation task for the OpenModelica compiler is shown in Listing 3.2

```
struct Equation
: public Task
{
  Equation();
  long index;
  std::set<long> depends;
  std::set<long> updates;
  virtual bool depends_on(const Task&) const;
  virtual void execute();
};
```

Listing 3.2: A custom class representing an equation task

A node in the internal task graph representation is not a task but rather a cluster of tasks. Each cluster initially consists of a single task i.e. when the library is asked to create a new task it creates a new cluster with a single task. Later the different clustering algorithms rewrite the system graph by moving tasks from one cluster to another. If a cluster becomes empty due to one of these task movements it gets detached from the system and destroyed. Clusters are responsible for the profiling and execution of their tasks. Tasks in a single cluster are always executed sequentially and in the order they are added to the cluster.

#### 3.5.2.2 Task System Construction

The construction of a task system starts by creation and addition of tasks. The library can read the necessary dependency information from an XML file and create the system automatically. Tasks can also be added manually and one by one directly into the task system. This method can be used if it is not possible to generate an XML representation of the system.

Creation of edges or dependency representations can be handled in two ways as well. The first method involves the programmer/compiler directly creating edges between vertexes. Another option is to let the library handle the analysis to some extent. Whenever a task is added to the system the library traverses each existing task and determines if there is a dependency between the existing task and the new task.

If creation of edges or dependencies is to be handled by the library as explained above then the new task class should override and implement the method `depends_on(const Task)`. This function is used to determine if a given task depends on another task and is used by the system to create edges between new tasks and existing tasks. It is completely up to the programmer/compiler to specify how dependencies are decided between the tasks. The library will just call this method for each task instance and creates a dependency if it gets a `true` return value. For example in the OpenModelica equation task class shown in Listing 3.2 this function will check the if any of the variable ids in the depends set of the current task exist in the set updates of the other task passed as a parameter to it as shown in Listing 3.3. Note that there is no need to check for output or anti dependencies as explained in Section 3.4.5.

```
bool Equation::depends_on(const Task& other_b) const {
    const Equation& other = static_cast<const Equation&>(other_b);
    bool found_dep = false;

    // True dependency
    found_dep =
         utility::has_intersection(this->depends.begin(),this->depends.end(),
                             other.updates.begin(), other.updates.end());
    // // Anti-dependency
    // if(!found_dep)
        // found_dep =
             utility::has_intersection(this->updates.begin(),this->updates.end(),
                             // other.depends.begin(),
                                  other.depends.end());
    // // output-dependency
    // if(!found_dep)
        // found_dep =
             utility::has_intersection(this->updates.begin(),this->updates.end(),
                             // other.updates.begin(),
                                  other.updates.end());
    return found_dep;
}
```

Listing 3.3: Depdency Detection for OpenModelica Equation Task

If the execution of tasks is also to be handled with the system then the new derived task class should also override and implement the method `execute()` which is used by the executors of schedulers to launch each task.

## 3.5.3 Clustering Algorithms

The library also provides a few methods for clustering the system graph as well as mechanisms for writing customized clustering algorithms. These algorithms traverse the task system and move tasks from one cluster to another depending on a specific criteria. The available algorithms can generally be divided in two categories as: cost-based and non-cost-based.

```
for each node N in breadth_first_visit()
  if number_of_parents(N) > 1 then
    FP = get_next_parent(N)
    FPL = get_level(FP)
    while P in get_next_parent(N)
      if get_level(P) == FPL then
        merge_nodes(FP,P)
      end if
    end while
  end if
end for
```

Listing 3.4: MLP clustering

#### 3.5.3.1 Merge Single Parent (MSP)

The simplest clustering available is the merge single parent (MSP) rule. This clustering algorithm traverses the system graph and merges all tasks that have only one parent task to their parents. There are a number of advantages with this simple rule. The first advantage is that it eliminates the need for treating cheap tasks, i.e., tasks with low execution time, separately by the schedulers thereby reducing overheads for schedulers. Furthermore, since a dependency (parenthood/childhood) means sharing of some data between the two tasks, it improves temporal locality (cache for CPUs and local/shared memory spaces for GPUs) and makes sure data is used as soon as it is available for the next task. If a cheap child task is left unmerged then it might end up further away from the parent task in terms of execution order either due to application of other clustering rules or scheduler task selections. This will result in a poor temporal locality for data without any gain in potential parallelism.

#### 3.5.3.2 Merge Level Parents (MLP)

The second available algorithm is called Merge Level Parents (MLP). This clustering rule utilizes the level of a node and parenthood relationships between nodes to apply merging rules. It starts by finding the level of each node in the task graph. Then nodes are traversed in a breadth first manner starting from the third level (level 0 contains only the root node and nodes at level 1 have no parent apart from the root node). Then the merging rule is applied as shown in Listing 4 2.

The rule visits each non root node or task and merges its parents. If the node has only one parent then nothing is done). However parents are not merged if they are different level tasks. This restriction is enforced in order to avoid creating paths that will potentially create cycles in the graph. For example consider the system shown in Figure 3.3. If the MLP rule is applied to node 5 of this task system without the level restriction on parent node merging the system will end up with a cycle between the merged nodes 1,3,4 and 2. Although it is possible to check if merging of nodes would introduce a cycle prior to performing a merge, we have opted not to do this for performance reasons. Path finding is an expensive operation and performing it for every single merge can severely impact the efficiency of the whole algorithm.

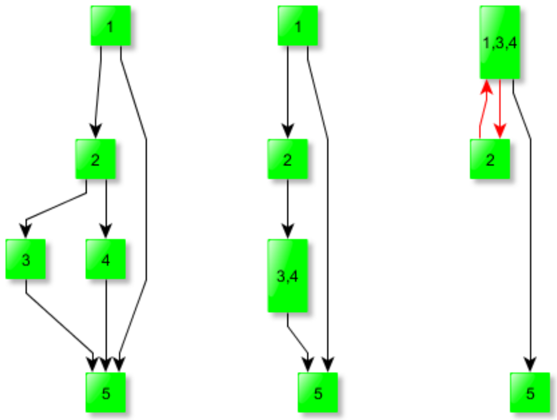


Figure 3.3: Example Cycles in MLP clustering

The two algorithms explained so far (MSP and MLP) are cost oblivious. They do not consider the cost of the child task, the parent task, or the merged task while applying the merging rules. They are useful for improving temporal locality so that tasks that operate on the same portion of data are executed as close in time as possible.

For task systems composed of tasks with relatively uniform costs these algorithm are convenient since they are fast and can generate balanced merged tasks. However for systems with tasks of varying cost they can end up with unbalanced branches that can affect parallelization severely for some scheduling algorithms, for example the level scheduler described in Section 3.5.5.1.

It should be noted that it is possible and rather easy to extend these algorithms to consider costs or to provide cost aware alternatives of them. The library already provides all the cost information needed as well as methods for cost based analysis e.g. selecting and sorting of tasks according to cost. However for now we have implemented other cost based algorithms.

The cost based algorithms traverse the task system and perform clustering of tasks until a given cost target is achieved. The target cutoff cost for a specific algorithm should be provided by the programmer/compiler. The cutoff costs for these algorithms should be selected carefully so that unnecessary clustering is avoided since this will limit the amount of parallelization available for the schedulers. Selecting too high cutoff cost means that more tasks will be merged into a single cluster. As mentioned before, tasks in a single cluster are always executed sequentially and in order. Selecting a high cutoff cost can result in forcing theses algorithms to merge two tasks that could have run in parallel. This will limit the amount of parallelization later available for the schedulers. On the other hand selecting a too small cutoff cost will prevent the algorithms form merging tiny tasks together. Having too many tiny tasks in the system, in turn, incurs extra overhead for the schedulers. The extra potential parallelization obtained by keeping these small tasks separate is not worth the overhead involved in keeping track of, scheduling and thread management of launching them individually. Therefore it is better to cluster them together.

For these reasons the cutoff cost should be selected cautiously for each of these algorithms. However, there is no universally convenient cost to be used for each algorithm since differences in architecture and intended use govern the decisions to be made. For example a task can be considered too small on a fast architecture with a high thread context switching overhead due to the fact that the architecture incurs more overhead to schedule the task separately than to merge it with another high cost task. On the other hand it might be better to keep it separate on a slow architecture but with very efficient thread management (e.g. GPU systems).

As mentioned earlier, the library can also handle profiling of the system for estimating costs (see Section 3.5.4 for detail). In this case, the effectiveness of these cost based algorithms is also very dependent on the clock resolution of the architecture.

Before describing the available cost based clustering algorithms we need to define the bin packing problem and related algorithms in the context of our task systems and cost model. For cost analysis of our task systems we define the upper bounded bin packing problem as

- given a set of nodes with costs $\boldsymbol{C} = \{\mathrm{c}_1,\mathrm{c}_2 \ldots,\mathrm{c}_n\}$ where

  $$c_i \leq \boldsymbol{cc}, \quad \forall c_i \in \boldsymbol{C}$$

  partition $\boldsymbol{C}$ to a number of mutually disjoint subsets $\boldsymbol{B} = \{\mathrm{B}_1,\mathrm{B}_2 \ldots,\mathrm{B}_n\}$ such that

  $$\sum\nolimits_{x \in B_i} x \leq \boldsymbol{cc}, \quad \forall B_i \in \boldsymbol{B} \qquad (3.7)$$

  where $\boldsymbol{cc}$ is the cutoff cost.

And the lower bound bin packing problem can be defined as

- given a set of nodes with costs $\boldsymbol{C} = \{\mathrm{c}_1,\mathrm{c}_2 \ldots,\mathrm{c}_n\}$ partition $\boldsymbol{C}$ to a number of mutually disjoint subsets $\boldsymbol{B} = \{\mathrm{B}_1,\mathrm{B}_2 \ldots,\mathrm{B}_n\}$ such that

  $$\sum\nolimits_{x \in B_i} x \geq \boldsymbol{cc}, \quad \forall B_i \in \boldsymbol{B} \qquad (3.8)$$

  where $\boldsymbol{cc}$ is the cutoff cost.

```
Sorted_Nodes = sort_nodes_descending(Given_Nodes)
if cc < get_cost_first_Node(Sorted_Nodes) then
  cc = get_cost_first_Node(Sorted_Nodes)
end if
while node N in get_next_node(Sorted_Nodes) loop
  gap = cc - get_cost(N)
  if gap > tolerance then
    while node M in visit_next_node(Sorted_Nodes) loop
      if get_cost(M) < gap then
        gap = gap - get_cost(M)
        merge_nodes(N,M)
      end if
    end while
  end if
end while
```

Listing 3.5: Upper bound FFD

We have implemented a modified versions of the First Fit Decreasing (FFD) packing [16] algorithm for problems of upper bound bin packing as well as lower bound packing problems.

Given a set of nodes both algorithms start by sorting nodes in descending order according to cost. Once sorted the upper bound FFD starts by selecting the first node as the first bucket. Notice that for upper bound packing the provided cutoff cost must be at least as big as the largest cost in the set. If the given cutoff cost is less than the cost of the largest task, then it is raised to this highest task cost. Then the algorithm iterates starting from the next node until it finds a node with a cost that can fit in to the first bin with in some pre-specified acceptable gap. If such a node is found it is added to the bin and removed from the set. Iteration continues until either the bin is full within the accepted tolerance or there are no more nodes to visit. Then the next available node is selected as the second bin (this is the biggest node in set now) and the process is repeated. This is performed until there are no more nodes left in the original set. Simplified pseudo-code for upper bound FFD is shown in Listing 3.5.

The lower bound FFD partitioning also starts by selecting the first node as the first bin. If the cost of the node is already bigger than the cutoff cost **cc** then the node is accepted as a bin and the next node is selected as the new bin. If the cost of the first node is less than **cc** then the algorithm iterates over the set of nodes in reverse order (i.e starting from the last node which is the smallest to the node next to itself). The first visited node (the last node) is added to the bin. If the merged cost is higher than **cc** then the next biggest node is selected as a new bin and iteration starts again. If not the next smallest node is added to the current bin and so on until the cost of the bin exceeds **cc**. Once the bin is full (cost exceeded) the next biggest

```
Sorted_Nodes = sort_nodes_descending(Given_Nodes)
while node N in get_next_node(Sorted_Nodes) loop
  gap = cc - get_cost(N)
  while gap > 0 loop
    M = get_last_node(Sorted_Nodes)
    gap = gap - get_cost(M)
    merge_nodes(N,M)
  end while
end while
```

Listing 3.6: Lower bound FFD

node and selected and the whole iteration is started. Simplified pseudo-code for lower bound FFD is shown in Listing 3.6.

#### 3.5.3.3 Merge Children Recursive (MCR)

The first cost based algorithm available is the merge children recursive (MCR) rule. This algorithm is somehow a mix of both MSP and MLP rules with some modification and cost considerations. The algorithm traverses all nodes of the graph in a depth first manner. For each node it collects all children of the node which have single parent (i.e. the current node). This subset of children nodes is then partitioned in to bins of minimum cost **cc** by applying the lower bound FFD.

Merging all single-parent children has improves temporal locality since all these tasks use portions of data which is updated by the parent task. If these tasks are merged together then it can be guaranteed that they will not be moved in to other clusters (merged with other nodes with different parents) later by additional clustering algorithms. Considering the cost of the tasks helps to achieve a trade-off between parallelizability and overhead involved in dealing with many small tasks. By removing tasks below the cutoff cost in the task system we make sure that tasks with higher overhead than execution time will not keep schedulers busy. In addition the lower bound FFD algorithm gives optimally balanced clusters that can be launched in parallel as soon as the parent task has finished execution.

#### 3.5.3.4 Merge Level for Cost (MLC)

Another available cost based algorithm is the merge level for cost (MLC) rule. This rule is implemented primarily to be used with the Level Scheduler presented in Section 3.5.5.1. However it can be used together with other schedulers which requires the number of processing units (cores, machines...) that can execute tasks to be known beforehand.

Given the number of bins **N**, the algorithm tries to collects all nodes at the same level and tries to pack them in to **N** bins while keeping the cumulative cost of each bin as close as possible to each other. This is a k-way spectral partitioning problem where we are interested in partitioning a given set S in to k subsets

$$B^k$$

so as to minimize some predefined objective function

$$F(B^k)$$

For our task system we can define the problem as

- given a set of nodes with costs $\boldsymbol{C} = \{c_1, c_2 \ldots, c_n\}$ partition $\boldsymbol{C}$ in to $k$ mutually disjoint subsets of costs $\boldsymbol{C_B} = \{C_{B1}, C_{B2} \ldots, C_{Bk}\}$ such that

  $$\begin{gathered} \sum_{i=1}^{k} |c_{opt} - C_{Bi}| \\ where \quad c_{opt} = \frac{1}{k} \sum_{i=1}^{k} c_i \end{gathered} \tag{3.9}$$

  is minimized.

That is given a set of nodes we have to partition them in to $k$ bins where the cost of each bin is as close as possible to the optimal parallel cost. The optimal parallel cost is where all the bins are of equal cost guaranteeing that, when run on $k$ identical processing units, all bins will finish at the same time. Our implementation uses approximations to convert this problem in to a upper bound bin packing problem followed by a sorted list scheduling . While the optimality of this approximation is open for interpretation it has turned out to be effective and fast for practical needs.

The algorithm works in two steps. The first step starts by calculating the optimal parallel cost $\mathbf{c}_{opt}$. That is the sum costs of all nodes divided by the number of bins. Now consider we have an upper bound bin packing problem where the cutoff cost **cc** is $\mathbf{c}_{opt}$. The upper bound FFD algorithm is applied to the sorted nodes as $\mathbf{c}_{opt}$ as the cutoff cost. Since tasks cannot be broken down to fit gaps and and upper cost limit is imposed on bin sizes the algorithm might not be able to fit all nodes into bins. These remaining nodes are handled in the next step of the algorithm. In this steps the algorithm picks the largest task from the set of remaining tasks and adds it to the bin with the smallest cost. This process is repeated until there are no more tasks remaining.

### 3.5.4 Profiling and Cost Estimation

There are two ways of handling cost estimation for tasks. The first method is static cost estimations. In a static cost estimation approach tasks are assigned costs when the task system is constructed. In other words these cost are decided by the compiler at compile time or by the user at task system construction stage. The second method of cost estimation is dynamic. It is dynamic in the sense that task costs are estimated by actually executing and profiling each task at least once. This means estimations are done at run-time. These two cost estimation methods are discussed here in detail.

#### 3.5.4.1 Static Cost Estimation

Static cost estimation relies on user provided cost values for the whole operation of clustering and scheduling. Users have to set costs of task either at task creation time or later but before applying any clustering rules.

Static cost estimation is suitable for handling tasks that are executed only once or very few times per program or whole system execution. For systems of this nature task costs have to be estimated manually during compile time since there will be no opportunity to actually measure and store execution times of the tasks at run time. This can be done by analyzing the internal representation of tasks (e.g., traversing abstract syntax trees or intermediate representations at compile time and estimating costs per expression).

Static cost estimation is also mandatory when the library is used just for offline scheduling. Since there is no execution of tasks in this case, all cost information should be provided by the user. Task costs can vary between executions due to multiple reasons. For example differences in architecture and current load on executing machines can significantly affect costs of tasks. In addition compile time analysis of abstract syntax or internal representations of code to estimate cost is not always possible. For example calls to external or library routines cannot be traversed to estimate cost at compile time since the source code is not available.

#### 3.5.4.2 Dynamic Cost Estimation

Most modeling and simulation languages and environments, in contrast to computation languages like C++ or Java, can involve different kids of iterations over a certain set of tasks to solve the given problem (e.g. DAE solvers, numerical iterations...). This provides the opportunity to measure and store execution costs of tasks at runtime and to use this information to perform more effective clustering and scheduling.

In dynamic cost estimation mode the implementation always assumes the first request to evaluate the whole task system as a profiling stage. More specifically the library will not perform scheduling before executing the task system once. In this first execution step all tasks are executed sequentially and in profiling mode. Execution times are recorded and stored in each task. Once the whole system has been executed the library will schedule the system. In all subsequent steps or calls for evaluation of the system, parallel execution is performed using the existing schedule unless rescheduling is requested explicitly. Periodic rescheduling might be useful since task costs might change significantly between evaluations of the task system due to state changes in the system.

To illustrate the dynamic cost estimation and execution process we can look at the simulation process of the OpenModelica Compiler (OMC). As mentioned earlier OMC takes an object oriented acausal Modelica model representation and generates a system of DAEs to be solved. This system of DAEs is then solved by OMC's runtime system. For example consider the system of ODEs only. Once initialization of the model system is completed, the ODE solvers of the runtime system evaluates the ODE system over multiple time steps. The first evaluation call of ODE systems is considered as a profiling step. Each equation or block of equations is evaluated sequentially and execution times recorded. As soon as the ODE system finishes one whole evaluation (i.e., one time step) the scheduling is performed. Every subsequent time step will involve parallel evaluations of each equation in the ODE system using this schedule.

Accurate cost measurements are vital for the effectiveness of clustering rules. Especially in systems containing relatively small and highly varying costs from task to task. Since all the cost based algorithms explained above utilize the cost times collected by the profiling stage, inaccurate or too crude timing results can reduce the effectiveness of the resulting clustering task system by forcing them to apply/not apply the optimal clustering. For this reason time measurements should have satisfactory resolutions to successfully measure even the smallest cost tasks in the system. On Windows systems the library uses the QueryPerformanceCounter API function. On POSIX systems the Boost chrono library is used. Although both of these systems should provide sufficient clock resolution for most task system purposes it should be noted that resolutions might vary from one architecture or configuration to another. As a result performance of parallelization can also vary respectively due to different scheduling outputs.

### 3.5.5 Schedulers

Currently two simple scheduler implementations are available in the library. The first scheduler is a level based scheduler. This is a lock step or wave front based scheduler. The second scheduler is built on top of Intel Thread Building Blocks' flow graph implementation .

Schedulers in the library are implemented as standalone C++ template classes. These classes are responsible for executing clusters in parallel and synchronizations in their own specific way. Actual clustering (if needed) is provided by the clustering classes presented in Section 3.5.3. These clustering classes are provided as template parameters for the schedulers and are applied in the order they are passed. For example the Level Scheduler class used mainly in the current OpenModelica parallelization implementation is shown in Listing 3.7.

#### 3.5.5.1 Level Scheduler

The level scheduler algorithm schedules tasks of the system one level at a time synchronizing between levels only. Node or tasks in the same level or wavefront are completely independent of each other and can run in parallel. Level schedulers are simple and incur very low overhead in managing tasks and synchronization since the only requirement is that we make sure all tasks of one level are finished before starting any task in the next level. This can be implemented as a single barrier between sets of tasks. No thread starts working on the next task before all threads have reached the barrier thereby guaranteeing that all tasks in the level have finished. This is provided by the core executor class `StepSync`.

The actual level scheduler class is a specialization of this core algorithm with specific clustering algorithms. Variations of this level scheduling approach can be created simply by specializing the `StepSync` executor with a different set of clustering algorithms in different orders.

Despite being a "simple" scheduler, a level scheduler can be a quite effective and fast scheduling algorithm when combined with appropriate clustering algorithms to make sure that merged tasks (clusters or bins) at the same level are as balanced as possible. For example the actual level scheduler class that is mainly used in the OMC parallelization is a `StepSync` executor with two clustering stages: the Merge Children Recursive rule applied first and then the Merge Level for Cost. This is the class shown in Listing 3.7.

```
template<typename TaskType>
struct LevelScheduler :
StepSync < TaskType
         ,MCR
         ,MLC
       > {};
```

Listing 3.7: Level Scheduler Class for OpenModelica

To illustrate the effect of applying specific clustering algorithms and their influence on the effectiveness of the level scheduling approach let us consider a Four Bit Binary Adder model (`Modelica.Electrical.Spice3Examples.Spice3BenchmarkFourBitBinaryAdder`) from the Modelica Standard Library (MSL) . The original task graph for the ODE system of this model is shown in Figure 3.4. The system contains 1122 nodes or tasks and 1360 edges. Scheduling all these tasks with the corresponding dependencies obeyed is clearly a very expensive task. Performance can further deteriorate if most of these tasks happen to incur more overhead than the gain from parallelization.

Applying MCR (Section 3.5.3.3) clustering algorithm to this task system reduces the number of individual tasks (now clusters with possibly multiple tasks) and the number of edges to 569 and 620 respectively.

After applying MLC clustering(Section 3.5.3.4) clustering algorithm with

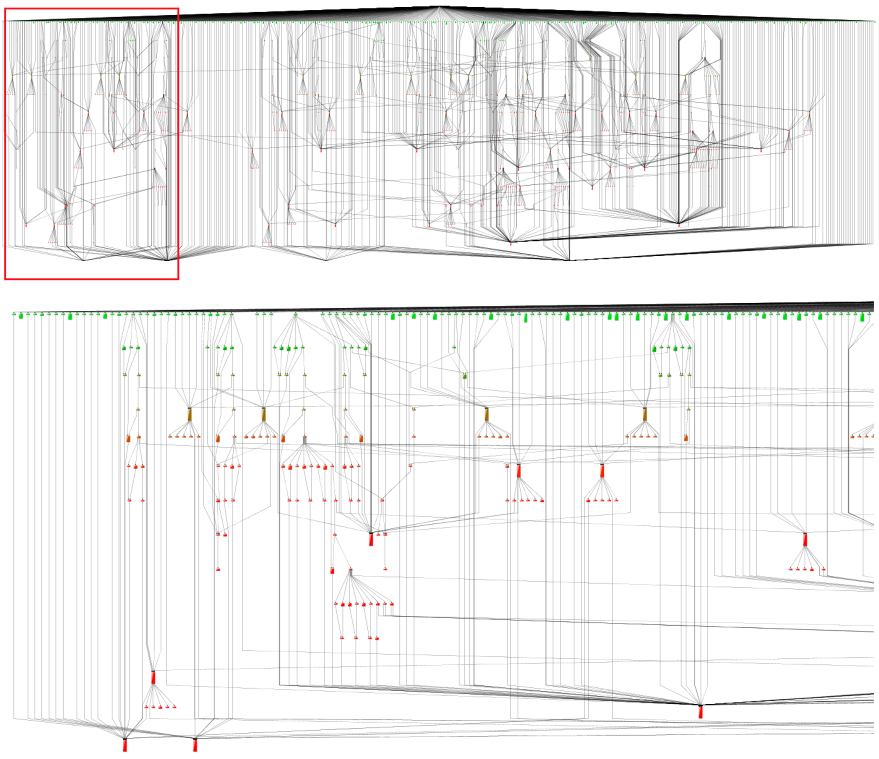
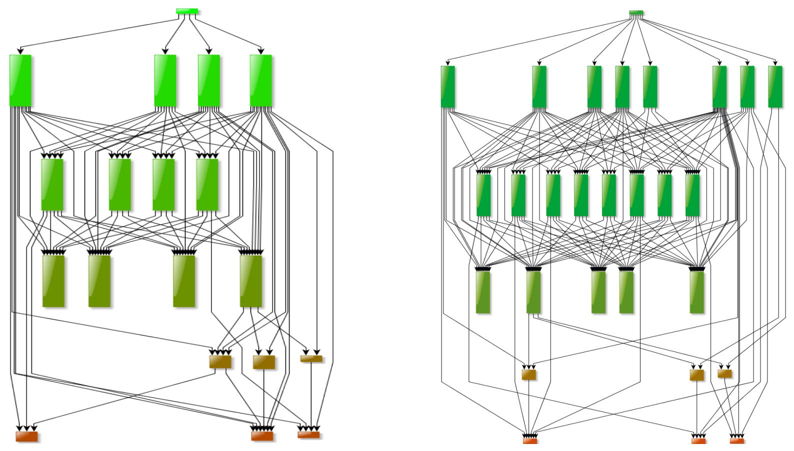


Figure 3.4: Original task system of Four Bit Binary Adder model

k=8 (i.e., targeted for 8 core execution with the level scheduler) to the resulting task graph we are left with a graph of 27 clusters and 121 edges. If instead we apply MLC with k = 4 (i.e., targeted for 4 core execution), the resulting task system ends up with a graph of 18 clusters and 72 edges. The resulting graphs for the task systems with application of 8-way and 4 way MLC are shown in Figure 3.5 respectively.

Note that many edges in the resulting task graphs of Figure 3.5 are shortcut edges. Given vertices $v_1$, $v_2$, and $v_3$ with directed edges e($v_1$,$v_2$) and e($v_1$,$v_1$), edge e(i,k) is a shortcut if there is a path from j to k, i.e, k is reachable from j.

Scheduling these simplified and balanced clusters is rather easy and straight-forward. Once the graph have been simplified as shown the `StepSync` executor traverses the final graph and collects the nodes based on levels. Then each level is executed one after the other and clusters at each level are launched in parallel.



(a) 4 way clustered (b) 8 way clustered

Figure 3.5: Task system of Four Bit Binary Adder model after MCR and MLC

#### 3.5.5.2 Intel Flow Graph Based Scheduler

This scheduler uses the Intel TBB's Flow Graph (TBBFL) [27] implementation as core executor. The Intel TBB Flow Graph implements its own a message based work stealing algorithm to dynamically execute tasks in the graph.

This scheduler class is implemented as a wrapping scheduler for TBBFL. The scheduler incorporates clustering and hides the details of TBB related primitives from the user. Similar to the Level Scheduler presented in section 3.5.5.1 this class is responsible for the profiling cost estimation and execution of tasks. It automatically creates the additional flow graph once profiled cost values are obtained and specified clustering rules are applied to the task system.

It should be noted here that it is possible to directly create a flow-graph representation from the original equation system. However there two main drawbacks to such an approach. The first drawback is that this will remove any flexibility we have with our Task System library. For example for the OpenModelica runtime system with a given task system switching between the Level Scheduler and Dynamic Scheduler is a matter of a single typedef change in the C++ code. Direct implementation with TBBFL would require a complete new implementation to be employed if any other scheduling algorithm is to be substituted later.

The second drawback is concerned with performance. Directly creating the flow-graph representation from the equation system eliminates the possibility of applying any clustering algorithms. Applying clustering rules based on dynamic cost estimations as previously presented above (3.5.4) considerably reduces the complexity of the original graph. This in turn reduces the number of individual tasks the flow graph has to deal with thereby reducing the overall overhead.

## 3.6 Performance Measurements

To evaluate the performance of the task scheduling implementation and the schedulers tests have been performed using models from the Modelica Standard Library. In the following two selected models with satisfactory performance improvements are presented.

### 3.6.1 Measurement Setup

All tests have been performed on a 64-bit Intel(R) Xeon(R) W3565 CPU with 4 cores at 3.2 GHz (3.46 GHz turbo) frequency. The machine is running Windows 7 professional Edition. Simulations have been performed from time 0 to 1 second with a step of 0.002 seconds. All simulations are done using the default OMC solver which right now is DASSL. Only ODE systems of models are currently parallelized.

The time results presented do not include model compilation time. Only simulation executable running times are measured. However, all parallelization related execution times are included. This includes all the extra overhead from task system creation, clustering and scheduling plus a sequential first step computation performed to collect cost information. Therefore the timing results here are what users should expect when running simulations normally (sequential or parallel).

For each model we have presented the estimated speedup with level scheduling and actual achieved speedup with level scheduling as well as the Intel flow graph based scheduler. The estimated speedup for the Level Scheduler is the ratio of the sequential cost to the ideal parallel cost. Sequential cost is obtained by summing the costs of all individual tasks in the system. The parallel cost is obtained by summing the costs of the largest tasks at each level in the clustered task graph.

- given a task system with tasks T of nodes $\{t_1, t_2 \ldots, t_n\}$ with costs $\boldsymbol{C} = \{c_1, c_2 \ldots, c_n\}$ and with the levels of each node calculated and given as a set of sets of nodes of same level as $\boldsymbol{L} = \{L_1, L_2 \ldots, L_m\}$ where $m$ is the critical path (maximum level)

$$\begin{aligned} C_{seq} &= \sum_{i=1}^{n} c_i \\ C_{par} &= \sum_{j=1}^{m} lc_j \quad where \\ lc_j &= max(c_i) \quad \forall c_i \in \boldsymbol{L}_j \end{aligned} \tag{3.10}$$

### 3.6.2 Results

The first test model presented is a fifth order low-pass-filter model (CauerLowPassSC) from the Electrical Analog library of MSL. Speedup curves for this model are shown in Figure 3.6. The second model presented is the Branching Dynamic Pipes model from the Fluid library of MSL demonstrating the use of distributed pipe models with dynamic energy, mass and momentum balances. The results for this model are shown in 3.7

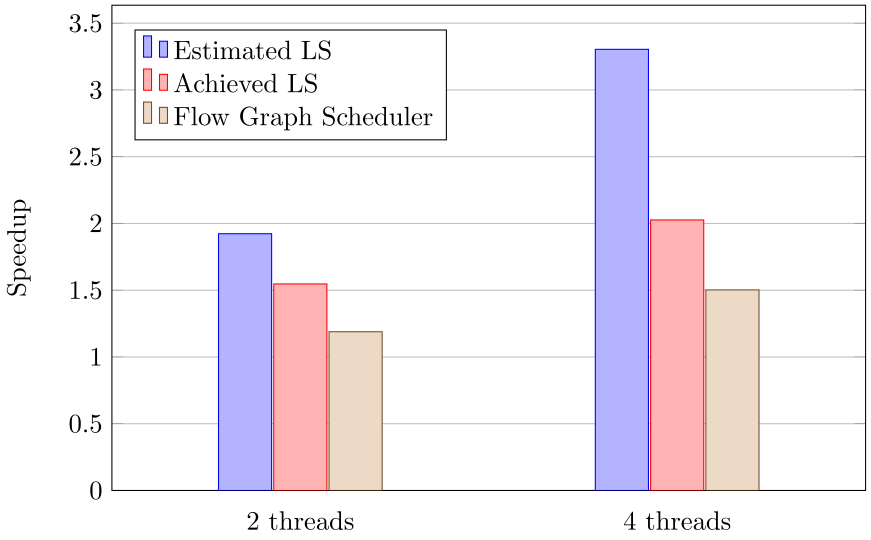


Figure 3.6: Speedup for CauerLowPassSC model

For the CauerLowPassSC model the Level Scheduler implementation outperforms the dynamic flow graph scheduler for both 2-threaded and 4-threaded executions. On the other hand for the BranchingDynamicPipes model the flow graph based scheduler outperforms the Level Scheduler on both runs. One reason for this behavior can be the different nature of equation system composition in the two models. The BranchingDynamicPipes model results in an ODE task system with 48 nonlinear systems while the CauerLowPassSC has no nonlinear systems at all. Nonlinear equation blocks are by far the most expensive parts of the simulation executable (i.e., they are large tasks). Having such large tasks in abundance gives the dynamic flow graph scheduler a higher parallelization to overhead ratio since threads spend most of their time working on these large tasks.

On both test cases we can observe that the level scheduler shows very promising estimated speedups. Although it is not practically possible to achieve this ideal speedup the implementation can be improved to achieve close to estimated speedups.

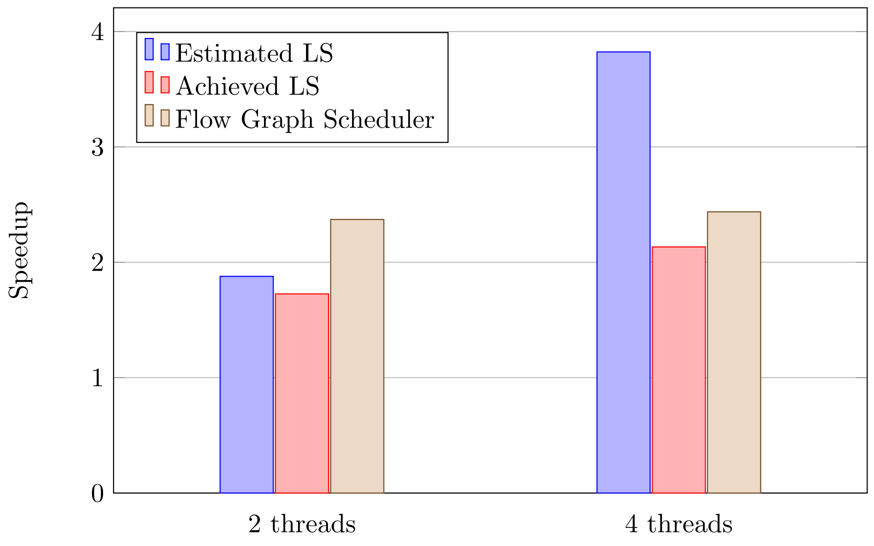


Figure 3.7: Speedup for BranchingDynamicPipes model

## 3.7 Conclusion

The current automatic parallelization implementation already shows significant performance improvements over sequential execution. This performance gain is expected to improve even further in the future with introduction of more powerful schedulers and clustering rules as well as improvements of the current ones.

We have presented a simple approach to analyze, extract, and represent dependencies in complex equation systems. We have also presented a task-graph based approach of extracting parallelism from these equation systems and utilizing it in a convenient way with the help of the Task Systems library.

The clustering algorithms that are available right now have already shown satisfactory clustering capabilities for generating efficient schedules. Combinations of these available algorithms can be used to manipulate the task graph to offer a more customized output that is fit for a selected scheduler approach as demonstrated by the scheduler implementations presented in Section 3.5.5.

Two scheduler implementations have been developed. These schedulers have demonstrated good speedup results on a few test models, typically 2 to 3 on a 4-core laptop. Different flavors of the existing schedulers can already be created by using different combinations and orders of the available clustering algorithms while keeping the same core execution routine. Depending on the target architecture and application area this can improve performance.

It is possible to extend the implementation by adding new clustering or scheduling algorithms. The necessary mechanisms to ease the introduction of new scheduling and clustering algorithms is available. This allows users to focus on their actual algorithms and spend less time and effort on implementing the routine support procedures.

## 3.8 Future Work

There is substantial room for further improvement. First of all, more clustering algorithms should be implemented in order to give users/compliers a wider range of options to use with the existing schedulers. Better clustering algorithms lead to even better data temporal locality which should improve performance by increasing cache usage efficiency in CPU systems. Additional scheduling algorithms should also be implemented. Even though the two available schedulers show some performance improvement, they are by no means suitable for all kinds of systems. This can be observed by seeing how the two schedulers already provide different performance behaviors for models from different application areas as presented in Section 3.6.2.

Perhaps the most important improvement needed is the introduction of rescheduling or adaptive scheduling support. The current implementation, as discussed earlier in Section 3.5.4, always uses the first execution of the task system as a profiling stage. For simulations this is normally the first time step. Of course it is preferable to perform profiling as early as possible since it is not possible to apply the cost-based clustering algorithms and scheduling without having the cost information available. No schedule available in turn means no parallelization can be done and the system has to be executed sequentially.

On the other hand the first execution of the system (or the first time step of simulation) is usually not the best representation of the execution behavior of the system. Simulation is dynamic by nature and the execution cost of complex tasks can vary considerably throughout the duration of simulation. Therefore it is vital to have adaptive rescheduling mechanisms to keep up with the changes in the system behavior. This can be realized by introducing periodic profiling of the system to detect variations in execution costs of tasks. If the execution cost behavior of the system has changed considerably then a new schedule should be generated using the current cost information. This new schedule can be used for subsequent executions or time as long as the system behavior remains consistent. The support for adaptive scheduling is already planned to be added to the implementation and will hopefully help reduce the gap between the estimated speedups and achieved speedups.

# Chapter 4

# Explicit Parallelization

## 4.1 Introduction

As mentioned earlier most previous work regarding parallel execution support in the OpenModelica compiler has been focused on automatic parallelization where the burden of finding and analyzing parallelism has been put on the compiler. In this work, however, this responsibility is left to the end user programmer. The compiler provides additional high level language constructs needed for explicitly stating parallelism in the algorithmic part of the modeling language. This among others includes parallel variables, parallel functions, kernel functions and parallel for loops indicated by the parfor keyword. There are also some target language specific constructs and functions (in this case for OpenCL). All these extensions are collectively called ParModelica Extensions. These will all be presented in this chapter.

## 4.2 Background

### 4.2.1 General Purpose Graphic Processing Unit (GPGPU) programming

A GPGPU is a general purpose Graphics Processing Units (GPUs) designed for use in data-parallel graphic as well as non-graphic computations. Traditionally the use of most GPUs has been limited to processing of only graphics data. However, in recent years it has become more common to use them also for processing of non-graphic scientific and engineering computations.

GPGPU programming is based on the concept of using the CPU and GPU as heterogeneous computing units. The CPU is used to execute serial parts of the computation and manage the GPU whereas the GPU is used as another highly parallel processing unit to perform parallel parts of the computation. Different frameworks of programming for GPUs are available now. OpenCL, CUDA, DirectX [44], OpenGL [36] and DirectCompute

[43] are some examples. The last three frameworks are more focused on the traditional use of using GPUs for processing of graphic data. However CUDA and OpenCL provide rather complete support for proper GPGPU programming. These two are used widely to implement non-graphic heavy computations.

### 4.2.2 OpenCL

#### 4.2.2.1 The OpenCL Architecture

OpenCL is the first open, royalty-free standard for cross-platform, parallel programming of modern processors found in personal computers, servers and hand-held/embedded devices. The OpenCL programming language is based on C99 with some extensions for parallel execution management. By using OpenCL it is possible to write parallel algorithms that can be easily ported between multiple devices with minimal or no changes to the source code. The framework is composed of the OpenCL; programming language, API, libraries and a runtime system to support software development. The OpenCL framework can be divided in to a hierarchy of models: Platform Model, Memory model, Execution model and Programming model. A brief description of these models is given in the following sections. However, for a complete understanding of the OpenCL framework it is recommended that the reader accesses [37].

#### 4.2.2.2 Platform Model

The OpenCL platform model is defined as a Host processor connected to one or more OpenCL devices. The OpenCL devices are divided into one or more Computing Units (CU) which in turn are divided into one or more Processing Elements (PE). The host is responsible for managing the executions on OpenCL devices. This management includes: identifying and initializing OpenCL devices, data copy operations and submitting parallel jobs to the OpenCL device.

#### 4.2.2.3 Execution Model

The execution of an OpenCL program consists of two parts. The Host program which executes on the host processor and the OpenCL program which executes on the OpenCL device. The host program manages the execution of the OpenCL program. An OpenCL program is a collection of kernels which execute as separate and independent programs. Kernels are executed simultaneously by all threads specified for the kernel execution. The number and mapping of threads to Computing Units of the OpenCL device is handled by the host program. Each thread executing an instance of a kernel is called a `work-item`. Each thread or work-item has unique id to help identify it. A work-item can have additional id fields depending on the arrangement specified by the host program. Work-items can be arranged into `work-groups`. Each work-group has a unique id. Work-items are assigned a unique local id within a work-group so that a single work-item can be uniquely identified by its global id or by a combination of its local id and work-group id. The work-items in a given work-group execute concurrently on the processing elements of a single compute unit.

A wide range of programming models can be mapped onto this execution model. OpenCL explicitly supports two of these models; the data parallel programming model and the task parallel programming model.

#### 4.2.2.4 Memory Model

The OpenCL memory space is divided into four parts:

- Global Memory: This memory region permits read/write access to all work-items in all work-groups. Work-items can read from or write to any element of a memory object. Reads and writes to global memory may be cached depending on the capabilities of the device.
- Constant Memory: A region of global memory that remains constant during the execution of a kernel. The host allocates and initializes memory objects placed into constant memory.
- Local Memory: A memory region local to a work-group. This memory region can be used to allocate variables that are shared by all work-items in that work-group. The local memory space maybe implemented as dedicated regions of memory on the OpenCL device. Alternatively, it may be mapped onto sections of the global memory.
- Private Memory: A region of memory private to a work-item. Variables defined in one work-item's private memory are not visible to another work-item.

This division of memory spaces is shown in Figure 4.1.[1]. The access and allocation rights of the host and kernels to these memory spaces are shown in Table 4.1.

#### 4.2.2.5 Programming Model

The OpenCL execution model supports data parallel and task parallel programming models, as well as supporting hybrids of these two models. The primary programming model driving the design of OpenCL is data parallel. The data parallel programming model defines a computation in terms of a sequence of instructions applied to multiple elements of a memory object. In a strictly data parallel model, there is a one-to-one mapping between the work-item and the element in a memory object over which a kernel can be executed in parallel. OpenCL implements a relaxed version of the

[1]Picture taken from [37]

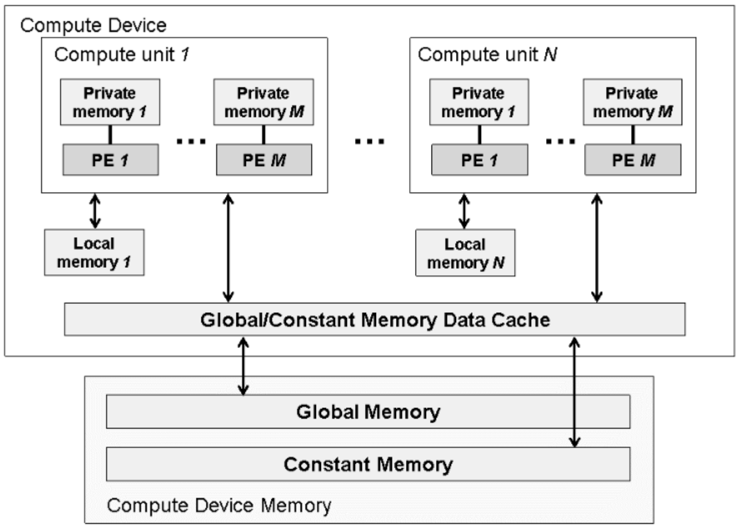


Figure 4.1: OpenCL Memory Model.

| | Global | Constant | Local | Private |
|---|---|---|---|---|
| Host | Dynamic allocation, Read/Write access | Dynamic allocation, Read/Write access | Dynamic allocation, No access | No allocation, No access |
| Kernel | No allocation, Read/Write access | Static allocation, Read-only access | Static allocation, Read/Write access | Static allocation, Read/Write access |

Table 4.1: OpenCL Allocation and Memory Access Capabilities

data parallel programming model where a strict one-to-one mapping is not a requirement.

OpenCL provides a hierarchical data parallel programming model. There are two ways to specify the hierarchical subdivision. In the explicit model a programmer defines the total number of work-items to execute in parallel and also how the work-items are divided among work-groups. In the implicit model, the programmer only specifies the total number of work-items to execute in parallel, and the division into work-groups is managed by the OpenCL implementation.

The OpenCL task parallel programming model defines a model in which a single instance of a kernel is executed independent of any index space. It is logically equivalent to executing a kernel on a Compute Unit with a work-group containing a single work-item. Under this model, users can express parallelism by:

- Using vector data types implemented by the device,
- Enqueing multiple tasks, and/or
- Enqueing native kernels developed using a programming model orthogonal to OpenCL.

## 4.3 Related work

There hasn't been as much research focused on explicit parallelism around Modelica as automatic approaches. One implementation that offered the potential for explicit parallel programming was the NestStepModelica [32] that was based on NestStep [33] targeting BSP (Bulk-Synchronous Parallel) computation model which is an abstraction of a restricted message passing architecture and charges cost for communication.

The focus of the work presented here is on parallelizing executions for highly data parallel SPMD (Single Program, Multiple Data) architectures. It generates OpenCL code for parallel algorithms. OpenCL was given priority over CUDA because of its portability. Generating OpenCL code ensures that simulations can be run with parallel support on OpenCL enabled Graphics and Central Processor Units (GPU and CPU). This includes many multi-core CPUs from Intel [25] and Advanced Micro Devices (AMD) [2] as well as a range of GPUs from NVIDIA [49] and AMD (for a complete list of supported devices see [35]). However explicit CUDA code generation is also planned to be supported and the current implementation provides most, if not all, constructs needed for CUDA code generation and execution as well.

## 4.4 ParModelica Extensions

The ParModelica explicit parallel programming extensions available with the current implementation are explained in this section. There are a number of programmability and usage differences compared to the base OpenCL extensions. These are explained in the sections for their respective counterparts.

### 4.4.1 Parallel Variables

Parallel variables are variables allocated in the memory space of the device used for parallel computation. OpenCL code can be executed on host CPU as well as on GPUs whereas CUDA code executes only on GPU. Since the OpenCL and CUDA enabled GPUs use their own local (different from CPU) memory for execution all necessary data should be available on the specific

```
function parvar
protected
  Integer m = 1000;         // Host Scalar
  Integer A[m,m];              // Host Matrix
  Integer B[m,m];           // Host Matrix
  // global and local device memories
  parglobal Integer pm;     // Global Scalar
  parglobal Integer pA[m,m]; // Glob Matrix
  parglobal Integer pB[m,m]; // Glob Matrix
  parlocal Integer pn;      // Local Scalar
  parlocal Integer pS[m];    // Local Array
end parvar;
```

Listing 4.1: ParModelica device variables

| | |
|---|---|
| A := B | Serial assignment. |
| pA := A | Copy from host memory (A) to device global memory (pA). write operation |
| B := pB | Copy from device global memory (pB) to host memory (B). read operation |
| pA := pB | Copy from one device global memory (pB) to other memory space on the same device (pA). |
| pm := m<br>n := pm<br>pn := pm | Scalar versions of the above three assignments. |

Table 4.2: Parallel Variable Assignment Operation

device's memory. Even when running OpenCL computations on CPU the variables used for parallel execution need to be explicitly stated so that the OpenCL drivers and APIs can handle them properly.

ParModelica parallel variables are declared simply by preceding the variable declaration with the corresponding keyword for the intended memory space. The implementation currently does not support allocations or accesses to the constant mmory space, i.e., only global variables identified by `parglobal` and local variables identified by `parlocal` are allowed.

Usage of these parallel variables is shown in Listing 4.1.The first three variables are allocated in the host memory. The last four variables are allocated in the corresponding memory space of the device used for parallel execution. In OpenCL case this can be the host CPU itself or any available GPU.

These parallel variables can be passed between functions as arguments. Copying data between host and parallel device memory is as simple as assigning the variables to each other. The compiler and the runtime system handle the details of the copy operation. The assignments shown in Table 4.2 would all be valid in the function shown above.

Parallel variables can only be declared inside a serial function. Variables in `kernel` (Section 4.4.3) and `parallel` functions (Section 4.4.2) with out the qualifiers `parglobal` or `parlocal` are allocated as private variables.

The current implementation has some restrictions on parallel variables:

- Any computational algorithmic statements involving parallel variables should be in either in parallel for loops, parallel functions or kernel functions. These include arithmetic operations on scalar parallel variables and indexing of parallel arrays. Assignments are allowed anywhere in the algorithmic section of Modelica.

- Parallel variables cannot be initialized with default values. The first declaration in Listing 3.2-1 shows a default value initialization. Some initialization options for arrays currently work. However it is not properly tested and is not supported with this implementation. Full support for default initialization should be supported soon.

### 4.4.2 Parallel Functions

ParModelica parallel functions in this implementation correspond to OpenCL functions defined in kernel files or CUDA's `device` functions. These are functions available independently to every thread executing on a device. Parallel functions in ParModelica are defined in the same way as normal Modelica functions except that they are preceded by the parallel keyword as shown in Listing 4.2.

```
parallel function multiply
  input Integer a;
  input Integer b;
  output Integer c;
algorithm
  c := a * b;
end multiply;
```

Listing 4.2: ParModelica parallel functions

The code for parallel functions is generated in the target language for parallel execution. In the current implementation OpenCL code is generated. Parallel functions have some constraints

- They cannot have parallel for loops in their algorithm sections.

- They can only call other parallel functions or supported built-in functions.

- Recursion is not allowed.

- They can only be called from a body of a parfor loop or from kernel functions, i.e., they are not directly accessible to serial parts of the algorithm.

### 4.4.3 Kernel Functions

```
parkernel function arrayElemWiseMultiply
  input Integer m;
  input Integer A[m];
  input parglobal Integer B[m];
  output parglobal Integer C[m];
  Integer id;
algorithm
  id = oclGetGlobalId(1);
  C[id] := multiply(A[id],B[id]);
end arrayElemWiseMultiply;
```

Listing 4.3: ParModelica kernel functions

Kernels functions correspond to OpenCL and CUDA `kernel` functions and `global` functions respectively. These are entry functions to execution on a device. They can be called from serial parts of Modelica code to start parallel execution on a parallel device. Kernel functions are independently executed by every thread.

ParModelica kernel functions are defined in the same way as normal functions except that they are preceded by the `parkernel` keyword. A possible implementation example is shown in Listing 4.3. multiply is the parallel function listed in 4.2. The special built-in utility function `oclGetGlobalId` is discussed in Section 4.4.6. The number of threads to be used for the kernel execution can be set by using the function `oclSetNumThreads` also discussed in Section 4.4.6. This function should be called before any kernel call if the number of threads and their dimensional arrangement is needed to be different than the default behavior. Otherwise the default number of threads will be used to execute the kernel function which maximum number of threads of the parallel execution device in a one dimensional arrangement. The implementation supports full three dimensional arrangement of work groups and threads.

There are some constraints on usages of ParModelica kernel functions :

- They cannot have parfor loops in their algorithm body.
- They can only call parallel functions or supported built-in functions. They cannot call other kernel functions.
- They cannot be called from a body of parfor loop or from other kernel functions.

```
parfor i in 1:m loop
  for j in 1:pm loop
    ptemp := 0;
    for h in 1:pm loop
      ptemp := multiply(pA[i,h], pB[h,j]) + ptemp;
    end for;
    pC[i,j] := ptemp;
  end for;
end parfor;
```

Listing 4.4: ParModelica parallel for loops

### 4.4.4 Parallel For Loop: parfor

ParModelica parallel for loops are syntactically very similar to normal for loops with some additional constraints on the body of the loop. These constraints are needed to make sure the iterations can be run simultaneously and independently without any specific order while giving the desired result, i.e., no loop-carried dependencies from one iteration to the next. A Modelica parallel for loop is identified by the keyword parfor as shown in Listing 4.4. multiply is the parallel function listed in Listing 4.2.

The iterations of a parfor loop are equally distributed among available processors. If the range of the iteration is smaller than or equal to the number of threads the parallel device supports, each iteration will be done by a separate thread. If for example our device supports 1024 threads and the loop has 512 iterations then 512 threads will be launched and will each execute a separate iteration. If the number of iterations is larger than the number of threads available then some threads might perform more than one iteration. If for example we have a loop with 768 iterations and a device with a 512 thread limit then 512 threads will be launched which will execute iterations 1 to 512. The remaining 256 iterations will be done by the first 256 threads out of the 512 as a second step. In future enhancements parfor will be given the extra feature for specifying the desired number of threads explicitly instead of automatically launching threads as described above.

The choice of target architecture and language has put some constraints on parfor loops.

- All variable references in the loop body must be to `parglobal` variables.
- Iterations should not be dependent on other iterations : no loop-carried dependencies.
- All function calls in the body should be to `parallel` functions or supported built-in functions only.
- The iterator of a parallel for loop must be of integral type.

- The start, step and end values of a parallel for loop iterator should be integral types.

The first constraint is needed since OpenCL executions can be run on another device than the host CPU where the rest of the simulation code is being executed. To make sure that desired data is made available in the device memory before start of parallel execution this rule must be obeyed. If for example OpenMP has been used for the parallel execution then we would not need this constraint since OpenMP code always runs on the CPU with threads accessing CPU shared memory. There is a reason why the compiler does not automatically detect and copy all variables used or referenced in the loop body. Even if it would be reasonable to automatically copy all needed variables to the device memory, which variables should be copied back? Copying all variables back after the execution of the parfor loop means that means potentially unnecessary copy operations would have to be performed. In addition this gives the programmer a better control over the rather expensive memory operations.

### 4.4.5 Built-in Functions

Some built-in functions have been extended to accept parallel variables as arguments. Accepting parallel arguments means that the computations of the function will be performed on the parallel execution device instead of a single thread on the host CPU. The return values from these extended parallel built-in functions are currently only parallel variables. For example consider the built-in function transpose which is used to compute the transpose of a matrix. If a serial matrix is passed to this function as argument the computation will be done on the host CPU and a serial matrix is returned. However if a parallel matrix is given as argument then the computation will be done in parallel on the available device. The return variable will be a parallel variable.

The serial/parallel combination of arguments/return values should be diversified in the future to give more options for the programmer. The compiler should detect the types assigned to return variables and handle any necessary copying automatically. The rules set above on serial/parallel arguments/return-values combination are not hard rules. They are more of choice of implementation and might change in the future. However according to the current implementation any built-in function call involving parallel arguments will return parallel variables.

### 4.4.6 Synchronization and Thread Management

A number of functions related to Synchronization and thread management are also available. These functions are very similar to the OpenCL work-item function (see [34]). These functions are:

- `oclSetNumThreads(..., ...)` : is used to specify the number of threads to be used for a kernel function execution. The implementation supports full three dimensional arrangement work groups and items. The function is overloaded for each corresponding dimensional arrangement and should only be called from inside a serial function. It should be called prior to any kernel function call if the number of threads is to be specified for the kernel. Otherwise the kernels will execute with the default number of threads which is the supported maximum. This function is also overloaded to take just one Integer argument. In this case the given integer value will specify the number of threads or work-items to be launched. The actual arrangement of these threads into work-groups will be decided automatically by the OpenCL runtime system. This usage can be seen in 4.5.

- `oclGetWorkDim()` : returns the number of dimensions used in thread arrangement.

- `oclGetGlobalSize(Integer)` : returns the total number of threads currently executing the function or kernel. This function should only be called from inside a parallel function or kernel function.

- `oclGetLocalSize(Integer)` : returns the total number of threads in the work-group of the calling thread in the given dimension. This function should only be called from inside a parallel function or kernel function.

- `oclGetGlobalId(Integer)` : returns the global id of the calling thread in the given dimension. This function should only be called from inside a parallel function or kernel function.

- `oclGetLocalId(Integer)` : returns the local id of the calling thread in the given dimension. This function should only be called from inside a parallel function or kernel function.

- `oclGetNumGroups(Integer)` : returns the number of work groups in the given dimension. This function should only be called from inside a parallel function or kernel function.

- `oclGetGroupId(Integer)` : returns the work group id of the calling thread in the given dimension. This function should only be called from inside a parallel function or kernel function.

- `oclGlobalBarrier()` : will either flush any variables stored in local memory or queue a memory fence to ensure correct ordering of memory operations to local memory of the parallel device. corresponds to OpenCL `barrier(CLK_GLOBAL_MEM_FENCE)`. This function can only be called from inside a parallel function, kernel function or body of parfor loop.

- `oclGLocalBarrier()` : is used to queue a memory fence to ensure correct ordering of memory operations to global memory of the parallel device. corresponds to OpenCL `barrier(CLK_LOCAL_MEM_FENCE)`. This function can only be called from inside a parallel function, kernel function or body of parfor loop.

### 4.4.7 Extra OpenCL Functionalities

Automatically generated code might not always be as efficient as a manually written code. If the need arrives for a finer control over operations like data distribution and synchronization built-in functions are available for compiling and executing user-written OpenCL code directly from another source file.

- `oclbuild(String)` : This function takes only one String argument. The argument is the name of the OpenCL source file to be built. It returns an integer (type defined as cl_program for clarity) which is used as an id of the built program. This id is used in consequent calls to refer to this OpenCL program. Users can have as many as 10 files built in the same Modelica code (10 within scope) at a time. This limit can be increased in the future. It is just assumed to be enough.

- `oclkernel(oclprogram, String)` : This function takes two arguments. The first one is the id (Integer) of the OpenCL program built by a previous call to `oclbuild`. The second argument is the name the kernel or function in that specific program which the user wants to create a kernel for. Users can create as many as 10 kernels at any time. This function also returns an Integer (type defined as cl_kernel) for the same reason as `oclbuild`.

- `oclsetargs(oclkernel,...)` : This function is used to set arguments to an OpenCL kernel. It takes a variable number of arguments. However the first argument should be the id of the kernel to be executed (an Integer or cl_kernel). After the first argument a variable number of parallel variables follow. These are the actual arguments to the OpenCL kernel. This function does not return anything.

- `oclexecute(oclkernel)` : This function is used to execute a kernel. It takes the id of the kernel as an argument. After executing the kernel the user can copy back any of the arguments attached to the kernel earlier to obtain just the desired results.

Users can declare OpenCL programs as `cl_program` and kernels as `cl_kernel`. These types are just normal type definitions of Integer made just for clarity purposes. They are included with built-in functions so they can be used readily any time. A simple usage of theses utility functions is shown in Listing 4.5. The OpenCL kernel function can perform any operation as long as the arguments match.

```
function externalKernel
  input Integer m;
  parglobal input Integer pA[m,m];
  parglobal input Integer pB[m,m];
  parglobal output Integer pC[m,m];
  cl_program pro;
  cl_kernel ker;
algorithm
  // build the opencl program from the file
  pro := oclbuild("testmat.cl");
  // create the desired kernel from
  // the available kernels in the built program
  ker := oclkernel(pro, "user_func");
  // set the arguments to the kernel created
  oclsetargs(ker,pA,pB,pC,m);
  // set m threads to run.
  oclSetNumThreads(m);
  // run the kernel
  oclexecute(ker);
end externalKernel;
```

Listing 4.5: Loading and executing external OpenCL kernels

All of the above operations are synchronous in OpenCL jargon. They will return only when the specified operation is completed. Further functionality is planned to be added to these functions to provide better control over execution.

## 4.5 ParModelica OpenCL Runtime

The ParModelica OpenCL runtime provides support for execution and interoperation of the generated OpenCL code and the existing OpenModelica runtime system. The runtime system implementation mainly consists of the ParModelica OpenCL-C runtime library and the ParModelica OpenCL utility headers.

### 4.5.1 ParModelica OpenCL-C Runtime Library

The OpenCL-C runtime library provides the mechanisms for connecting the OpenCL device execution and the host serial C execution. The library provides a number of functionalities which allow the OpenModelica normal runtime system and the generated OpenCL code work seamlessly.

It defines the data structures used to represent parallel variables. These data structures are used to characterize parallel variables in the serial C code. The device integer array structure representing an integer array on a parallel device is shown in Listing 4.6.

```
struct gi_array
{
  int ndims;
  __global modelica_integer* dim_size;
  __global modelica_integer* data;
};
typedef struct gi_array integer_array;
```

Listing 4.6: ParModelica device array

The library is also responsible for all OpenCL related initialization operations like device selection, creating contexts on devices, building OpenCL source code from a source file or a string buffer, setting arguments to and launching kernels an so on. It provides clear and concise functions for OpenCL operations by hiding the rather long and complicated OpenCL operations in the background. It also provides all the mechanisms necessary for data transfer operations between the host and the OpenCL device. These include allocation and copy operations of: host to device, device to host and device to device.

Parallel implementations of some built-in functions (e.g. `transpose()`) are also available in the library. All the necessary mechanism for runtime error reporting related to OpenCL operations are also part of this library. Functions for debugging operations are also available in the library.

### 4.5.2 ParModelica OpenCL Utility Headers

Simulating a model using the OpenModelica compiler and runtime system involves C code generation. The generated C code is then compiled and linked with the libraries which provide operations for the simulation. For example the OpenModelica SimulationRuntimeC library (SimulationRuntimeC.a Windows or SimulationRuntimeC.so Linux) provides, among many other things, the structures and operations necessary to represent arrays.

With C/C++ it is possible to implement operations in one library and link them later if they are needed. OpenCL, on the other hand, has no linking mechanisms. This lack of linking mechanisms means that any utility methods needed by the implementation have to be in source code format and have to be compiled with the rest of the generated OpenCL code. the ParModelica utility headers provide this functionality. These headers provide the structures used for representing arrays, methods for copying and manipulating these arrays, a number of Modelica built-in functions, device specific configurations and so on.

## 4.6 Performance Measurements

To be able to evaluate the relative performance and behavior of the new language extensions, performing systematic benchmarking on a set of appropriate Modelica models is required. For this purpose we have constructed a benchmark test suite containing some models that represent heavy and high-performance computation, relevant for simulation on parallel architectures.

### 4.6.1 The MPAR Benchmark Suite

The MPAR benchmark test suite is a set of Modelica test models written using the new parallel extensions. It was developed as a separate Master's thesis work [3] to evaluate the performance of the ParModelica implementations.

The suite contains seven different algorithms from well-known benchmark applications such as the LINear equations software PACKage (LINPACK) [15], and Heat Conduction [40]. These benchmarks have been collected and implemented as algorithmic time-independent Modelica models. The algorithms implemented in this suite involve rather large computations and impose well defined work-loads on the OpenModelica compiler and the run-time system. Moreover, they include different kinds of for-loops and function calls which provide parallelism for domain and task decomposition.

Performance results for three out of the seven test models: Matrix Multiplication, Eigen value computation, and Stationary Heat Conduction, are presented in the next section. Time measurements have been performed for both sequential and parallel implementations of three models on both CPU and GPU architectures. For executing sequential codes generated by the standard sequential OpenModelica compiler an Intel Xeon E5520 CPU [24] which has 16 cores, each with 2.27 GHz clock frequency is used. For executing generated code by our the OpenCL based parallel code generator, we have used the same CPU as well as the NVIDIA Fermi-Tesla M2050 GPU with 448 cores [50].

### 4.6.2 Results

The first test case that is presented is the classic Matrix Multiplication computation. the function computes the result of multiplication of two square matrices of size N as shown in Equation 4.1.

$$c_{ij} = \sum_{i=1}^{N} \sum_{j=1}^{N} \sum_{k=1}^{N} a_{ik} * b_{kj} \tag{4.1}$$

Although rather straight forward, matrix multiplication is a good way to evaluate the performance of the new extensions and implementation. There are a number of ways to parallelize matrix multiplication with the available extensions. It can be implemented as a simple `parfor` loop, a parfor loop combined with parallel functions or a complete optimized parallelization using kernels and better thread and management features. This model presents a very large level of data-parallelism for which a considerable speedup has been achieved as a result of parallel simulation of this model on parallel platforms. Figure 4.2 shows the achieved speedups for the computation of different matrix sizes.

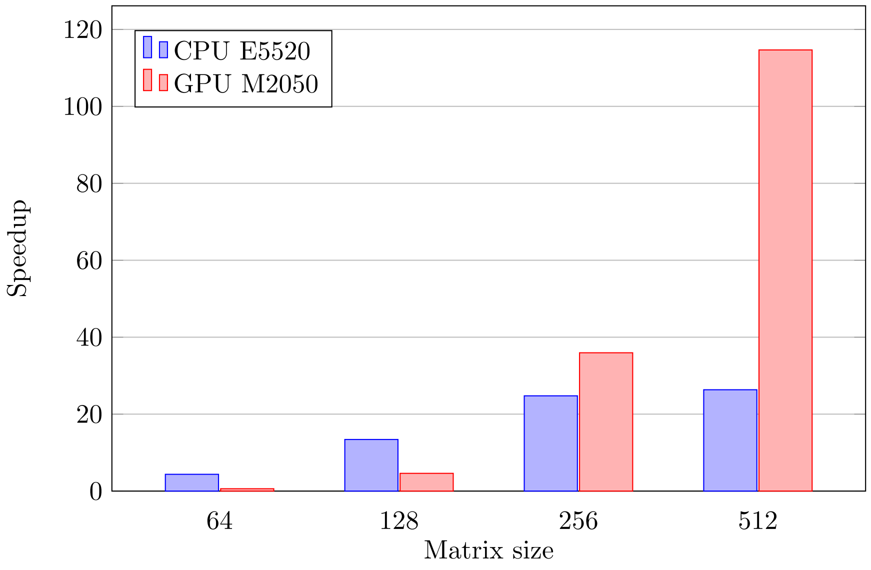


Figure 4.2: Speedups for matrix multiplication

The second test model performs eigenvalue computation which is also a common linear algebra routine with a wide range of applications in many modeling areas. The test model computes all the eigenvalues of computation of a tridiagonal symmetric matrix. The observed speedups for this model are show in Figure 4.3.

The third test models specifies a stationary thermal conduction in a 2-dimensional plate. The heat conduction problem computes the temperature distribution of a surface square plate [0,1] x [0,1] with given initial boundary conditions/temperatures. The thermal distribution can be represented by the differential equation shown in Equation 4.2.

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0, \quad 0 < x, y < 1 \tag{4.2}$$

By defining an equidistant grid $(x_i, y_i)_{i,j=0}^{N+1}$ as shown in Figure 4.4 and us-

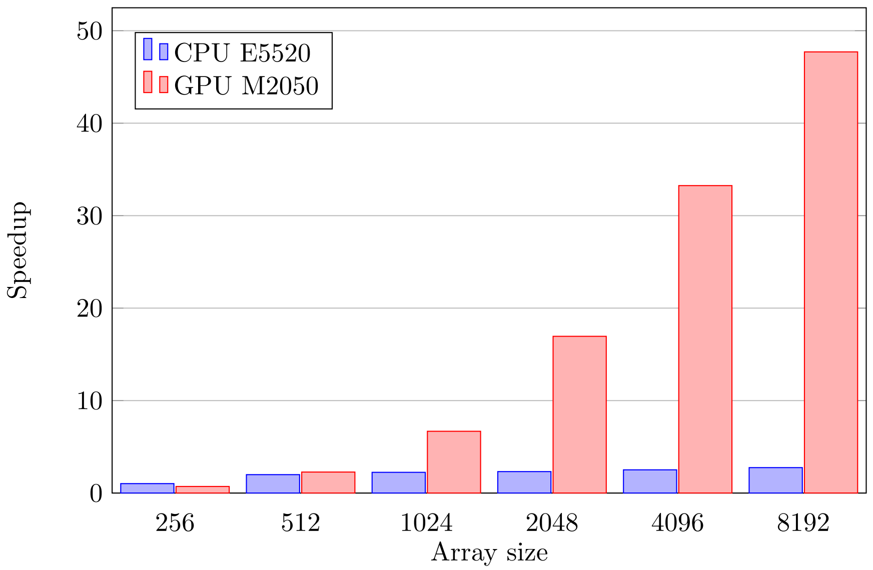


Figure 4.3: Speedups for Eigenvalue computations

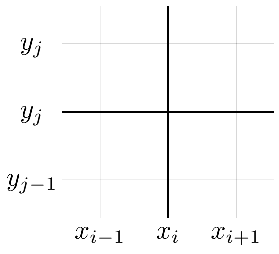


Figure 4.4: Equidistant computation grid

ing finite difference approximation methods [39] the heat distribution equation can be discretized as shown in Equation 4.3

$$-4T_{i,j} + T_{i+1,j} + T_{i-1,j} + T_{i,j-1} + T_{i,j-1} = 0, \tag{4.3}$$

where $T_{i,j} = T(x_i, y_j)$ is the approximated temperature at point $(x_i, y_j)$ in the discretized plate.

Direct numerical methods such as Gaussian Elimination can be used to solve the system of linear equations. However, if the number of grid points is large, iterative methods such as Jacobi method are better suited. The iteration tries to approximate the next solution of the equation using the obtained solutions in previous iterations. If the approximate temperature $T^k_{i,j}$ is known for point $(x_i, y_j)$ at the $k_{th}$ then the value at $T^{k+1}_{i,j}$ can be approximated as shown in Equation 4.4.

$$T^{k+1}_{i,j} = \frac{T^k_{i+1,j} + T^k_{i-1,j} + T^k_{i,j-1} + T^k_{i,j-1}}{4} \quad 1 \leq i, j \leq N - 1 \tag{4.4}$$

This computation is a bit more complicated to parallelize since iterations should be synchronized properly and data dependencies should be obeyed strictly. Specially since values from previous iterations are used to compute values for subsequent iterations.

The achieved speedups for the 2d plate thermal conduction model are shown in Figure 4.5.

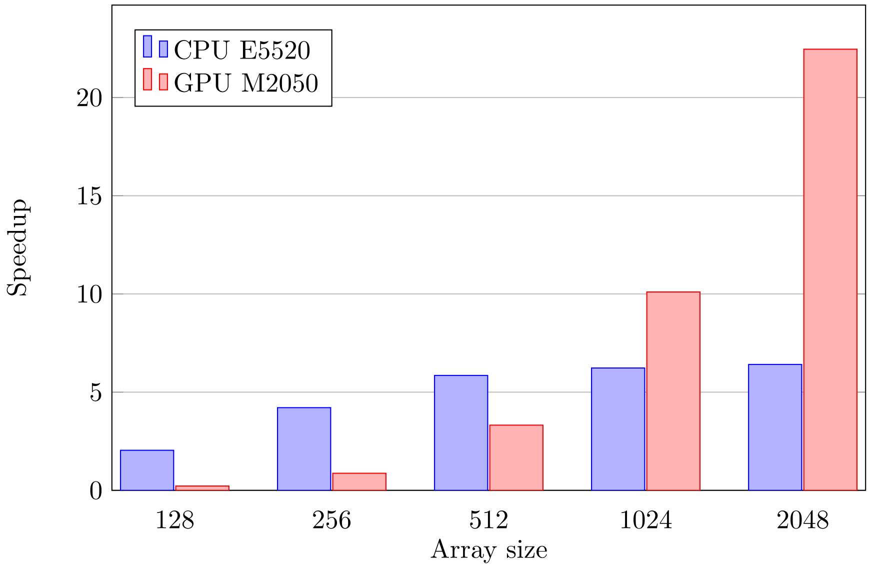


Figure 4.5: Speedups for Eigenvalue computations

## 4.7 Future Work

There are a number of things that can be improved or added to the current implementation of explicit parallel programming approach of ParModelica. Some of these are discussed here.

- CUDA code generation might be supported. This is relatively easy since the high level constructs are available and reusable. The only parts of OMC that will require changes are the SimCode module and the OpenModelica text template Code Generation.

- Semantic Error detection and reporting:
  - Currently there is support for semantic error checking for issues related to parallel operations and scopes of parallel environments. For example calling a parallel function from a non parallel environment (function, model, class ...) will be reported as error by the OpenModelica compiler with an error message. Assignments are also checked for semantic consistency. for example assigning a `parglobal` variable to a `parlocal` variable is detected and reported as an error. However the error detection and reporting right now does not cover all parallelization related issues. This is rather important when and if users start writing parallel code with complex data structures. For example declaring a `parlocal` record (corresponds to `local struct` in OpenCL C) with `parglobal` variables, which is an error, is not reported.
- The current parallel for loop implementation should be enhanced to support the following features:
  - Explicitly stating the desired number of threads to be used for parallel execution.
  - Automatically detecting the parallel variables used in the parfor loop and using only those as arguments to kernels to get a more concise target code. See 4.2.1.1.
  - Specifying the desired target language using annotations. This is important if other parallel programming paradigms are added to the extension e.g. if CUDA or OpenMP code generation is supported, the desired target language can be stated here.
- More built-in functions should be added or extended for parallel execution. The serial/parallel arguments/return-values combination should be extended.
- The extra OpenCL functionalities discussed in Section 4.4.7 should be improved to provide a better control over thread management and execution.
- Overloading of parallel functions should be implemented.

# Chapter 5

# Conclusion

We have presented two parallelization approaches: *automatic* and *explicit*. Both approaches already, with our current prototype, show significant performance improvements over sequential executions and promise further improvements.

The automatic parallelization support by the OpenModelica compiler gives modelers the opportunity to take advantage of their modern high-end multi-core processors without having to learn the details of parallel programming. Support for automatic parallelization is very desirable since it provides the means of parallelizing existing models and libraries without modifications. There are quite a large number of Modelica models already in use. Having to modify or rewrite parts of this large resource of libraries would be too time consuming and error prone.

We have presented a task graph based approach to extract and represents dependency information from equation systems to identify and utilize potential parallelism. The Task Systems library described in this thesis, designed and implemented for handling automatic parallelization, provides the mechanisms to handle this parallelization process conveniently. The library currently provides four clustering algorithms, two schedulers, as well as dynamic profiling and cost estimation mechanisms. The implementation performs clustering and scheduling at runtime. This means that it has the potential to continuously adapt to changes in behavior of the model system throughout simulation. We have also presented the parallelization performance results for selected models from the Modelica Standard Library.

The library is implemented as a loosely coupled, partly independent part of the existing OpenModelica runtime system. This allows for a concurrent development of the parallelization implementation and the rest of the OpenModelica environment. The parallelization implementation requires a minimal interface to be kept consistent by the compiler. For a continuously developing environment like OpenModelica, this minimal interface means that the parallelization can continue to function properly regardless of most changes to the rest of the compiler and simulation environment.

The ParModelica explicit parallel programming constructs presented in Chapter 4 allow modelers to write parallel algorithmic code directly in Modelica. These extensions provide the opportunity for Modelica parallel programming which has been missing so far. These constructs are well integrated with the rest of the language and will hopefully simplify learning and using parallel programming by modelers who are already familiar with Modelica.

Previously the only way to explicitly parallelize an algorithm that is part of a Modelica implementation was to write it as an external function targeting some low level language, usually C/C++, write the algorithm in that language, and parallelize it. To take advantage of modern GPUs this approach needed one more indirection to the GPU framework as well, e.g. Modelica external function calling a C function which in turn uses OpenCL/CUDA. This essentially requires modelers to be familiar with all the languages or frameworks involved. This thesis work has tried to simplify this complicated and error prone process by providing the means to write parallel code directly in Modelica.

The explicit parallel programming approach already shows significant performance improvements over sequential execution for highly data-parallel algorithms as demonstrated by the test results for small set of selected applications presented. Algorithms like matrix multiplication and LU decomposition are used directly or indirectly by more complex algorithms. This means that many models using these kinds of algorithms can benefit from the performance gains for the data parallel portions. Many linear algebra algorithms are already available for Modelica users as either built-in functions or as part of libraries. However not all algorithms implemented by users can be written using only these existing algorithms. Users who need to write their own algorithmic code can use these extensions to improve the performance of their implementation.

The explicit parallel programming extensions, even though they resemble the design of OpenCL and CUDA frameworks, can be used or targeted towards many parallelization frameworks. The implementation currently generates OpenCL code which is portable across a wide range of devices and architectures. However, this does not mean OpenCL is the only target framework that can be used to parallelize models with the current ParModelica extensions. Support for parallelization frameworks other than OpenCL can be added to the OpenModelica compiler by introducing new specific code generator implementations. Modelica code written with these extensions can then be used with the new framework with no or minor changes to the original code.

# Bibliography

[1] Advanced Micro Devices. OpenCL Static C++ Kernel Language Extension. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/CPP_kernel_language.pdf, 2013. [Online; accessed 10-April-2015].

[2] Advanced Micro Devices, Inc. The AMD OpenCL Zone. http://developer.amd.com/tools-and-sdks/opencl-zone/, 2015. [Online; accessed 10-April-2015].

[3] Afshin Hemmati Moghadam. Modelica PARallel benchmark suite (MPAR) - a test suite for evaluating the performance of parallel simulations of Modelica models). http://www.diva-portal.org/smash/get/diva2:461422/FULLTEXT01.pdf, 2012. [Online; accessed 10-April-2015].

[4] J. Åkesson. Optimica an extension of modelica supporting dynamic optimization. In *Proc. 6th International Modelica Conference 2008*, 2008.

[5] Alexander Souza. Combinatorial Algorithms). http://www2.informatik.hu-berlin.de/alcox/lehre/lvws1011/coalg/combinatorial_algorithms.pdf, 2011. [Online; accessed 10-April-2015].

[6] P. Aronsson. Automatic parallelization of equation-based simulation programs. 2006.

[7] Blaise Barney, Lawrence Livermore National Laboratory. OpenMP). https://computing.llnl.gov/tutorials/openMP/, 2014. [Online; accessed 10-April-2015].

[8] Blaise Barney, Lawrence Livermore National Laboratory. POSIX Threads Programming). https://computing.llnl.gov/tutorials/pthreads/, 2014. [Online; accessed 10-April-2015].

[9] Boost.org. The Boost Graph Library (BGL). http://www.boost.org/doc/libs/1_57_0/libs/graph/doc/index.html, 2011. [Online; accessed 10-April-2015].

[10] Boost.org. Boost C++ Libraries. `http://www.boost.org/`, 2015. [Online; accessed 10-April-2015].

[11] Boost.org. Boost Software License. `http://www.boost.org/users/license.html`, 2015. [Online; accessed 10-April-2015].

[12] F. Casella. A strategy for parallel simulation of declarative object-oriented models of generalized physical networks. In *5th Workshop on Equation-Based Object-Oriented Modeling Languages and Tools EOOLT, Linköping Univ. Electronic Press, Nottingham, UK*, pages 45–51, 2013.

[13] F. E. Cellier and E. Kofman. *Continuous System Simulation.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[14] E. Davis and J. M. Jaffe. Algorithms for scheduling tasks on unrelated processors. *Journal of the ACM (JACM)*, 28(4):721–736, 1981.

[15] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK users' guide*, volume 8. Siam, 1979.

[16] G. Dósa and J. Sgall. First fit bin packing: A tight analysis. In *STACS*, pages 538–549. Citeseer, 2013.

[17] Free Software Foundation. GNU General Public License, version 2). `https://www.gnu.org/licenses/gpl-2.0.html`, 2014. [Online; accessed 10-April-2015].

[18] P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach.* John Wiley & Sons, 2014.

[19] M. Gebremedhin, A. H. Moghadam, P. Fritzson, and K. Stavåker. Parmodelica: Extending the algorithmic subset of modelica with explicit parallel language constructs for multi-core simulation.

[20] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581, 1966.

[21] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429, 1969.

[22] D. S. Hochbaum and D. B. Shmoys. A polynomial approximation scheme for scheduling on uniform processors: Using the dual approximation approach. *SIAM journal on computing*, 17(3):539–551, 1988.

[23] O. H. Ibarra and C. E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of the ACM (JACM)*, 24(2):280–289, 1977.

[24] Intel Corporation. ARK — Intel; Xeon; Processor E5520 (8M Cache, 2.26 GHz, 5.86 GT/s Intel; QPI). `http://ark.intel.com/products/40200/Intel-Xeon-Processor-E5520-`, 2015. [Online; accessed 10-April-2015].

[25] Intel Corporation. OpenCL Technology - Intel Developer Zone. `https://software.intel.com/en-us/intel-opencl`, 2015. [Online; accessed 10-April-2015].

[26] Intel Corporation. Threading Building Blocks (Intel®TBB). `https://www.threadingbuildingblocks.org/`, 2015. [Online; accessed 10-April-2015].

[27] Intel Corporation. Threading Building Blocks (Intel®TBB) Flow Graph. `https://www.threadingbuildingblocks.org/docs/help/reference/flow_graph.htm`, 2015. [Online; accessed 10-April-2015].

[28] Iso.org. ISO/IEC 9899:1999 - Programming languages - C. `http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=29237`, 2011. [Online; accessed 10-April-2015].

[29] D. S. Johnson. Approximation algorithms for combinatorial problems. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 38–49. ACM, 1973.

[30] D. S. Johnson. *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973.

[31] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3(4):299–325, 1974.

[32] C. Kessler, P. Fritzson, and M. Eriksson. Neststepmodelica–mathematical modeling and bulk-synchronous parallel simulation. In *Applied Parallel Computing. State of the Art in Scientific Computing*, pages 1006–1015. Springer, 2007.

[33] C. W. Keßler. Neststep: nested parallelism and virtual shared memory for the bsp model. *The Journal of Supercomputing*, 17(3):245–262, 2000.

[34] Khronos Group. Work-Item Built-In Functions. `https://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/workItemFunctions.html`, 2009. [Online; accessed 10-April-2015].

[35] Khronos Group. Conformant Products. `https://www.khronos.org/conformance/adopters/conformant-products/#opencl`, 2015. [Online; accessed 10-April-2015].

[36] Khronos Group. OpenGL - The Industry Standard for High Performance Graphics. https://www.opengl.org/, 2015. [Online; accessed 10-April-2015].

[37] Khronos OpenCL Working Group. The OpenCL Specification. Version: 1.0 Document Revision: 29. https://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf, 2012. [Online; accessed 10-April-2015].

[38] J. K. Lenstra, D. B. Shmoys, and É. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical programming*, 46(1-3):259–271, 1990.

[39] R. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems*. SIAM e-books. Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 2007.

[40] J. H. Lienhard. *A heat transfer textbook*. Courier Corporation, 2013.

[41] H. Lundvall. Automatic parallelization using pipelining for equation-based simulation languages. 2008.

[42] Michael M. Tiller. Modelica by Example . http://book.xogeny.com/, 2014. [Online; accessed 10-April-2015].

[43] Microsoft. Developing games - Windows app development. https://msdn.microsoft.com/en-us/library/windows/desktop/ff476331%28v=vs.85%29.aspx, 2015. [Online; accessed 10-April-2015].

[44] Microsoft. DirectX. https://msdn.microsoft.com/library/windows/apps/hh452744, 2015. [Online; accessed 10-April-2015].

[45] Modelica Association. Modelica - A Unified Object-Oriented Language for Systems Modeling Language Specification. Version 3.3 . https://www.modelica.org/documents/ModelicaSpec33.pdf, 2012. [Online; accessed 10-April-2015].

[46] Modelica Association. Modelica and the Modelica Association. https://www.modelica.org/, 2015. [Online; accessed 10-April-2015].

[47] Modelica Association. Modelica Association. https://www.modelica.org/association/, 2015. [Online; accessed 10-April-2015].

[48] Modelica Association. Modelica Standard Library. https://github.com/modelica/Modelica, 2015. [Online; accessed 10-April-2015].

[49] NVIDIA Corporation. NVIDIA OpenCL. https://developer.nvidia.com/opencl, 2015. [Online; accessed 10-April-2015].

[50] NVIDIA Corporation. Tesla M2050 / M2070 GPU Computing Module. `http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_M2050_M2070_Apr10_LowRes.pdf`, 2015. [Online; accessed 10-April-2015].

[51] A. Pop and P. Fritzson. Metamodelica: A unified equation-based semantical and mathematical modeling language. In *Modular Programming Languages*, pages 211–229. Springer, 2006.

[52] M. Sjölund, M. Gebremedhin, and P. Fritzson. Parallelizing equation-based models for simulation on multi-core platforms by utilizing model structure. In *CPC 2013, 17th Workshop on Compilers for Parallel Computing*, 2013.

[53] M. Walther, V. Waurich, C. Schubert, I. Gubsch, A. Hofmann, L. Mikelsons, I. Gubsch, and C. Schubert. Equation based parallelization of modelica models. In *Proceedings of the 10th International Modelica Conference*, 2014.

Department of Computer and Information Science
Linköpings universitet

## Licentiate Theses

## Linköpings Studies in Science and Technology
## Faculty of Arts and Sciences

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 28 **Arne Jönsson, Mikael Patel:** An Interactive Flowcharting Technique for Communicating and Realizing Algorithms, 1984.

No 29 **Johnny Eckerland:** Retargeting of an Incremental Code Generator, 1984.

No 48 **Henrik Nordin:** On the Use of Typical Cases for Knowledge-Based Consultation and Teaching, 1985.

No 52 **Zebo Peng:** Steps Towards the Formalization of Designing VLSI Systems, 1985.

No 60 **Johan Fagerström:** Simulation and Evaluation of Architecture based on Asynchronous Processes, 1985.

No 71 **Jalal Maleki:** ICONStraint, A Dependency Directed Constraint Maintenance System, 1987.

No 72 **Tony Larsson:** On the Specification and Verification of VLSI Systems, 1986.

No 73 **Ola Strömfors:** A Structure Editor for Documents and Programs, 1986.

No 74 **Christos Levcopoulos:** New Results about the Approximation Behavior of the Greedy Triangulation, 1986.

No 104 **Shamsul I. Chowdhury: S**tatistical Expert Systems - a Special Application Area for Knowledge-Based Computer Methodology, 1987.

No 108 **Rober Bilos:** Incremental Scanning and Token-Based Editing, 1987.

No 111 **Hans Block:** SPORT-SORT Sorting Algorithms and Sport Tournaments, 1987.

No 113 **Ralph Rönnquist:** Network and Lattice Based Approaches to the Representation of Knowledge, 1987.

No 118 **Mariam Kamkar, Nahid Shahmehri:** Affect-Chaining in Program Flow Analysis Applied to Queries of Programs, 1987.

No 126 **Dan Strömberg:** Transfer and Distribution of Application Programs, 1987.

No 127 **Kristian Sandahl:** Case Studies in Knowledge Acquisition, Migration and User Acceptance of Expert Systems, 1987.

No 139 **Christer Bäckström:** Reasoning about Interdependent Actions, 1988.

No 140 **Mats Wirén:** On Control Strategies and Incrementality in Unification-Based Chart Parsing, 1988.

No 146 **Johan Hultman:** A Software System for Defining and Controlling Actions in a Mechanical System, 1988.

No 150 **Tim Hansen:** Diagnosing Faults using Knowledge about Malfunctioning Behavior, 1988.

No 165 **Jonas Löwgren:** Supporting Design and Management of Expert System User Interfaces, 1989.

No 166 **Ola Petersson:** On Adaptive Sorting in Sequential and Parallel Models, 1989.

No 174 **Yngve Larsson:** Dynamic Configuration in a Distributed Environment, 1989.

No 177 **Peter Åberg:** Design of a Multiple View Presentation and Interaction Manager, 1989.

No 181 **Henrik Eriksson:** A Study in Domain-Oriented Tool Support for Knowledge Acquisition, 1989.

No 184 **Ivan Rankin:** The Deep Generation of Text in Expert Critiquing Systems, 1989.

No 187 **Simin Nadjm-Tehrani:** Contributions to the Declarative Approach to Debugging Prolog Programs, 1989.

No 189 **Magnus Merkel:** Temporal Information in Natural Language, 1989.

No 196 **Ulf Nilsson:** A Systematic Approach to Abstract Interpretation of Logic Programs, 1989.

No 197 **Staffan Bonnier:** Horn Clause Logic with External Procedures: Towards a Theoretical Framework, 1989.

No 203 **Christer Hansson:** A Prototype System for Logical Reasoning about Time and Action, 1990.

No 212 **Björn Fjellborg:** An Approach to Extraction of Pipeline Structures for VLSI High-Level Synthesis, 1990.

No 230 **Patrick Doherty:** A Three-Valued Approach to Non-Monotonic Reasoning, 1990.

No 237 **Tomas Sokolnicki:** Coaching Partial Plans: An Approach to Knowledge-Based Tutoring, 1990.

No 250 **Lars Strömberg:** Postmortem Debugging of Distributed Systems, 1990.

No 253 **Torbjörn Näslund:** SLDFA-Resolution - Computing Answers for Negative Queries, 1990.

No 260 **Peter D. Holmes:** Using Connectivity Graphs to Support Map-Related Reasoning, 1991.

No 283 **Olof Johansson:** Improving Implementation of Graphical User Interfaces for Object-Oriented Knowledge- Bases, 1991.

No 298 **Rolf G Larsson:** Aktivitetsbaserad kalkylering i ett nytt ekonomisystem, 1991.

No 318 **Lena Srömbäck:** Studies in Extended Unification-Based Formalism for Linguistic Description: An Algorithm for Feature Structures with Disjunction and a Proposal for Flexible Systems, 1992.

No 319 **Mikael Pettersson:** DML-A Language and System for the Generation of Efficient Compilers from Denotational Specification, 1992.

No 326 **Andreas Kågedal:** Logic Programming with External Procedures: an Implementation, 1992.

No 328 **Patrick Lambrix:** Aspects of Version Management of Composite Objects, 1992.

No 333 **Xinli Gu:** Testability Analysis and Improvement in High-Level Synthesis Systems, 1992.

No 335 **Torbjörn Näslund:** On the Role of Evaluations in Iterative Development of Managerial Support Systems, 1992.

No 348 **Ulf Cederling:** Industrial Software Development - a Case Study, 1992.

No 352 **Magnus Morin:** Predictable Cyclic Computations in Autonomous Systems: A Computational Model and Implementation, 1992.

No 371 **Mehran Noghabai:** Evaluation of Strategic Investments in Information Technology, 1993.

No 378 **Mats Larsson:** A Transformational Approach to Formal Digital System Design, 1993.

No 380 **Johan Ringström:** Compiler Generation for Parallel Languages from Denotational Specifications, 1993.
No 381 **Michael Jansson:** Propagation of Change in an Intelligent Information System, 1993.
No 383 **Jonni Harrius:** An Architecture and a Knowledge Representation Model for Expert Critiquing Systems, 1993.
No 386 **Per Österling:** Symbolic Modelling of the Dynamic Environments of Autonomous Agents, 1993.
No 398 **Johan Boye:** Dependency-based Groudness Analysis of Functional Logic Programs, 1993.
No 402 **Lars Degerstedt:** Tabulated Resolution for Well Founded Semantics, 1993.
No 406 **Anna Moberg:** Satellitkontor - en studie av kommunikationsmönster vid arbete på distans, 1993.
No 414 **Peter Carlsson:** Separation av företagsledning och finansiering - fallstudier av företagsledarutköp ur ett agent-teoretiskt perspektiv, 1994.
No 417 **Camilla Sjöström:** Revision och lagreglering - ett historiskt perspektiv, 1994.
No 436 **Cecilia Sjöberg:** Voices in Design: Argumentation in Participatory Development, 1994.
No 437 **Lars Viklund:** Contributions to a High-level Programming Environment for a Scientific Computing, 1994.
No 440 **Peter Loborg:** Error Recovery Support in Manufacturing Control Systems, 1994.
FHS 3/94 **Owen Eriksson:** Informationssystem med verksamhetskvalitet - utvärdering baserat på ett verksamhetsinriktat och samskapande perspektiv, 1994.
FHS 4/94 **Karin Pettersson:** Informationssystemstrukturering, ansvarsfördelning och användarinflytande - En komparativ studie med utgångspunkt i två informationssystemstrategier, 1994.
No 441 **Lars Poignant:** Informationsteknologi och företagsetablering - Effekter på produktivitet och region, 1994.
No 446 **Gustav Fahl:** Object Views of Relational Data in Multidatabase Systems, 1994.
No 450 **Henrik Nilsson:** A Declarative Approach to Debugging for Lazy Functional Languages, 1994.
No 451 **Jonas Lind:** Creditor - Firm Relations: an Interdisciplinary Analysis, 1994.
No 452 **Martin Sköld:** Active Rules based on Object Relational Queries - Efficient Change Monitoring Techniques, 1994.
No 455 **Pär Carlshamre:** A Collaborative Approach to Usability Engineering: Technical Communicators and System Developers in Usability-Oriented Systems Development, 1994.
FHS 5/94 **Stefan Cronholm:** Varför CASE-verktyg i systemutveckling? - En motiv- och konsekvensstudie avseende arbetssätt och arbetsformer, 1994.
No 462 **Mikael Lindvall:** A Study of Traceability in Object-Oriented Systems Development, 1994.
No 463 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av Sandviks förvärv av Bahco Verktyg, 1994.
No 464 **Hans Olsén:** Collage Induction: Proving Properties of Logic Programs by Program Synthesis, 1994.
No 469 **Lars Karlsson:** Specification and Synthesis of Plans Using the Features and Fluents Framework, 1995.
No 473 **Ulf Söderman:** On Conceptual Modelling of Mode Switching Systems, 1995.
No 475 **Choong-ho Yi:** Reasoning about Concurrent Actions in the Trajectory Semantics, 1995.
No 476 **Bo Lagerström:** Successiv resultatavräkning av pågående arbeten. - Fallstudier i tre byggföretag, 1995.
No 478 **Peter Jonsson:** Complexity of State-Variable Planning under Structural Restrictions, 1995.
FHS 7/95 **Anders Avdic:** Arbetsintegrerad systemutveckling med kalkylprogram, 1995.
No 482 **Eva L Ragnemalm:** Towards Student Modelling through Collaborative Dialogue with a Learning Companion, 1995.
No 488 **Eva Toller:** Contributions to Parallel Multiparadigm Languages: Combining Object-Oriented and Rule-Based Programming, 1995.
No 489 **Erik Stoy:** A Petri Net Based Unified Representation for Hardware/Software Co-Design, 1995.
No 497 **Johan Herber:** Environment Support for Building Structured Mathematical Models, 1995.
No 498 **Stefan Svenberg:** Structure-Driven Derivation of Inter-Lingual Functor-Argument Trees for Multi-Lingual Generation, 1995.
No 503 **Hee-Cheol Kim:** Prediction and Postdiction under Uncertainty, 1995.
FHS 8/95 **Dan Fristedt:** Metoder i användning - mot förbättring av systemutveckling genom situationell metodkunskap och metodanalys, 1995.
FHS 9/95 **Malin Bergvall:** Systemförvaltning i praktiken - en kvalitativ studie avseende centrala begrepp, aktiviteter och ansvarsroller, 1995.
No 513 **Joachim Karlsson:** Towards a Strategy for Software Requirements Selection, 1995.
No 517 **Jakob Axelsson:** Schedulability-Driven Partitioning of Heterogeneous Real-Time Systems, 1995.
No 518 **Göran Forslund:** Toward Cooperative Advice-Giving Systems: The Expert Systems Experience, 1995.
No 522 **Jörgen Andersson:** Bilder av småföretagares ekonomistyrning, 1995.
No 538 **Staffan Flodin:** Efficient Management of Object-Oriented Queries with Late Binding, 1996.
No 545 **Vadim Engelson:** An Approach to Automatic Construction of Graphical User Interfaces for Applications in Scientific Computing, 1996.
No 546 **Magnus Werner :** Multidatabase Integration using Polymorphic Queries and Views, 1996.
FiF-a 1/96 **Mikael Lind:** Affärsprocessinriktad förändringsanalys - utveckling och tillämpning av synsätt och metod, 1996.
No 549 **Jonas Hallberg:** High-Level Synthesis under Local Timing Constraints, 1996.
No 550 **Kristina Larsen:** Förutsättningar och begränsningar för arbete på distans - erfarenheter från fyra svenska företag. 1996.
No 557 **Mikael Johansson:** Quality Functions for Requirements Engineering Methods, 1996.
No 558 **Patrik Nordling:** The Simulation of Rolling Bearing Dynamics on Parallel Computers, 1996.
No 561 **Anders Ekman:** Exploration of Polygonal Environments, 1996.
No 563 **Niclas Andersson:** Compilation of Mathematical Models to Parallel Code, 1996.

| | |
|---|---|
| No 567 | **Johan Jenvald:** Simulation and Data Collection in Battle Training, 1996. |
| No 575 | **Niclas Ohlsson:** Software Quality Engineering by Early Identification of Fault-Prone Modules, 1996. |
| No 576 | **Mikael Ericsson:** Commenting Systems as Design Support—A Wizard-of-Oz Study, 1996. |
| No 587 | **Jörgen Lindström:** Chefers användning av kommunikationsteknik, 1996. |
| No 589 | **Esa Falkenroth:** Data Management in Control Applications - A Proposal Based on Active Database Systems, 1996. |
| No 591 | **Niclas Wahllöf:** A Default Extension to Description Logics and its Applications, 1996. |
| No 595 | **Annika Larsson:** Ekonomisk Styrning och Organisatorisk Passion - ett interaktivt perspektiv, 1997. |
| No 597 | **Ling Lin:** A Value-based Indexing Technique for Time Sequences, 1997. |
| No 598 | **Rego Granlund:** $C^3$Fire - A Microworld Supporting Emergency Management Training, 1997. |
| No 599 | **Peter Ingels:** A Robust Text Processing Technique Applied to Lexical Error Recovery, 1997. |
| No 607 | **Per-Arne Persson:** Toward a Grounded Theory for Support of Command and Control in Military Coalitions, 1997. |
| No 609 | **Jonas S Karlsson:** A Scalable Data Structure for a Parallel Data Server, 1997. |
| FiF-a 4 | **Carita Åbom:** Videomötesteknik i olika affärssituationer - möjligheter och hinder, 1997. |
| FiF-a 6 | **Tommy Wedlund**: Att skapa en företagsanpassad systemutvecklingsmodell - genom rekonstruktion, värdering och vidareutveckling i T50-bolag inom ABB, 1997. |
| No 615 | **Silvia Coradeschi**: A Decision-Mechanism for Reactive and Coordinated Agents, 1997. |
| No 623 | **Jan Ollinen:** Det flexibla kontorets utveckling på Digital - Ett stöd för multiflex? 1997. |
| No 626 | **David Byers:** Towards Estimating Software Testability Using Static Analysis, 1997. |
| No 627 | **Fredrik Eklund:** Declarative Error Diagnosis of GAPLog Programs, 1997. |
| No 629 | **Gunilla Ivefors:** Krigsspel och Informationsteknik inför en oförutsägbar framtid, 1997**.** |
| No 631 | **Jens-Olof Lindh:** Analysing Traffic Safety from a Case-Based Reasoning Perspective, 1997 |
| No 639 | **Jukka Mäki-Turja:**. Smalltalk - a suitable Real-Time Language, 1997. |
| No 640 | **Juha Takkinen**: CAFE: Towards a Conceptual Model for Information Management in Electronic Mail, 1997. |
| No 643 | **Man Lin**: Formal Analysis of Reactive Rule-based Programs, 1997. |
| No 653 | **Mats Gustafsson**: Bringing Role-Based Access Control to Distributed Systems, 1997. |
| FiF-a 13 | **Boris Karlsson:** Metodanalys för förståelse och utveckling av systemutvecklingsverksamhet. Analys och värdering av systemutvecklingsmodeller och dess användning, 1997. |
| No 674 | **Marcus Bjäreland:** Two Aspects of Automating Logics of Action and Change - Regression and Tractability, 1998. |
| No 676 | **Jan Håkegård**: Hierarchical Test Architecture and Board-Level Test Controller Synthesis, 1998. |
| No 668 | **Per-Ove Zetterlund**: Normering av svensk redovisning - En studie av tillkomsten av Redovisningsrådets rekommendation om koncernredovisning (RR01:91), 1998. |
| No 675 | **Jimmy Tjäder**: Projektledaren & planen - en studie av projektledning i tre installations- och systemutvecklingsprojekt, 1998. |
| FiF-a 14 | **Ulf Melin**: Informationssystem vid ökad affärs- och processorientering - egenskaper, strategier och utveckling, 1998. |
| No 695 | **Tim Heyer**: COMPASS: Introduction of Formal Methods in Code Development and Inspection, 1998. |
| No 700 | **Patrik Hägglund:** Programming Languages for Computer Algebra, 1998. |
| FiF-a 16 | **Marie-Therese Christiansson:** Inter-organisatorisk verksamhetsutveckling - metoder som stöd vid utveckling av partnerskap och informationssystem, 1998. |
| No 712 | **Christina Wennestam:** Information om immateriella resurser. Investeringar i forskning och utveckling samt i personal inom skogsindustrin, 1998. |
| No 719 | **Joakim Gustafsson:** Extending Temporal Action Logic for Ramification and Concurrency, 1998. |
| No 723 | **Henrik André-Jönsson:** Indexing time-series data using text indexing methods, 1999. |
| No 725 | **Erik Larsson:** High-Level Testability Analysis and Enhancement Techniques, 1998. |
| No 730 | **Carl-Johan Westin:** Informationsförsörjning: en fråga om ansvar - aktiviteter och uppdrag i fem stora svenska organisationers operativa informationsförsörjning, 1998. |
| No 731 | **Åse Jansson:** Miljöhänsyn - en del i företags styrning, 1998. |
| No 733 | **Thomas Padron-McCarthy:** Performance-Polymorphic Declarative Queries, 1998. |
| No 734 | **Anders Bäckström:** Värdeskapande kreditgivning - Kreditriskhantering ur ett agentteoretiskt perspektiv, 1998. |
| FiF-a 21 | **Ulf Seigerroth:** Integration av förändringsmetoder - en modell för välgrundad metodintegration, 1999. |
| FiF-a 22 | **Fredrik Öberg:** Object-Oriented Frameworks - A New Strategy for Case Tool Development, 1998. |
| No 737 | **Jonas Mellin:** Predictable Event Monitoring, 1998. |
| No 738 | **Joakim Eriksson:** Specifying and Managing Rules in an Active Real-Time Database System, 1998. |
| FiF-a 25 | **Bengt E W Andersson:** Samverkande informationssystem mellan aktörer i offentliga åtaganden - En teori om aktörsarenor i samverkan om utbyte av information, 1998. |
| No 742 | **Pawel Pietrzak:** Static Incorrectness Diagnosis of CLP (FD), 1999. |
| No 748 | **Tobias Ritzau:** Real-Time Reference Counting in RT-Java, 1999. |
| No 751 | **Anders Ferntoft:** Elektronisk affärskommunikation - kontaktkostnader och kontaktprocesser mellan kunder och leverantörer på producentmarknader, 1999. |
| No 752 | **Jo Skåmedal:** Arbete på distans och arbetsformens påverkan på resor och resmönster, 1999. |
| No 753 | **Johan Alvehus:** Mötets metaforer. En studie av berättelser om möten, 1999. |

| | |
|---|---|
| No 754 | **Magnus Lindahl:** Bankens villkor i låneavtal vid kreditgivning till högt belånade företagsförvärv: En studie ur ett agentteoretiskt perspektiv, 2000. |
| No 766 | **Martin V. Howard:** Designing dynamic visualizations of temporal data, 1999. |
| No 769 | **Jesper Andersson:** Towards Reactive Software Architectures, 1999. |
| No 775 | **Anders Henriksson:** Unique kernel diagnosis, 1999. |
| FiF-a 30 | **Pär J. Ågerfalk:** Pragmatization of Information Systems - A Theoretical and Methodological Outline, 1999. |
| No 787 | **Charlotte Björkegren:** Learning for the next project - Bearers and barriers in knowledge transfer within an organisation, 1999. |
| No 788 | **Håkan Nilsson:** Informationsteknik som drivkraft i granskningsprocessen - En studie av fyra revisionsbyråer, 2000. |
| No 790 | **Erik Berglund:** Use-Oriented Documentation in Software Development, 1999. |
| No 791 | **Klas Gäre:** Verksamhetsförändringar i samband med IS-införande, 1999. |
| No 800 | **Anders Subotic:** Software Quality Inspection, 1999. |
| No 807 | **Svein Bergum**: Managerial communication in telework, 2000. |
| No 809 | **Flavius Gruian:** Energy-Aware Design of Digital Systems, 2000. |
| FiF-a 32 | **Karin Hedström:** Kunskapsanvändning och kunskapsutveckling hos verksamhetskonsulter - Erfarenheter från ett FOU-samarbete, 2000. |
| No 808 | **Linda Askenäs:** Affärssystemet - En studie om teknikens aktiva och passiva roll i en organisation, 2000. |
| No 820 | **Jean Paul Meynard:** Control of industrial robots through high-level task programming, 2000. |
| No 823 | **Lars Hult:** Publika Gränsytor - ett designexempel, 2000. |
| No 832 | **Paul Pop:** Scheduling and Communication Synthesis for Distributed Real-Time Systems, 2000. |
| FiF-a 34 | **Göran Hultgren:** Nätverksinriktad Förändringsanalys - perspektiv och metoder som stöd för förståelse och utveckling av affärsrelationer och informationssystem, 2000. |
| No 842 | **Magnus Kald:** The role of management control systems in strategic business units, 2000. |
| No 844 | **Mikael Cäker:** Vad kostar kunden? Modeller för intern redovisning, 2000. |
| FiF-a 37 | **Ewa Braf**: Organisationers kunskapsverksamheter - en kritisk studie av ”knowledge management”, 2000. |
| FiF-a 40 | **Henrik Lindberg:** Webbaserade affärsprocesser - Möjligheter och begränsningar, 2000. |
| FiF-a 41 | **Benneth Christiansson:** Att komponentbasera informationssystem - Vad säger teori och praktik?, 2000. |
| No. 854 | **Ola Pettersson:** Deliberation in a Mobile Robot, 2000. |
| No 863 | **Dan Lawesson**: Towards Behavioral Model Fault Isolation for Object Oriented Control Systems, 2000. |
| No 881 | **Johan Moe:** Execution Tracing of Large Distributed Systems, 2001. |
| No 882 | **Yuxiao Zhao:** XML-based Frameworks for Internet Commerce and an Implementation of B2B e-procurement, 2001. |
| No 890 | **Annika Flycht-Eriksson:** Domain Knowledge Management in Information-providing Dialogue systems, 2001. |
| FiF-a 47 | **Per-Arne Segerkvist**: Webbaserade imaginära organisationers samverkansformer: Informationssystemarkitektur och aktörssamverkan som förutsättningar för affärsprocesser, 2001. |
| No 894 | **Stefan Svarén:** Styrning av investeringar i divisionaliserade företag - Ett koncernperspektiv, 2001. |
| No 906 | **Lin Han**: Secure and Scalable E-Service Software Delivery, 2001. |
| No 917 | **Emma Hansson:** Optionsprogram för anställda - en studie av svenska börsföretag, 2001. |
| No 916 | **Susanne Odar:** IT som stöd för strategiska beslut, en studie av datorimplementerade modeller av verksamhet som stöd för beslut om anskaffning av JAS 1982, 2002. |
| FiF-a-49 | **Stefan Holgersson:** IT-system och filtrering av verksamhetskunskap - kvalitetsproblem vid analyser och beslutsfattande som bygger på uppgifter hämtade från polisens IT-system, 2001. |
| FiF-a-51 | **Per Oscarsson:** Informationssäkerhet i verksamheter - begrepp och modeller som stöd för förståelse av informationssäkerhet och dess hantering, 2001. |
| No 919 | **Luis Alejandro Cortes:** A Petri Net Based Modeling and Verification Technique for Real-Time Embedded Systems, 2001. |
| No 915 | **Niklas Sandell:** Redovisning i skuggan av en bankkris - Värdering av fastigheter. 2001. |
| No 931 | **Fredrik Elg:** Ett dynamiskt perspektiv på individuella skillnader av heuristisk kompetens, intelligens, mentala modeller, mål och konfidens i kontroll av mikrovärlden Moro, 2002. |
| No 933 | **Peter Aronsson:** Automatic Parallelization of Simulation Code from Equation Based Simulation Languages, 2002. |
| No 938 | **Bourhane Kadmiry**: Fuzzy Control of Unmanned Helicopter, 2002. |
| No 942 | **Patrik Haslum**: Prediction as a Knowledge Representation Problem: A Case Study in Model Design, 2002. |
| No 956 | **Robert Sevenius:** On the instruments of governance - A law & economics study of capital instruments in limited liability companies, 2002. |
| FiF-a 58 | **Johan Petersson:** Lokala elektroniska marknadsplatser - informationssystem för platsbundna affärer, 2002. |
| No 964 | **Peter Bunus:** Debugging and Structural Analysis of Declarative Equation-Based Languages, 2002. |
| No 973 | **Gert Jervan:** High-Level Test Generation and Built-In Self-Test Techniques for Digital Systems, 2002. |
| No 958 | **Fredrika Berglund:** Management Control and Strategy - a Case Study of Pharmaceutical Drug Development, 2002. |
| FiF-a 61 | **Fredrik Karlsson:** Meta-Method for Method Configuration - A Rational Unified Process Case, 2002. |
| No 985 | **Sorin Manolache:** Schedulability Analysis of Real-Time Systems with Stochastic Task Execution Times, 2002. |
| No 982 | **Diana Szentiványi:** Performance and Availability Trade-offs in Fault-Tolerant Middleware, 2002. |
| No 989 | **Iakov Nakhimovski:** Modeling and Simulation of Contacting Flexible Bodies in Multibody Systems, 2002. |
| No 990 | **Levon Saldamli:** PDEModelica - Towards a High-Level Language for Modeling with Partial Differential Equations, 2002. |
| No 991 | **Almut Herzog:** Secure Execution Environment for Java Electronic Services, 2002. |

No 999 **Jon Edvardsson:** Contributions to Program- and Specification-based Test Data Generation, 2002.

No 1000 **Anders Arpteg:** Adaptive Semi-structured Information Extraction, 2002.

No 1001 **Andrzej Bednarski:** A Dynamic Programming Approach to Optimal Retargetable Code Generation for Irregular Architectures, 2002.

No 988 **Mattias Arvola:** Good to use! : Use quality of multi-user applications in the home, 2003.

FiF-a 62 **Lennart Ljung:** Utveckling av en projektivitetsmodell - om organisationers förmåga att tillämpa projektarbetsformen, 2003.

No 1003 **Pernilla Qvarfordt:** User experience of spoken feedback in multimodal interaction, 2003.

No 1005 **Alexander Siemers:** Visualization of Dynamic Multibody Simulation With Special Reference to Contacts, 2003.

No 1008 **Jens Gustavsson:** Towards Unanticipated Runtime Software Evolution, 2003.

No 1010 **Calin Curescu:** Adaptive QoS-aware Resource Allocation for Wireless Networks, 2003.

No 1015 **Anna Andersson:** Management Information Systems in Process-oriented Healthcare Organisations, 2003.

No 1018 **Björn Johansson:** Feedforward Control in Dynamic Situations, 2003.

No 1022 **Traian Pop:** Scheduling and Optimisation of Heterogeneous Time/Event-Triggered Distributed Embedded Systems, 2003.

FiF-a 65 **Britt-Marie Johansson:** Kundkommunikation på distans - en studie om kommunikationsmediets betydelse i affärstransaktioner, 2003.

No 1024 **Aleksandra Tešanovic:** Towards Aspectual Component-Based Real-Time System Development, 2003.

No 1034 **Arja Vainio-Larsson**: Designing for Use in a Future Context - Five Case Studies in Retrospect, 2003.

No 1033 **Peter Nilsson:** Svenska bankers redovisningsval vid reservering för befarade kreditförluster - En studie vid införandet av nya redovisningsregler, 2003.

FiF-a 69 **Fredrik Ericsson:** Information Technology for Learning and Acquiring of Work Knowledge, 2003.

No 1049 **Marcus Comstedt:** Towards Fine-Grained Binary Composition through Link Time Weaving, 2003.

No 1052 **Åsa Hedenskog:** Increasing the Automation of Radio Network Control, 2003.

No 1054 **Claudiu Duma:** Security and Efficiency Tradeoffs in Multicast Group Key Management, 2003.

FiF-a 71 **Emma Eliason:** Effektanalys av IT-systems handlingsutrymme, 2003.

No 1055 **Carl Cederberg:** Experiments in Indirect Fault Injection with Open Source and Industrial Software, 2003.

No 1058 **Daniel Karlsson:** Towards Formal Verification in a Component-based Reuse Methodology, 2003.

FiF-a 73 **Anders Hjalmarsson:** Att etablera och vidmakthålla förbättringsverksamhet - behovet av koordination och interaktion vid förändring av systemutvecklingsverksamheter, 2004.

No 1079 **Pontus Johansson:** Design and Development of Recommender Dialogue Systems, 2004.

No 1084 **Charlotte Stoltz:** Calling for Call Centres - A Study of Call Centre Locations in a Swedish Rural Region, 2004.

FiF-a 74 **Björn Johansson:** Deciding on Using Application Service Provision in SMEs, 2004.

No 1094 **Genevieve Gorrell:** Language Modelling and Error Handling in Spoken Dialogue Systems, 2004.

No 1095 **Ulf Johansson:** Rule Extraction - the Key to Accurate and Comprehensible Data Mining Models, 2004.

No 1099 **Sonia Sangari:** Computational Models of Some Communicative Head Movements, 2004.

No 1110 **Hans Nässla:** Intra-Family Information Flow and Prospects for Communication Systems, 2004.

No 1116 **Henrik Sällberg:** On the value of customer loyalty programs - A study of point programs and switching costs, 2004.

FiF-a 77 **Ulf Larsson:** Designarbete i dialog - karaktärisering av interaktionen mellan användare och utvecklare i en systemutvecklingsprocess, 2004.

No 1126 **Andreas Borg:** Contribution to Management and Validation of Non-Functional Requirements, 2004.

No 1127 **Per-Ola Kristensson:** Large Vocabulary Shorthand Writing on Stylus Keyboard, 2004.

No 1132 **Pär-Anders Albinsson:** Interacting with Command and Control Systems: Tools for Operators and Designers, 2004.

No 1130 **Ioan Chisalita:** Safety-Oriented Communication in Mobile Networks for Vehicles, 2004.

No 1138 **Thomas Gustafsson:** Maintaining Data Consistency in Embedded Databases for Vehicular Systems, 2004.

No 1149 **Vaida Jakoniené:** A Study in Integrating Multiple Biological Data Sources, 2005.

No 1156 **Abdil Rashid Mohamed**: High-Level Techniques for Built-In Self-Test Resources Optimization, 2005.

No 1162 **Adrian Pop:** Contributions to Meta-Modeling Tools and Methods, 2005.

No 1165 **Fidel Vascós Palacios:** On the information exchange between physicians and social insurance officers in the sick leave process: an Activity Theoretical perspective, 2005.

FiF-a 84 **Jenny Lagsten:** Verksamhetsutvecklande utvärdering i informationssystemprojekt, 2005.

No 1166 **Emma Larsdotter Nilsson:** Modeling, Simulation, and Visualization of Metabolic Pathways Using Modelica, 2005.

No 1167 **Christina Keller:** Virtual Learning Environments in higher education. A study of students' acceptance of educational technology, 2005.

No 1168 **Cécile Åberg:** Integration of organizational workflows and the Semantic Web, 2005.

FiF-a 85 **Anders Forsman:** Standardisering som grund för informationssamverkan och IT-tjänster - En fallstudie baserad på trafikinformationstjänsten RDS-TMC, 2005.

No 1171 **Yu-Hsing Huang:** A systemic traffic accident model, 2005.

FiF-a 86 **Jan Olausson:** Att modellera uppdrag - grunder för förståelse av processinriktade informationssystem i transaktionsintensiva verksamheter, 2005.

No 1172 **Petter Ahlström**: Affärsstrategier för seniorbostadsmarknaden, 2005.

No 1183 **Mathias Cöster**: Beyond IT and Productivity - How Digitization Transformed the Graphic Industry, 2005.

No 1184 **Åsa Horzella**: Beyond IT and Productivity - Effects of Digitized Information Flows in Grocery Distribution, 2005.

No 1185 **Maria Kollberg**: Beyond IT and Productivity - Effects of Digitized Information Flows in the Logging Industry, 2005.

No 1190 **David Dinka**: Role and Identity - Experience of technology in professional settings, 2005.

No 1191 **Andreas Hansson**: Increasing the Storage Capacity of Recursive Auto-associative Memory by Segmenting Data, 2005.
No 1192 **Nicklas Bergfeldt:** Towards Detached Communication for Robot Cooperation, 2005.
No 1194 **Dennis Maciuszek:** Towards Dependable Virtual Companions for Later Life, 2005.
No 1204 **Beatrice Alenljung:** Decision-making in the Requirements Engineering Process: A Human-centered Approach, 2005.
No 1206 **Anders Larsson:** System-on-Chip Test Scheduling and Test Infrastructure Design, 2005.
No 1207 **John Wilander:** Policy and Implementation Assurance for Software Security, 2005.
No 1209 **Andreas Käll:** Översättningar av en managementmodell - En studie av införandet av Balanced Scorecard i ett landsting, 2005.
No 1225 **He Tan:** Aligning and Merging Biomedical Ontologies, 2006.
No 1228 **Artur Wilk:** Descriptive Types for XML Query Language Xcerpt, 2006.
No 1229 **Per Olof Pettersson:** Sampling-based Path Planning for an Autonomous Helicopter, 2006.
No 1231 **Kalle Burbeck:** Adaptive Real-time Anomaly Detection for Safeguarding Critical Networks, 2006.
No 1233 **Daniela Mihailescu:** Implementation Methodology in Action: A Study of an Enterprise Systems Implementation Methodology, 2006.
No 1244 **Jörgen Skågeby:** Public and Non-public gifting on the Internet, 2006.
No 1248 **Karolina Eliasson:** The Use of Case-Based Reasoning in a Human-Robot Dialog System, 2006.
No 1263 **Misook Park-Westman:** Managing Competence Development Programs in a Cross-Cultural Organisation - What are the Barriers and Enablers, 2006.
FiF-a 90 **Amra Halilovic:** Ett praktikperspektiv på hantering av mjukvarukomponenter, 2006.
No 1272 **Raquel Flodström**: A Framework for the Strategic Management of Information Technology, 2006.
No 1277 **Viacheslav Izosimov:** Scheduling and Optimization of Fault-Tolerant Embedded Systems, 2006.
No 1283 **Håkan Hasewinkel:** A Blueprint for Using Commercial Games off the Shelf in Defence Training, Education and Research Simulations, 2006.
FiF-a 91 **Hanna Broberg:** Verksamhetsanpassade IT-stöd - Designteori och metod, 2006.
No 1286 **Robert Kaminski:** Towards an XML Document Restructuring Framework, 2006.
No 1293 **Jiri Trnka:** Prerequisites for data sharing in emergency management, 2007.
No 1302 **Björn Hägglund:** A Framework for Designing Constraint Stores, 2007.
No 1303 **Daniel Andreasson**: Slack-Time Aware Dynamic Routing Schemes for On-Chip Networks, 2007.
No 1305 **Magnus Ingmarsson:** Modelling User Tasks and Intentions for Service Discovery in Ubiquitous Computing, 2007.
No 1306 **Gustaf Svedjemo**: Ontology as Conceptual Schema when Modelling Historical Maps for Database Storage, 2007.
No 1307 **Gianpaolo Conte**: Navigation Functionalities for an Autonomous UAV Helicopter, 2007.
No 1309 **Ola Leifler**: User-Centric Critiquing in Command and Control: The DKExpert and ComPlan Approaches, 2007.
No 1312 **Henrik Svensson**: Embodied simulation as off-line representation, 2007.
No 1313 **Zhiyuan He:** System-on-Chip Test Scheduling with Defect-Probability and Temperature Considerations, 2007.
No 1317 **Jonas Elmqvist:** Components, Safety Interfaces and Compositional Analysis, 2007.
No 1320 **Håkan Sundblad:** Question Classification in Question Answering Systems, 2007.
No 1323 **Magnus Lundqvist**: Information Demand and Use: Improving Information Flow within Small-scale Business Contexts, 2007.
No 1329 **Martin Magnusson:** Deductive Planning and Composite Actions in Temporal Action Logic, 2007.
No 1331 **Mikael Asplund:** Restoring Consistency after Network Partitions, 2007.
No 1332 **Martin Fransson:** Towards Individualized Drug Dosage - General Methods and Case Studies, 2007.
No 1333 **Karin Camara:** A Visual Query Language Served by a Multi-sensor Environment, 2007.
No 1337 **David Broman:** Safety, Security, and Semantic Aspects of Equation-Based Object-Oriented Languages and Environments, 2007.
No 1339 **Mikhail Chalabine:** Invasive Interactive Parallelization, 2007.
No 1351 **Susanna Nilsson:** A Holistic Approach to Usability Evaluations of Mixed Reality Systems, 2008.
No 1353 **Shanai Ardi:** A Model and Implementation of a Security Plug-in for the Software Life Cycle, 2008.
No 1356 **Erik Kuiper:** Mobility and Routing in a Delay-tolerant Network of Unmanned Aerial Vehicles, 2008.
No 1359 **Jana Rambusch**: Situated Play, 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.
No 1371 **Fredrik Lantz:** Terrain Object Recognition and Context Fusion for Decision Support, 2008.
No 1373 **Martin Östlund:** Assistance Plus: 3D-mediated Advice-giving on Pharmaceutical Products, 2008.
No 1381 **Håkan Lundvall:** Automatic Parallelization using Pipelining for Equation-Based Simulation Languages, 2008.
No 1386 **Mirko Thorstensson:** Using Observers for Model Based Data Collection in Distributed Tactical Operations, 2008.
No 1387 **Bahlol Rahimi:** Implementation of Health Information Systems, 2008.
No 1392 **Maria Holmqvist:** Word Alignment by Re-using Parallel Phrases, 2008.
No 1393 **Mattias Eriksson**: Integrated Software Pipelining, 2009.
No 1401 **Annika Öhgren:** Towards an Ontology Development Methodology for Small and Medium-sized Enterprises, 2009.
No 1410 **Rickard Holsmark:** Deadlock Free Routing in Mesh Networks on Chip with Regions, 2009.
No 1421 **Sara Stymne:** Compound Processing for Phrase-Based Statistical Machine Translation, 2009.
No 1427 **Tommy Ellqvist**: Supporting Scientific Collaboration through Workflows and Provenance, 2009.
No 1450 **Fabian Segelström:** Visualisations in Service Design, 2010.
No 1459 **Min Bao**: System Level Techniques for Temperature-Aware Energy Optimization, 2010.
No 1466 **Mohammad Saifullah**: Exploring Biologically Inspired Interactive Networks for Object Recognition, 2011

| | |
|---|---|
| No 1468 | **Qiang Liu**: Dealing with Missing Mappings and Structure in a Network of Ontologies, 2011. |
| No 1469 | **Ruxandra Pop**: Mapping Concurrent Applications to Multiprocessor Systems with Multithreaded Processors and Network on Chip-Based Interconnections, 2011. |
| No 1476 | **Per-Magnus Olsson**: Positioning Algorithms for Surveillance Using Unmanned Aerial Vehicles, 2011. |
| No 1481 | **Anna Vapen**: Contributions to Web Authentication for Untrusted Computers, 2011. |
| No 1485 | **Loove Broms:** Sustainable Interactions: Studies in the Design of Energy Awareness Artefacts, 2011. |
| FiF-a 101 | **Johan Blomkvist:** Conceptualising Prototypes in Service Design, 2011. |
| No 1490 | **Håkan Warnquist:** Computer-Assisted Troubleshooting for Efficient Off-board Diagnosis, 2011. |
| No 1503 | **Jakob Rosén:** Predictable Real-Time Applications on Multiprocessor Systems-on-Chip, 2011. |
| No 1504 | **Usman Dastgeer:** Skeleton Programming for Heterogeneous GPU-based Systems, 2011. |
| No 1506 | **David Landén:** Complex Task Allocation for Delegation: From Theory to Practice, 2011. |
| No 1507 | **Kristian Stavåker**: Contributions to Parallel Simulation of Equation-Based Models on Graphics Processing Units, 2011. |
| No 1509 | **Mariusz Wzorek:** Selected Aspects of Navigation and Path Planning in Unmanned Aircraft Systems, 2011. |
| No 1510 | **Piotr Rudol:** Increasing Autonomy of Unmanned Aircraft Systems Through the Use of Imaging Sensors, 2011. |
| No 1513 | **Anders Carstensen:** The Evolution of the Connector View Concept: Enterprise Models for Interoperability Solutions in the Extended Enterprise, 2011. |
| No 1523 | **Jody Foo:** Computational Terminology: Exploring Bilingual and Monolingual Term Extraction, 2012. |
| No 1550 | **Anders Fröberg:** Models and Tools for Distributed User Interface Development, 2012. |
| No 1558 | **Dimitar Nikolov:** Optimizing Fault Tolerance for Real-Time Systems, 2012. |
| No 1582 | **Dennis Andersson:** Mission Experience: How to Model and Capture it to Enable Vicarious Learning, 2013. |
| No 1586 | **Massimiliano Raciti:** Anomaly Detection and its Adaptation: Studies on Cyber-physical Systems, 2013. |
| No 1588 | **Banafsheh Khademhosseinieh:** Towards an Approach for Efficiency Evaluation of Enterprise Modeling Methods, 2013. |
| No 1589 | **Amy Rankin:** Resilience in High Risk Work: Analysing Adaptive Performance, 2013. |
| No 1592 | **Martin Sjölund:** Tools for Understanding, Debugging, and Simulation Performance Improvement of Equation-Based Models, 2013. |
| No 1606 | **Karl Hammar:** Towards an Ontology Design Pattern Quality Model, 2013. |
| No 1624 | **Maria Vasilevskaya:** Designing Security-enhanced Embedded Systems: Bridging Two Islands of Expertise, 2013. |
| No 1627 | **Ekhiotz Vergara:** Exploiting Energy Awareness in Mobile Communication, 2013. |
| No 1644 | **Valentina Ivanova:** Integration of Ontology Alignment and Ontology Debugging for Taxonomy Networks, 2014. |
| No 1647 | **Dag Sonntag:** A Study of Chain Graph Interpretations, 2014. |
| No 1657 | **Kiril Kiryazov:** Grounding Emotion Appraisal in Autonomous Humanoids, 2014. |
| No 1683 | **Zlatan Dragisic:** Completing the Is-a Structure in Description Logics Ontologies, 2014. |
| No 1688 | **Erik Hansson:** Code Generation and Global Optimization Techniques for a Reconfigurable PRAM-NUMA Multicore Architecture, 2014. |
| No 1715 | **Nicolas Melot:** Energy-Efficient Computing over Streams with Massively Parallel Architectures, 2015. |
| No 1716 | **Mahder Gebremedhin:** Automatic and Explicit Parallelization Approaches for Mathematical Simulation Models, 2015. |