Linköping Studies in Science and Technology
Dissertation No. 1022

Automatic Parallelization of Equation-Based Simulation Programs

Peter Aronsson

Linköping University
INSTITUTE OF TECHNOLOGY

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

Linköping 2006

ISBN 91-85523-68-2
ISSN 0345-7524

Printed by LiU-Tryck, Linköping 2006

Acknowledgement

I am grateful for all the help, guidance, and the many lessons learned during the making of this thesis. First of all, I would like to thank my supervisor, Peter Fritzson, who made this work possible and has supported me in many ways. I would also like to thank my co-supervisor Christoph Kessler for his support.

I've enjoyed the stay at PELAB during this years and especially all the interesting discussions during our coffee breaks, so many thanks also goes to all colleagues at PELAB. I also want to specially thank Bodil for her support in administrative tasks and for making life easier for the rest of us at PELAB.

Many thanks goes to all the employees at MathCore who have provided a nice working atmosphere. I have also received much help and assistance from them, especially for problems in modeling and simulation.

I am also grateful that I have had the opportunity to use the parallel computers at NSC. Without these, the research would not have been possible.

Finally, I would like to thank my big love in life, Marie, for always being there for me and supporting me in all ways possible.

Peter Aronsson
Linköping, March 2006

Automatic Parallelization of Equation-Based Simulation Programs

by

Peter Aronsson

March 2006
ISBN 91-85523-68-2
Linköping Studies in Science and Technology
Thesis No. 1022
ISSN 0345-7524

ABSTRACT

Modern equation-based object-oriented modeling languages which have emerged during the past decades make it easier to build models of large and complex systems. The increasing size and complexity of modeled systems requires high performance execution of the simulation code derived from such models. More efficient compilation and code optimization techniques can help to some extent. However, a number of heavy-duty simulation applications require the use of high performance parallel computers in order to obtain acceptable execution times. Unfortunately, the possible additional performance offered by parallel computer architectures requires the simulation program to be expressed in a way that makes the potential parallelism accessible to the parallel computer. Manual parallelization of computer programs is generally a tedious and error prone process. Therefore, it would be very attractive to achieve automatic parallelization of simulation programs.

This thesis presents solutions to the research problem of finding practically usable methods for automatic parallelization of simulation codes produced from models in typical equation-based object-oriented languages. The methods have been implemented in a tool to automatically translate models in the Modelica modeling language to parallel codes which can be efficiently executed on parallel computers. The tool has been evaluated on several application models. The research problem includes the problem of how to extract a sufficient amount of parallelism from equations represented in the form of a data dependency graph (task graph), requiring analysis of the code at a level as detailed as individual expressions. Moreover, efficient clustering algorithms for building clusters of tasks from the task graph are also required. One of the major contributions of this thesis work is a new approach for merging fine-grained tasks by using a graph rewrite system. Results from using this method show that it is efficient in merging task graphs, thereby decreasing

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

their size, while still retaining a reasonable amount of parallelism. Moreover, the new task-merging approach is generally applicable to programs which can be represented as static (or almost static) task graphs, not only to code from equation-based models.

An early prototype called DSBPart was developed to perform parallelization of codes produced by the Dymola tool. The final research prototype is the ModPar tool which is part of the OpenModelica framework. Results from using the DSBPart and ModPar tools show that the amount of parallelism of complex models varies substantially between different application models, and in some cases can produce reasonable speedups. Also, different optimization techniques used on the system of equations from a model affect the amount of parallelism of the model and thus influence how much is gained by parallelization.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter we introduce the thesis problem and its research areas. We present thesis work contributions and give an introductory background.

With modern equation-based object-oriented modeling languages it is becoming easier to build large and complex models. With this increasing complexity it is crucial to use all possible ways of speeding up simulation execution time. In this thesis, we solve the problem of automatic parallelization of these simulation codes. This research problem contains several issues such as clustering, scheduling and identification of parallelism in the code produced from the object-oriented equation-based modeling language Modelica.

Most contributions in this thesis work are in the area of clustering and scheduling for multiprocessors and one of the major contributions is a new method of clustering or merging tasks to create a task (data dependence) graph, based on simple rules that define a graph rewrite system.

Throughout this work, several contributions in the form of prototype implementations have been made. The earliest prototype parallelized simulation code from a commercial tool, Dymola, later replaced by a prototype for the OpenModelica compiler. Additionally, a prototype for clustering and scheduling algorithms was developed in Mathematica.

1.1 Outline

This thesis is outlined as follows.

Chapter two presents automatic parallelization starting with task graphs, i.e. a fine-grained graph representation of computer programs used for the analysis. It also presents parallel programming models, scheduling and clustering, and task merging algorithms to parallelize programs.

Chapter three presents modeling and simulation using mathematical equations. This chapter presents the Modelica modeling language, for which the prototype tools are developed.

Chapter four presents a prototyping framework called DAGS, which is a software package for writing algorithms related to task graphs, such as scheduling and clustering algorithms.

Chapter five presents the first prototype parallelization tool built which translated C-code from the commercial Modelica tool Dymola to parallel C-code. It also presents some discussion and conclusions on why the second prototype was built.

Chapter six presents the prototype parallelization tool called ModPar, which is an integrated part of the OpenModelica compiler.

Chapter seven presents a recently developed task merging technique, one of the major contributions of the thesis work.

Chapter eight presents several application examples using the Modelica modeling language on which the automatic parallelization tool has been tested.

Chapter nine presents the thesis work results.

Chapter ten presents related work in different areas, e.g. parallel simulation, automatic parallelization, task clustering and merging, etc.

Chapter eleven presents the future directions of research in different areas.

Finally, chapter twelve presents the conclusions drawn from the hypothesis and the results.

1.2 The Need for Modeling and Simulation

Modeling and simulation tools are becoming a powerful aid in the industrial product development process. By building a computer-based model of the product using advanced tools and languages, and simulating its behavior prior to producing a physical prototype, errors in the design or in production can be detected at an early stage in the development process, leading to shorter development time, since the earlier an error is detected, the cheaper it is to correct.

Modeling and simulation is also a powerful way of increasing the knowledge and understanding of complex physical systems, often involving hundreds of components, ranging from mechanical parts, electrical circuits, hydraulic fluids, chemical reactions, etc. By building and simulating mathematical models of such a large and complex system, the system can be better understood, its design flaws detected and corrected and the system can be optimized according to different criterias.

Until quite recently in the history of modeling and simulation technology, mathematical models were built completely by hand. The equations and

formulas describing the physical behavior of a system described by a model were written by hand and manually transformed and simplified so that an executable implementation of the model could be written in an ordinary programming language such as Fortran or C. Since most of this work was done manually, it was expensive to maintain and change mathematical models in order to adapt them to new requirements. In fact, this process of manual model development is still in use today, but is gradually being replaced by the use of more automatic tools.

In this manual approach in use today, the knowledge of the models is typically divided between different persons possibly at different places. Some people are responsible for the physical behavior of the model with knowledge about the equations and variables of the model, others have the technology and knowledge on how the model equations are implemented in the programming language used for the simulation of the model. This scattering of knowledge and technology renders maintenance expensive and reuse of models very difficult.

In addition to manual implementation in a programming language there are also graphical composition tools based on manually implemented model components, such as Simulink [44]. These tools have the underlying causality of the model components fixed, for instance always calculating the torque and angular velocity of an electrical motor given its source voltage as input. These limitations make the model components less reusable since changing the causality, i.e., what is input and what is output, forces users to totally rewrite (or redraw) their models.

To remedy these problems, object oriented equation-based modeling languages such as Modelica [48] have been developed. By using an object oriented modeling language it is possible to describe the physical behavior of individual objects e.g. by using Modelica classes, corresponding to models of real physical objects. Modelica classes can then be instantiated as objects inside so called composite objects and connected together to form larger models. By modeling each physical object as an entity (or object) combined with the possibility of reusing objects through inheritance, a true object oriented modeling approach is obtained.

Moreover, if the modeling language is equation-based, the reuse opportunities increase even further. By describing the physical behavior of an object with equations, the causality (i.e., the direction of the "data flow") through the object is left unspecified. This makes it possible to use the component both in input and in output contexts. For instance, an electrical motor can both be used as a traditional motor giving rotational energy from an electrical current or as a generator transforming rotational energy into electrical energy. The causality can be left to the tool to find out, based on the computation

needs of the user.

1.2.1 Building Models of Systems

A system can be many things. Websters dictionary defines a *system* as "a regularly interacting or interdependent group of items forming a unified whole". In computer science, a system can also be defined in several different ways. However, in the context of this thesis, a system can be seen as an entity taking some input, having an internal state, and producing some output. Traditionally, in for instance control theory, a system is depicted as in Figure 1.1.

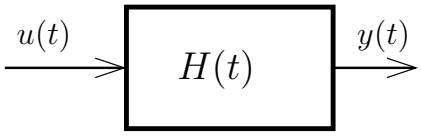


Figure 1.1. A control theory point of view of a system as a function $H(t)$ reacting on an input $u(t)$ and producing an output $y(t)$.

The modeling process starts by choosing a formalism and granularity or level-of-detail for the model. This means that the level of detail in the model must be chosen by the modeler. For instance should a model of a tank system be made using discrete-time variables, using for instance queuing theory for describing handling of discrete packets of material of fluid, or by using continuous-time variables, using differential equations. The modeler also needs to choose the correct approximation of the model. For instance, should an electrical resistor be modeled using a temperature dependent resistance or should it be made temperature independent, etc. These choices are different for each usage of the model, and should preferably be changed for each individual case in an easy manner.

All these design choices determine the complexity and accuracy of the model compared to the real world. These choices also indirectly influence the execution time of the simulations, a more detailed model will usually take longer time to simulate.

As an example, let us consider a resistor which for many applications can be modeled using Ohms law:

$$u = R\,i \tag{1.1}$$

where u is the voltage drop over the resistor and i is the current through the resistor, proportional to the voltage by the resistance, R.

However, in some cases a more advanced and accurate model might be needed, for instance taking a possible temperature dependence of the resistance into consideration [33]:

$$u = R_0(1 + \alpha(T - 20))i \tag{1.2}$$

where T is the temperature and α is the temperature coefficient.

Equation 1.2 involves both more variables and a more expensive computation compared to Equation 1.1. For this simple case the two models are not dramatically different, but in other more complicated cases a more accurate model could mean replacing a linear equation with a non-linear, resulting in a substantial increase in computational cost for simulating the model.

1.2.2 Simulation of Models

Once the system has been modeled, it can be simulated to produce time varying variable values in the model and typically produce a visualization of the values as plots. These plots are then used to draw conclusions about the system, e.g. extrapolations of the model or to verify it against measured data.

For instance, if we consider the equation for a mass connected to a spring and a damper, as depicted in Figure 1.2, it has the following form:

$$m\ddot{x} + k\dot{x} + cx = 0 \tag{1.3}$$

where x is the position of the body, m is its mass and c and k are the spring and damping constants, respectively.

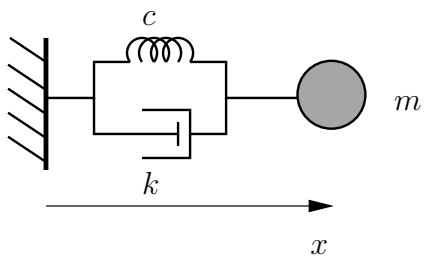


Figure 1.2. A body mass connected to a fixed frame with a spring and a damper.

A simulation of this small system together with a starting condition of $x(0) = 1$ will result in a time varying dataset plotted in Figure 1.3. The

system has been simulated for 20 seconds, and shows a damped oscillation of the position of the body.

From this plot we can e.g. draw the conclusion that the chosen damping coefficient is sufficiently large to reduce the oscillation to 10 percent within approximately 20 seconds.

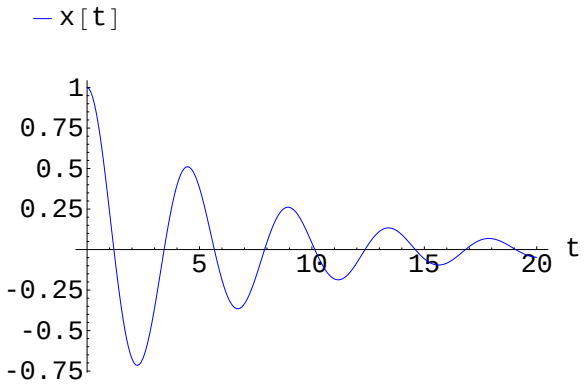


Figure 1.3. The plot of the position of the body resulted from the simulation.

1.3 Introduction to Parallel Computing

Parallel computing is concerned with execution of computer programs in parallel on computers with multiple processing units. The goal is to execute the program faster compared to executing the same program on a single computer or processing unit. For this to be achieved, several issues must be considered.

- Efficient parallel architectures are essential to build efficient parallel computers with high speed communication between different processors or functional units. These are often realized by a combination of hardware and software.
- Parallel programming languages and application packages are needed to be able to construct computer programs for these parallel machines.
- Automated tools that construct parallel programs from different kinds of applications help users in the parallel programming task, rendering

it easier to focus on their main problem instead of performing manual parallelization.

The following sections give short introductions to these three areas.

1.3.1 Parallel Architectures

Parallel computing involves execution of computer programs on computers with a parallel architecture. Parallel architectures range from a processor with several functional units that can be executed independently in parallel up to a multiprocessor computer with thousands of processors communicating through an interconnection network.

Typically a parallel computer is a multiprocessor computer with some way of communication between different processors. There are basically two kinds of parallel computer architectures. A parallel computer can consist of N processors communicating through a *shared memory*, a so called *shared memory architecture* as depicted in Figure 1.4. In the figure each processor has a cache to speed up memory accesses to the shared memory bank. An example of such architecture is the SGI Origin 3800 computer at NSC (National Super Computer Center) [53] which has 128 processors that communicate through a 128 GB shared memory, connected to a shared bus.

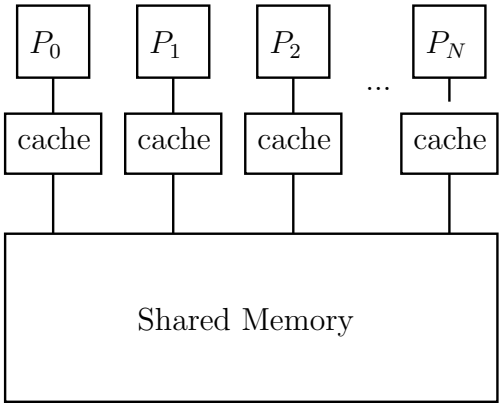


Figure 1.4. A shared memory architecture.

The second major kind of parallel architecture is a *distributed memory architecture*. They have a *distributed memory*, typically divided into one local memory for each processor, and communicates through a communication network. Linux cluster computers fall into this category, like for instance the

Monolith cluster at NSC [50], having 200 PC computers connected through a high speed SCI interconnection network [73]. The Monolith cluster computer is actually a combination of distributed and a shared memory architecture since each computer node contains two processors, sharing the common memory of the computer node. Figure 1.5 shows a distributed memory architecture.

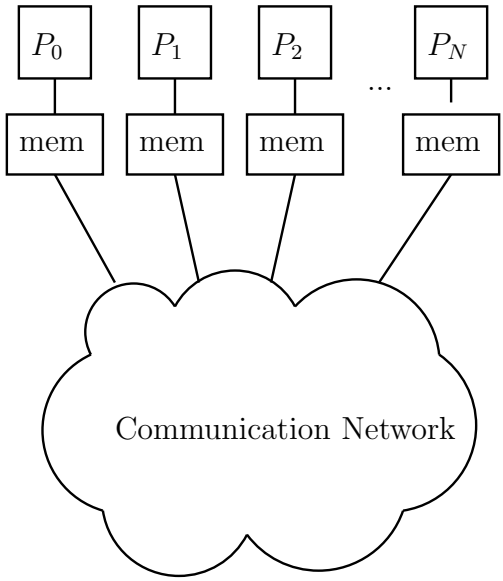


Figure 1.5. A distributed memory architecture.

Writing computer programs for these two architectures can be done using two different programming models, as we shall see in the next section.

1.3.2 Parallel Programming Languages and Tools

Writing computer programs for parallel machines is very complex. The programmer needs to consider issues like dividing the computational work into parallel tasks and their dependencies, distribution of the data to the processors, communication between tasks on different processors, etc. For a shared memory architecture the explicit sending of data need not be programmed but there are other problems instead. For instance, what happens if several processors are writing to the same memory location? Some of these programming tasks are greatly simplified by appropriate support in programming languages and tools.

There are several programming languages and extensions to programming languages for supporting parallel programming. For instance, when writing a parallel program for a distributed memory machine one has to explicitly insert send and receive instructions to communicate the data from one processor to another. MPI (Message Passing Interface) [45] is a wide-spread standardized API for such tasks with both commercial high quality implementations [71] and open source implementations [85]. Another similar message passing library is PVM [64], which basically provides the same functionality as MPI but has some additional support for dynamically adding processors.

Below is a small MPI program for execution on two processors. The program reads data from a file that is processed element-wise in some manner, thus it can be parallelized easily by dividing the data between several processors and executing the data processing in parallel. The first two calls (`MPI_Init` and `MPI_Comm_Rank`) initialize the MPI system and sets the `rank` variable to the processor number. Each processor will execute the same program with a different rank integer value. Thus, the next part of the code makes a conditional selection based on this processor number. Processor 0 reads some data from a file and then sends half of this data to processor 1. The `MPI_Send` function takes

- a pointer to the data
- the size of the data
- the type of data, which must be a type declared in MPI, e.g. `MPI_REAL` or `MPI_INTEGER`
- the destination process
- a message tag, to be able to distinguish between messages.
- a processor group, in this case the default group of all processors, `MPI_COMM_WORLD`

Similarly, the `MPI_Send` call must be received on the other processor using the `MPI_Recv` function. It has similar arguments and an additional argument for setting status information on the received data.

```
int main( int argc, char**argv)
{
 int tag1=1,tag2=2; // Two separate tags:
                    // tag1 - message from P1 -> P0
                    // tag2 - message from P0 -> P1
 int rank;
```

```
 int *data, size;
 MPI_Status status;
 MPI_Init(&argc,&argv);
 MPI_Comm_rank(MPI_COMM_WORLD,&rank);
 if (rank == 0) { //Proc 0
   size = read_data(data); // Read data from file
   MPI_Send(data,size/2,MPI_REAL,1,  // Send to 1
            tag1,MPI_COMM_WORLD);
   process_data(data+size/2);
   MPI_Recv(data,size/2,MPI_REAL,1,  // Recv from 1
            tag2,MPI_COMM_WORLD,&status);
   save_data(data,size); // Save the data to file
  } else if (rank == 1) { // Proc 1
   MPI_Recv(data,size/2,MPI_REAL,0, // Recv from 0
            tag1,MPI_COMM_WORLD,&status);
   process_data(data);
   MPI_Send(data,size/2,MPI_REAL,0, //  Send to 0
            tag2,MPI_COMM_WORLD);
  }
 MPI_Finalize();
 return 0;
}
```

When programming using MPI (or PVM or similar) the programmer must explicitly insert all the send and receive commands for all processes. This is an error prone and cumbersome task, which can often lead to deadlocks, missing receive or send calls, etc. Hence, writing parallel programs using a distributed memory programming model with message passing is an advanced programming task which requires an experienced programmer. However, parallel programming mistakes are common even among experienced programmers.

When considering shared memory architectures one can use programming language extensions such as OpenMP [54], where Fortran or C++ code can be annotated with instructions how to execute loop iterations as well as other constructs in parallel. An alternative is to use a thread library and program your parallel application using threads and semaphores, for instance using Posix threads [76].

Thread programming also requires advanced and experienced programmers to avoid deadlocks and other thread programming specific pitfalls. The same goes for programming using OpenMP e.g. when declaring shared memory variables. The programmer must himself guarantee that different processes do not conflict when accessing these variables (depending on what the architecture

allows). For example, below is a C program with an OpenMP directive stating that the for-loop can be executed in parallel. By issuing this compiler pragma the programmer guarantees that there are no dependencies across the iterations of the loop.

```
int main()
{
  int i;
  double Res[1000];
#pragma omp parallel for
  for (i=0 ; i<1000; i++) {
    huge_comp(Res[i]);
 }
}
```

There are also tools for automatic parallelization of program code. For instance, most Fortran compilers have special flags for automatically parallelizing the program. The compiler will then analyze loops to detect when they can be executed in parallel, and generate parallel code for such loops.

For instance, lets consider a program written in Fortran that performs an element-wise multiplication of two vectors.

```
do i=1,N
  v[i]=a[i]*b[i];
end do
```

This piece of program code is quite trivial to parallelize. The automatic parallelization will split the loop and schedule execution of part of the loop iterations on each processor. This can be performed since there are no data dependencies between different iterations of the loop, making it possible for the different iterations of the loop to be executed independently of each other. For more complex loops a data dependency analysis must be performed to investigate which iterations of the loop can be executed in parallel. Many techniques have been developed to transform loops and data layouts to increase the amount of parallelism in such loops [46, 14].

1.3.3 Measuring Parallel Performance

Once a parallel program has been constructed, either using automatic parallelization tools or by manual programming, it is executed on a parallel machine. However, we are also usually interested obtaining information about the efficiency of the parallel execution.

The execution time of the parallel program is not a suitable metric to measure the efficiency of parallel programs in an independent way. Instead the term *relative* speedup is used, defined as [25]:

$$S_{relative} = \frac{T_1}{T_N} \tag{1.4}$$

where:

- T_1 is the execution time for running the parallel program on one processor and
- T_N is the execution time for running the parallel program on N processors.

This speedup is called relative because the same program is used for measuring both the sequential and parallel time. However, there might also exist a more efficient sequential implementation of the program. Therefore, there is also a definition of *absolute* speedup where the sequential execution time is measured on the most efficient sequential implementation instead of using the same parallel program also for the one processor case. The definition of absolute speedup is thus:

$$S_{absolute} = \frac{T_{seq}}{T_N} \tag{1.5}$$

where:

- T_{seq} is the execution time of the most effective sequential implementation.
- T_N is the execution time of the parallel program for N processors as above.

Since a sequential version of the simulation code exist for all models targeted by the tool presented in this work, e.g. the code produced by the OpenModelica compiler, the speedup definition used throughout the rest of this thesis can be viewed as the absolute speedup, even though the OpenModelica compiler might not produce the most effective sequential code.

Another commonly used term in parallel computing for measuring parallel performance is the *efficiency* of a parallel program. The relative efficiency is defined as:

$$E_{relative} = S_{relative}/P \tag{1.6}$$

and the corresponding absolute efficiency is similarly defined as:

$$E_{absolute} = S_{absolute}/P \tag{1.7}$$

The efficiency of a parallel program indicates how much useful work is performed by the processors. Ideally, for linear speedup the efficiency is one, normally however, it is below one. For instance, if each processor in the execution of a parallel program spends 10 percent of its time communicating data instead of performing computational work, the efficiency of the parallel program becomes 0.9.

Another observation regarding the performance of parallel programs is *Amdahl's law*. It states that a parallel program that has a constant fraction $1/s$ of its work executed by a sequential piece of program code will have a maximum possible speedup of s. For instance, if a parallel program has a sequential part taking 10 percent of the one-processor execution time, the maximum speedup is 10. This law is especially important in the context of this thesis work since the parallelization of simulation code as performed here will leave an un-parallelized sequential part of the parallel program.

1.3.4 Parallel Simulation

Efficient simulation is becoming more important as the modeled systems increase in size and complexity. By using an object-oriented component based modeling language such as Modelica, it is possible to model large and complex systems with reasonably little effort. Even an inexperienced user with no detailed modeling knowledge can build large and complex models by connecting components from already developed Modelica packages, such as the Modelica Standard Library or commercial packages from various vendors. Therefore, the number of equations and variables of such models tend to grow since it is easier to build large simulation models when using object-oriented component based languages such as Modelica. Thus, to increase the size of problems that can be efficiently simulated within a reasonable time it is necessary to exploit all possible ways of reducing simulation execution time.

Parallelism in simulation can be categorized in three groups:

- **Parallelism over the method**
  One approach is to adapt the numerical solver for parallel computation, i.e., to exploit parallelism over the numerical method. For instance, by using a Runge-Kutta method in the numerical solver some degree of parallelism can be exploited within the numerical solver [67]. Since the Runge-Kutta methods can involve calculations of several time steps simultaneously, parallelism is easy to extract by letting each time step calculation be performed in parallel. This approach will typically give a limited speedup in the range 3-4, depending on the choice of solver.

- **Parallelism over time**
  A second alternative is to parallelize a simulation over time. This ap-

proach is however best suited for discrete event simulations and less suitable for simulation of continuous systems, since the solutions to continuous time dependent equation systems develop sequentially over time, where each new solution step is dependent on the immediately preceding steps.

- **Parallelism over the system**
  The approach taken in this work is to parallelize over the system, which means that the calculation of the modeled system (the model equations) are parallelized. For an ODE (or a DAE) this means parallelizing the calculation of the states, i.e., the functions f and g (see Equation 3.5 and 3.6). It can also mean for an DAE to calculate the Jacobian (i.e., the partial derivatives) of the model equations, since this is required by several DAE solvers.

The simulation code consists of two separate parts, a numerical solver and the code that computes new values of the state variables, i.e., calculating f in the ODE case or solving g in the DAE case. The numerical solver is usually a standard numerical solver for solving ODE or DAE equation systems. For each integration step, the solver needs the values of the derivatives of each state variable (and the state variable values as well as some of the algebraic variables in the DAE case) for calculation of the next step. The solver is naturally sequential and therefore in general not possible to parallelize. However, the largest part of the simulation execution time is typically used for the calculation of f and g. Therefore, we focus on parallelizing the computation of these parts. This approach has for example successfully been used in [2, 26].

When the simulation code has been parallelized, timing measurements on the execution time of the simulation code are performed.

1.4 Automatic Parallelization

By automatic parallelization we mean the process of automatically translating a program into a parallel program to be executed on a parallel computer. The first step in this process, referred to as parallelization, is to translate the source program into a data dependence graph (or task graph). The data dependence graph consists of nodes that represent computational tasks of the program and edges representing the data sent between tasks. In this research a fined grained task graph is built from individual scalar expressions of equations from the simulation code, or from larger tasks such as solving a linear or non-linear system of equations. This step is quite straight forward and do not require any deeper analysis.

The next step of the process is task scheduling for multi processors. This process maps the task graph onto N processors by giving each task a starting time and processor assignment. This step can also include building clusters or merging of tasks to simplify the scheduling process. In this research, clustering or merging of the task graph prior to multi processor scheduling is essential to overcome the otherwise poor performance of scheduling of the kind of fine grained task graphs produced. This is also where the main contribution of the thesis is made.

The final step in the automatic parallelization process is to generate the parallel program from the scheduled task graph. This includes creating code for the computation of the tasks and inserting code for sending of messages between tasks that are allocated to different processors, as given by the edges of the task graph. In this work, C-programs with MPI-calls as communication interface is produced by the parallelization tool.

1.5 Research Problem

The research problem of this thesis work is to parallelize simulation code in equation-based modeling and simulation languages such as Modelica. The parallelization approach taken is to parallelize over the system, i.e., parallelizing the equations given by the Modelica model.

The problem can be summarized by the following hypothesis:

Hypothesis 1
It is possible to build an automatic parallelization tool that translates automatically generated simulation code from equation-based simulation languages into a platform independent parallel version of the simulation code that can be executed more efficiently on a parallel rather than sequential computer.

Hypothesis 1 says that from an equation-based modeling language, such as Modelica, it is possible to automatically parallelize the code and obtain speedups on parallel computers. The tool should be efficient enough to ensure that producing the parallel code is possible within reasonable time limits. The parallel code should also be efficient, i.e., the parallel program should run substantially faster compared to the sequential simulation code. Finally, the parallel code should be platform independent so that it can easily be executed on a variety of different parallel architectures.

The following sections split the research problem stated in Hypothesis 1 into three subproblems.

Parallelism in Model Equations

The most important problem that needs to be solved to verify the hypothesis is how parallelism can be extracted from the simulation code, i.e., the model equations. Earlier work investigated the extent of parallelism in simulation code at three different levels [2] for an equation-based modeling language called ObjectMath [83, 47].

The highest level where parallelism can be extracted is at component level. Each component of a large complex system usually contains many equations. The computational work for each component can potentially be put on one processor per component, with communication of the connector variables in between processors. However, earlier work [2] has demonstrated that in the general case there is typically not enough parallelism at this level.

The middle level is to extract parallelism at the equation level, i.e., each equation is considered as a unit. This approach produces better parallelism compared to extracting parallelism at the component level, but the degree of parallelism is in general not sufficient [2].

The third level is to go down to the sub-equation level, where we consider parallelism between parts of equations like for instance arithmetic operations. At this level, the greatest degree of parallelism was found among the three levels [2].

However, compilation techniques for optimization of the code generated from equation systems has improved since earlier work (ObjectMath [83]), resulting in a more optimized sequential code. Therefore, parallelism is harder to extract in this case compared to previous work. Thus, the research problem of extracting parallelism from simulation code generated from highly optimized model equations still remains to be solved. Also, even if parallelism is extracted, the problem of clustering (a part of the multiprocessor scheduling problem) has become even more important for obtaining speedups, since processor speed has increased more than communication speed during recent years. The clustering problem is one of the key issues in the research problem presented in this thesis.

Clustering and Scheduling Algorithms

Clustering and scheduling algorithms are two important parts of the solution to the multiprocessor scheduling problem which is at the core of any automatic parallelization tool. Clustering algorithms deal with creating clusters of tasks designated to execute on the same processor, while scheduling algorithm assigns tasks to processors.

The results, i.e the speedups achieved in earlier work in automatic parallelization of ObjectMath models [2] were not good enough, due to bad clus-

tering techniques. Therefore, the research problem of performing an efficient clustering of such simulation code still also remains unsolved. The scheduling (including clustering) of task graphs for parallel machines has been studied extensively in this and other work. Efficient algorithms with a low complexity should be used in order to fulfill Hypothesis 1. Task replication has to be used to better exploit the sometimes low amount of parallelism that can be extracted from the simulation code at the sub-equation level, i.e., looking at expressions and statements in the generated C-code.

Note that it is of substantial practical importance that the scheduling algorithms used have a low time complexity, so that a parallel program can be generated within reasonable time.

Cost Estimation

Another research problem is to estimate the cost of each task in the task graph built internally by the parallelization tool, see Section 6.4.1. The costs of some tasks can be determined with high accuracy, for instance the cost of an arithmetic operation, or a function call to any standard math function. More complex tasks, e.g. the task of solving a non-linear system of equations, can prove difficult to estimate accurately. The problem is to estimate such tasks in a convenient and general way, so that combined with the scheduling algorithm, it will produce an accurate estimation of the actual speedup that can be achieved when the parallel program is executed on a parallel computer.

A related research problem that also influences the scheduling algorithm is which parallel computer model (i.e parallel computational model of communication and computation time) should be used, see Section 2.2. If the model is too simple and unrealistic the difference between estimated speedup and measured speedup will be too large. However, if the parallel model is too complicated the scheduling algorithm might increase in computational complexity since it has too many parameters to consider.

1.5.1 Relevance

The relevance of the research problem stated in Hypothesis 1 can be motivated in several ways. First, modeling and simulation is expanding into new areas where earlier it was not possible to model and/or simulate a given problem. However, with modern modeling techniques, such as object oriented modeling languages combined with advanced combined graphical and textual modeling tools, it is now possible to model larger and more complex models. This is a strong motivation for why new methods of speeding up the execution time of simulations are important, since larger and more complex models will otherwise require unacceptable long simulation time.

Moreover, by using modern state-of-the-art modeling tools and languages the modeling and simulation area is opened up to new end-users with no advanced knowledge of modeling and simulation who will probably have even less knowledge of parallel computing. This makes an *automatic* parallelization tool highly relevant if the tool is to become widely used by the modeling and simulation community.

Finally, as indicated above, there is still theoretical work to be done regarding better algorithms for the clustering of fine grained task graphs that are typically produced in this work. For instance, new scheduling and clustering algorithms adapted for a more accurate programming model are needed to increasing the performance of parallel programs, as is further discussed in Chapter 11.

1.5.2 Scientific Method

The scientific method used within this work is the traditional system-oriented computer science method. To validate the hypothesis stated in Hypothesis 1, a prototype implementation of the automatic parallelization was built. Also, theoretical analysis of the scheduling and clustering algorithms used can be used for validating the hypothesis. The newly designed and adapted scheduling and clustering algorithms described in the following chapters have also been implemented in this tool. The parallelization tool produces a parallel version of the simulation code that is executed on several parallel computers. Measurements of the execution time are collected from these executions. When comparing the parallel execution time with that of a simulation performed on a sequential processor (which is preferably a single processor on the parallel computer) an exact measure of the achieved speedup is gained.

Finally, the hypothesis can be validated from the measurements from executions of the generated code and the automatic parallelization tool, together with the theoretical analysis performed on the scheduling and clustering algorithms,

1.6 Assumptions

This research work is based on the several assumptions collected in this section.

The automatic parallelization approach taken in this work only considers static scheduling of parallel programs, i.e., we do not consider dynamic scheduling of tasks. This means that the parallelization tool can, during compile time, determine on how many processors and in which way the parallel program is executed.

Furthermore, the scheduling and task merging algorithms in this work assumes non-preemptive tasks, i.e., that once a task has started its execution it is not interrupted or delayed by other tasks. It executes until its work is completed. Each task starts by receiving its data from its immediate predecessors tasks (i.e., its parents) and once it has finished its computation it sends its data to the immediate successor tasks.

The parallelization approach, including the scheduling and the task merging algorithm, requires that the program can be described using a task graph. It must be possible to build a data dependency graph of the program in the form of a task graph. This restriction means that the program flow can not depend on input data to the program. Such programs are referred to as oblivious programs. However, the research work also discusses how this scope can be partly extended by introducing malleable tasks, which are tasks that can be executed on more than one processor (the number of processors for such tasks can be determined at runtime), giving at least some dynamic flexibility.

1.7 Implementation Work

This research work evolved through several prototype implementations. Figure 1.6 below presents the different prototypes and their relationship. The first automatic parallelization tool called DSBPart was parallelizing the C-code generated by the Dymola tool. This tool was later replaced by the ModPar tool in OpenModelica, to gain more control over solvers and optimization techniques. The DAGS task graph prototyping tool was developed in parallel with ModPar to experiment on clustering, scheduling, and task merging algorithms.

1.8 Contributions

The main contributions of this research include a task merging method based on a graph rewrite system where tasks are merged in a task graph given as a set of graph rewrite rules. This method shows promising results in merging tasks particularly in fine grained task graphs to reduce its size and increase granularity so that it can be better scheduled using existing scheduling algorithms.

Another contribution is the automatic parallelization tool itself, which is integrated into the OpenModelica compiler. It can successfully and automatically parallelize simulations from Modelica models and among the results are speedup figures of executing simulation for up to 16 processors.

A third contribution is insights and experiences in writing compilers using the RML language, based on Natural Semantics. The pros and cons of using

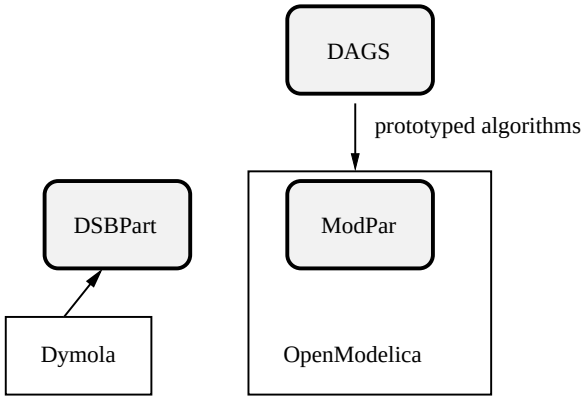


Figure 1.6. The three different implementations made in this thesis work and how they relate.

such language for writing something as advanced as a compiler are discussed.

Other contributions include a prototype environment for designing task scheduling algorithms in the Mathematica tool and contributions of the design of the Modelica language.

1.9 Publications

Parts of this work has been published in the following papers.

Main Author

- Peter Aronsson and Peter Fritzson. A Task Merging Technique for Parallelization of Modelica Models. (Conference paper) In Proceedings of the 4th International Modelica Conference, Hamburg, Germany, March 7-8, 2005.

- Peter Aronsson, Peter Fritzson. Automatic Parallelization in OpenModelica (Conference paper) Proceedings of 5th EUROSIM Congress on Modeling and Simulation, Paris, France. ISBN (CD-ROM) 3-901608-28-1, Sept 2004.

- Peter Aronsson, Levon Saldamli, Peter Bunus, Kaj Nyström, Peter Fritzson. Meta Programming and Function Overloading in OpenModelica

(Conference paper) Proceedings of the 3rd International Modelica Conference (November 3-4, Linköping, Sweden) 2003

- Peter Aronsson, Peter Fritzson. Task Merging and Replication using Graph Rewriting (Conference paper) Tenth International Workshop on Compilers for Parallel Computers, Amsterdam, the Netherlands, Jan 8-10, 2003

- Peter Aronsson, Peter Fritzson. Multiprocessor Scheduling of Simulation Code from Modelica Models (Conference paper) Proceedings of the 2nd International Modelica Conference March 18-19, 2002, DLR, Oberpfaffenhofen, Germany

- Peter Aronsson, Peter Fritzson, Levon Saldamli, Peter Bunus. Incremental Declaration Handling in Open Source Modelica (Conference paper) In Proceedings, SIMS - 43rd Conference on Simulation and Modeling on September 26-27, 2002 at Oulu, Finland.

- Peter Aronsson, Peter Fritzson. Parallel Code Generation in MathModelica / An Object Oriented Component Based Simulation Environment (Conference paper) In Proceedings, Parallel / High Performance Object-Oriented Scientific Computing, Workshop, POOSC01 at OOPSLA01, 14-18 October, 2001, Tampa Bay, Fl. USA

- Peter Aronsson, Peter Fritzson. Clustering and Scheduling of simulation code from equation-based simulation languages (Conference paper) In Proceedings, Compilers for Parallel Computers CPC2001, Workshop, 27-29 June, 2001, Edingburgh, Scotland, UK.

Additional Co-authored Papers

- Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. The OpenModelica Modeling, Simulation, and Software Development Environment. In Simulation News Europe, 44/45, December 2005

- Peter Fritzson, Adrian Pop, Peter Aronsson. Towards Comprehensive Meta-Modeling and Meta-Programming Capabilities in Modelica (Conference paper) 4th International Modelica Conference (Modelica2005), 7-9 March 2005, Hamburg, Germany

- Kaj Nyström, Peter Aronsson, Peter Fritzson. Parallelization in Modelica (Conference paper) 4th International Modelica Conference, March 2005, Hamburg Germany

- Kaj Nyström, Peter Aronsson, Peter Fritzson. GridModelica - A Modeling and Simulation Framework for the Grid (Conference paper) Proceedings of the 45th Conference on Simulation and Modelling, (SIMS'04) 23-24 September 2004, Copenhagen

- Peter Fritzson, Vadim Engelson, Andreas Idebrant, Peter Aronsson, Håkan Lundvall, Peter Bunus, Kaj Nyström. Modelica A Strongly Typed System Specification Language for Safe Engineering Practices (Conference paper) Proceedings of the SIMSafe 2004 conference, Karlskoga, Sweden, June 15-17, 2004

- Peter Fritzson, Adrian Pop, and Peter Aronsson. Towards Comprehensive Meta-Modeling and Meta-Programming Capabilities in Modelica. In Proceedings of the 4th International Modelica Conference, Hamburg, Germany, March 7-8, 2005.

- Peter Fritzson, Peter Aronsson, Peter Bunus, Vadim Engelson, Levon Saldamli, Henrik Johansson, Andreas Karström. The Open Source Modelica Project (Conference paper) Proceedings of the 2nd International Modelica Conference. Oberpfaffenhofen, Germany, March 18-19, 2002.

Other

- Peter Aronsson. Automatic Parallelization of Simulation Code from Equation-Based Simulation Languages (Licentiate thesis) Linköping Studies in Science and Technology, Thesis No. 933, Linköpings Universitet, April 2002

Chapter 2

Automatic Parallelization

This chapter describes tools and techniques for automatically parallelize programs. This includes data structures for representing programs, analysis techniques and algorithms, clustering and scheduling algorithm and code generation techniques.

2.1 Task Graphs

To analyze and detect what pieces of a program can be executed in parallel an internal representation of the program code is needed. Source-to-source restructurers commonly use Abstract Syntax Trees (AST) as internal representation of computer programs together with other data structures for specific program optimizations. For instance, to perform global optimization a program dependence graph is used and for instruction scheduling a data dependency graph with nodes being individual CPU instructions can be used.

In automatic parallelization a task graph can be used for the analysis of programs. A task graph is a Directed Acyclic Graph (DAG), with costs associated with edges and nodes. It is described by the tuple

$$G = (V, E, c, \tau) \tag{2.1}$$

where

- V is the set of vertices (nodes), i.e., tasks in the task graph.
- E is the set of edges, which imposes a precedence constraint on the tasks. An edge $e = (v_1, v_2)$ indicates that node v_1 must be executed before v_2 and send data (resulting from the execution of v_1) to v_2.

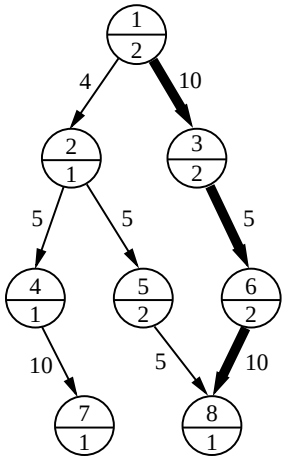


Figure 2.1. Task graph with communication and execution costs.

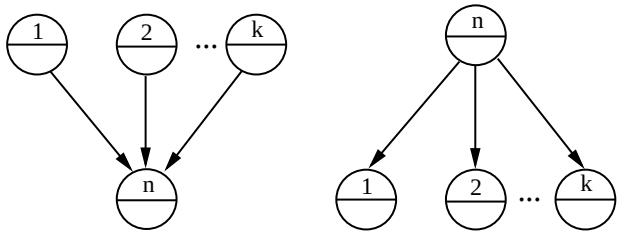
- $c(e)$ gives the cost of sending the data along an edge $e \in E$.
- $\tau(n)$ gives the execution cost for each node $v \in V$.

Figure 2.1 illustrates how a task graph can be represented graphically. Each node is split by a horizontal line. The value above the line represents a unique node number and the value below the line is the execution cost (τ). Each edge has its communication cost (c) labeled close to the edge.

A *predecessor* to a node n is any node in the task graph that has a path to n. An *immediate predecessor* (also called *parent*) to a node n is any node from which there is an edge leading to n. The set of all immediate predecessors of a node n is denoted by $pred(n)$, while the set of all predecessors of a node n is denoted by $pred^m(n)$. Analogously, a *successor* to a node n is any node in the task graph that has a path from n to that node. An *immediate successor* (also called *child*) is any node that has an edge with n as source, and the set of all immediate successors of a node n is denoted by $succ(n)$. Similarly the set of all successors is denoted $succ^m(n)$.

A *join* node is a node with more than one immediate predecessor, illustrated in Figure 2.2(a). A *split* node is a node with more than one immediate successor node, see Figure 2.2(b).

The edges in the task graph impose a precedence constraint: a task can only start to execute when all its immediate predecessors have sent their data to the task. This means that all predecessors to a node has to be executed



(a) A join node (b) A split node

Figure 2.2. Graph definitions

before the node itself can start to execute. In the case when an adjacent node executes on the same physical processor no sending of data is required.

Since the task graph representation is used in this research problem as an input for the scheduling and clustering algorithms the research problem can be generalized to partition any program that can be translated into a task graph. Thus, the algorithms and results given in this thesis can be useful for scheduling sequential programs of any type of scientific computations, given that the programs can be mapped to a task graph.

2.1.1 Malleable Tasks

Sometimes it can be useful to model a task that can be executed on several processors as a single task. For instance a task that has no internal static representation in the form of a task graph but has a parallel implementation that can run on several processors could be represented as a *malleable task*. A malleable task n_m is a task that can be executed on one or on several processors. It has an execution cost function taking the number of processors into consideration, see Equation 2.1.1. This function is always decreasing, since the execution time when increasing the number of processors is decreasing[1]

$$\tau(n_m) = \tau_m(P) \tag{2.2}$$

Here P is the number of processors the task will execute on.

By introducing malleable tasks into task graphs one can generalize the usage of task graph in static scheduling, where the task graph is built at compile time and mapped onto a fixed number of processors. With malleable tasks scheduling decisions that must be delayed until runtime, i.e., dynamic

[1]If the execution time does not decrease by adding more resources/processors, the lowest execution time is gained by only using a subset of the allocated processors.

scheduling, can still be partly handled by a static scheduling approach. The prerequisite being that the number of processors for each malleable task can be set during static analysis.

In this thesis work, malleable task are used for solving linear and non-linear systems of equations, see Chapter 6.

2.1.2 Graph Attributes For Scheduling Algorithms

Scheduling algorithms map the task graph onto a set of processors by using values, or attributes, mostly associated with the nodes of the task graph. Some attributes are used by several algorithms. Others are specific to one particular algorithm. This section defines a subset of such attributes commonly used in the literature.

The most important attribute of a task graph is its *critical path*. The critical path of a task graph is its longest path. The length is calculated by accumulating the communication costs c and the execution costs τ along a path in the task graph. For instance, the critical path in Figure 2.1 is indicated by the thick edges of the task graph, which has a critical path length of 32. The term *parallel time* is also often used for the critical path length [7, 81, 88], and is used as a measure of the optimal parallel execution time. Another term used for the critical path is the *dominant sequence*, used in for instance [88].

The *level* of each task, i.e., of each node in the graph, is defined as:

$$level(n) = \begin{cases} 0 & , pred(n) = \emptyset \\ \max_{k \in pred(n)}(level(k) + \tau(k) + c(k,n)) & , pred(n) \neq \emptyset \end{cases} \quad (2.3)$$

The level of a node is thus the longest path (following edges in opposite direction) from the node itself to a node without predecessors, and accumulating execution costs and communication costs along the path. The level can also be defined in the inverse way as in Equation 2.4 and is then referred to as the *bottom level*. In these cases the first level definition is referred to as the *top level*.

$$blevel(n) = \begin{cases} 0 & , succ(n) = \emptyset \\ \max_{k \in succ(n)}(level(k) + \tau(k) + c(k,n)) & , succ(n) \neq \emptyset \end{cases} \quad (2.4)$$

The relation between the critical path and the level attribute is that for a node on the critical path, the successor with the maximum level will also be on the critical path.

Another pair of important attributes used in many scheduling algorithms, with some varieties regarding the definitions, are the earliest starting time and latest starting time of a node. Other references use different names, such as ASAP (As Soon As Possible) time and ALAP (As Late As Possible) time [87].

We use the terms $est(n)$ and $last(n)$ and the definitions found in [17], which will later be used when explaining the TDS algorithm in Section 2.5.1.

$$est(n) = \begin{cases} 0 & , pred(n) = \emptyset \\ \min_{k \in pred(n)} \max_{l \in pred(n), k \neq l} (ect(l), ect(k) + c_{k,n}) & , pred(n) \neq \emptyset \end{cases} \tag{2.5}$$

$$ect(n) = est(n) + \tau(n) \tag{2.6}$$

$$fpred(n) = max_{k \in pred(n)} (ect(k) + c_{k,n}) \tag{2.7}$$

$$lact(n) = \begin{cases} ect(n) & , succ(n) = \emptyset \\ \min\limits_{k \in succ(n), k \neq fpred(n)} (min \quad (last(k) - c_{n,k}, \quad \min\limits_{k \in succ(n), k = fpred(n)} min(last(k)))) & , succ(n) \neq \emptyset \end{cases} \tag{2.8}$$

$$last(n) = lact(n) - \tau(n) \tag{2.9}$$

- *est(n)* is the definition for the earliest starting time for node n, which means the earliest possible starting time of a node, considering the precedence constraint and the communication costs. It is defined in Equation 2.5.
- *ect(n)* is the earliest completion time for node n, which is defined as the earliest starting time plus the execution time of the node. The definition of the earliest starting time assumes a linear clustering approach, i.e., if the first predecessor of a node is scheduled on the same processor as the node itself, then the rest of the predecessors to the node will not be scheduled on the same processor. This is why the definition, see Equation 2.6, takes the maximum value of the *ect* value of one successor and the *ect* value plus the communication cost for another successor.
- *last(n)* is the latest (allowable) starting time of a node n, i.e., the latest time a node has to start executing to fulfill an optimal schedule, as defined in Equation 2.9.
- *lact(n)* is the latest allowable completion time for a node n, i.e the latest time a node is allowed to finish its execution. The definition is found in Equation 2.8.
- *fpred(n)* is the *favorite predecessor* of a node n, see Equation 2.7, used in the TDS algorithm. It is the predecessor of a node which finishes execution last among all predecessors, thus should be put on the same processor as the node itself to reduce the parallel time.

The difference of the latest allowable starting time and the earliest starting time of a task is sometimes referred to as the scheduling window for the tasks. If the window is large the scheduler has many alternatives of scheduling that particular task. However, if the scheduling window is small, as for tasks on the critical path, the scheduler has less opportunity of moving that task around in time when trying to schedule the task graph.

2.2 Parallel Programming Models

Creating parallel programs, whether it is performed automatically by a tool or manually by a developer or a team of developers, is an complicated and error-prone process. It is often difficult for a developer to estimate the structural or computational complexity of the parallel program he/she has written. By having a parallel programming model to follow, the programmer can be guided both in the design and the implementation of his/her parallel program, and obtain a model over the complexity of the program with regards to time and memory consumption for different multiprocessor architectures as well as for problem instances of varying size.

There are several factors to consider, when choosing a parallel programming model. For instance, how complicated should the model be? Each model is a simplification of the real world. Some programming models which are particularly simple may give large errors in comparison to real world examples. For manual parallel program development one must also consider how easy the model is to comprehend and use for implementation of parallel programs. Since parallel programming is complicated and error prone, it is important that the programming model be simple enough to minimize the effort needed for developers to implement and understand their parallel programs.

The parallel programming models are also important for automatic parallelization. Many of the programming models also include a cost model for the computational complexity of the program. These cost models are commonly used in parallelization tools to guide the decisions of the scheduling and clustering algorithms.

The following sections present some of the most common parallel programming models.

2.2.1 The PRAM Model

The Parallel Random Access Machine (PRAM) model [24] is a simple programming model and also the most popular one, at least if one considers publications of multiprocessor scheduling algorithms. Its popularity is due to its

simplicity, all global data is instantly available on all processors. This is a powerful simplification which can lead to large errors when comparing the model with reality. The PRAM model divides a parallel computation into a series of steps, where each step can be either a read or write operation between local and shared memory or an elementary operation with operands from the local memory. There are also model refinements within the PRAM model based on the strategy taken when two processors access the same shared memory. For instance, the PRAM-Concurrent Read, Concurrent Write (CRCW) model allows both concurrent reads and concurrent writes of shared memory. Concurrent writes to the same memory address leads to conflicts. These can be resolved in different ways, for instance by giving each processor a priority and letting the processor with the highest priority write to the memory address.

The greatest advantage, and the reason for the popularity of the PRAM model, is its simplicity. However, simplicity is also its major drawback. The communication cost is neglected, which often leads to large differences between the model and the reality.

2.2.2 The Logp Model

The Logp model [15] is more sophisticated than the PRAM model. Its name is composed from the four parameters of the model.

- **Latency, L**
  The latency is the fixed size independent time delay involved when sending a message from one processor to another. For instance, when sending data between two processors connected through an Ethernet based network, one term of the latency is proportional to the length of the physical cable connecting the two computers.

- **Overhead, o**
  The overhead of sending a message between two processors is the extra time needed for preparing the sending of the message. Such preparations can be for instance be copying of data into send buffers, calling of send primitives in a communication API, etc. This parameter can vary depending on the underlying architecture. For instance, if the communication is performed by a co-processor the overhead is lower compared to the case when the communication must be handled by the main processor itself. During the overhead period of time, the processor is busy and can not perform other tasks.

- **Gap, g**
  The gap is defined as the time interval between two consecutive sends (or receives) of messages. This parameter can be motivated if for instance

a co-processor handles the communication and it is busy some period of time after a message sending request has been received. During that time, additional requests have to be postponed, thus giving a gap time between consecutive sends.

- **Processors, p**
  The last parameter defines how many processors the problem is partitioned for.

The model also includes a capacity limit on the connecting network. A maximum of $\lceil L/g \rceil$ messages can be sent between processors at the same time.

2.2.3 The BSP Model

In the Bulk-Synchronous Parallel (BSP) model [82] all tasks (processors) synchronize at given time steps. Between these steps each processor performs individual work only on local data. At each synchronization step, global communication between processors occur. The time between two synchronization steps is called a super-step. The BSP model defines a computer as a set of components, each consisting of a processor and local memory, connected together through a network. The BSP model has the following parameters:

- **P**
  The number of processors.
- **l**
  The cost of performing a barrier synchronization is given by the parameter l.
- **g**
  The parameter g is associated with the bandwidth capacity. It is defined such that $g \cdot h$ is the time it takes to communicate (i.e., either send or receive) h messages.

The total execution cost of a super-step, using the parameters above, is $l + x + g \cdot h$, where x is the maximum execution cost among the processors within the super-step and h is the maximum number of messages sent or received by one of the processors.

2.2.4 The Delay Model

The models mentioned so far have a stronger focus on data parallel programs than task parallel programs. For task parallel programs there is a simple model called the *Delay* model, which corresponds to the task graph model

described earlier in Section 2.1. Some authors instead use the term *macro data flow model*, for instance in [81]. The delay model has a task graph where a communication cost, i.e., a *delay* cost, is associated with each edge. This model is the most common for scheduling and partitioning of task graphs, even if alternative models for scheduling and clustering algorithms are starting to appear in literature, like for instance the *Logp* model.

2.3 Related Work on Task Scheduling and Clustering

During the past three decades extensive research has been made in the area of scheduling and clustering of parallel programs for execution on multiprocessor architectures. Early research in the area focused on simple parallel programming models with many restrictions on the models. Over the years, these early restricted models have become more precise and therefore also more complicated. More efficient scheduling and clustering algorithms have also been developed. The following two sections introduce some of the many scheduling and clustering algorithms found in literature and explain common techniques used in these scheduling and clustering algorithms.

2.4 Task Scheduling

A task scheduling algorithm traverses the task graph (a directed acyclic graph, DAG), as defined in Section 2.1. The output of the algorithm is an assignment of each node $n \in V$ to a processor, and a starting time of each node i.e., a partial order of the nodes in the graph. The general case of the task scheduling problem has been proved to be NP-complete [80], thus it is often required to use heuristics in scheduling algorithms.

The scheduling problem usually has some other constraints except for the task graph itself. Many scheduling algorithms assume that once a task has started executing it will continue until its termination, i.e., so called non-preemptive scheduling. The opposite, that a task can be interleaved with execution of other tasks or even migrate to other processors, is called preemptive scheduling. In this thesis we only consider the first case, non-preemptive scheduling.

Some classes of scheduling algorithms can schedule the DAG for any given number of processors, whereas other algorithms require an unlimited number of processors. An unlimited number of processors as a requirement means that the number of processors available can not be specified as an input to the algorithm. Thus the processor requirement varies over different problem

instances. This is often referred to as scheduling for a set of virtual processors. There are even algorithms for a specific number of processors, e.g. there exist a polynomial time optimal scheduling algorithm for two processors [13].

2.4.1 Classification

There are many ways to classify task scheduling algorithms. One classification scheme that covers a broad area is given in [12]. It gives a hierarchical categorization of the algorithms, see Figure 2.3. At the top classification level, algorithms belong to one of two categories, *local scheduling* or *global scheduling*. Local scheduling involves scheduling of tasks locally on one processor, while global scheduling considers the scheduling problem involving multiple processors. Furthermore, global scheduling can be subdivided into *static* and *dynamic* scheduling. In this work we have so far only considered static scheduling, i.e., the scheduling takes place at compile time. Static scheduling is further sub-categorized into *optimal* and *suboptimal* scheduling.

The *suboptimal* category contains two subcategories, *heuristic* and *approximate* scheduling. The difference between the two is that heuristic scheduling algorithms tend to use heuristic parameters to control the scheduling behavior whereas approximate scheduling algorithms instead can use a more ad-hoc method of finding a good enough solution. Approximate scheduling algorithms can be of four different kinds: *enumerative*, *graph-theory*, *mathematical programming*, and *queuing theory*. These four classifications are also used for subdividing the optimal static scheduling algorithms.

The contributions in this thesis mainly belong to the graph theory suboptimal category.

2.4.2 List Scheduling Algorithms

The list scheduling technique has been extensively studied in the literature [28, 36, 65, 66, 74]. The list scheduling algorithms belong to the *heuristics* category in the classification scheme given in section 2.4.1. Thus, list scheduling is a suboptimal static scheduling technique. This is illustrated when studying list scheduling in closer detail.

All list scheduling algorithms keep a list of tasks that are ready to be scheduled, often called the *ready-list*. The ready-list contains all task nodes that are free, which means that all predecessors of the node already have been scheduled. In each step of the algorithm, a heuristic assigns a priority to each task node in the ready list and chooses one of the nodes with the highest priority value to be scheduled for execution on one of the processors. A common parameter that is included in the heuristic is the (bottom) *level* of the node, see Equation 2.4. If the bottom level of a node has a high value,

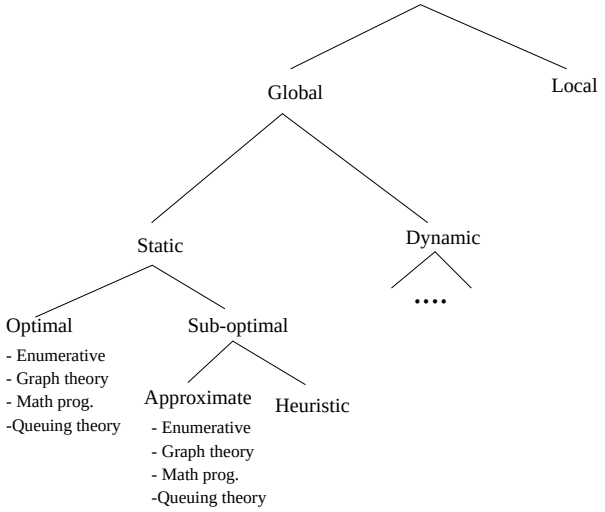


Figure 2.3. An hierarchical classification scheme of task scheduling algorithms.

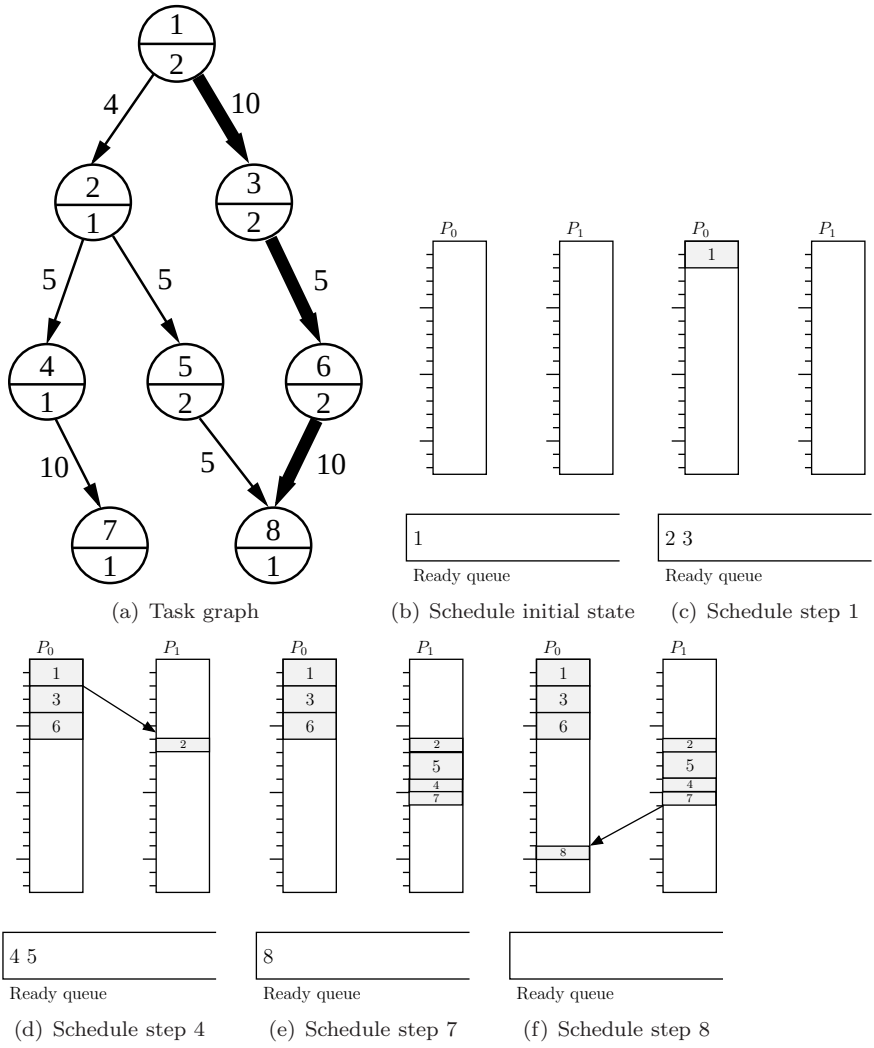
there are many computations to be performed after the node has finished its execution. Therefore, that task node should be given a higher priority when choosing task nodes from the ready list, compared to another task node with a much lower level value.

When a task node has been scheduled it is removed from the ready list and potential successor nodes to the scheduled task node are added. The algorithm terminates when the ready list is empty, i.e., all task nodes have been scheduled. For instance, consider the example in Figure 2.1 to be scheduled using a list scheduler. Figure 2.4 shows how it could be scheduled and how the ready queue gets populated with new task nodes ready to be scheduled.

Note that list scheduling is a compile-time scheduling technique, i.e., all scheduling is performed before execution starts. There are also dynamic techniques very similar to the list scheduling, like for instance dynamic load balancing and dynamic list scheduling algorithms. They often have a load-balancing heuristic that tries to balance the execution load among all processors of the parallel computer.

The ERT Algorithm

One well-known list scheduling algorithm is the Earliest Ready Task (ERT) algorithm [36]. As for all list scheduling algorithms the algorithm starts by



(a) Task graph (b) Schedule initial state (c) Schedule step 1

(d) Schedule step 4 (e) Schedule step 7 (f) Schedule step 8

Figure 2.4. The work of a list scheduling algorithm

.

putting all tasks without any predecessors in the list of tasks ready to schedule. The next step in the algorithm is to calculate (for each processor) the earliest starting time (called ready time) for all tasks in the ready list. This is done by taking the maximum finishing time $F(k)$ added to the communication time among all parents of the task. This calculation is shown in Equation 2.10 according to [36]. In the ERT algorithm, the communication cost is divided into two parameters: the first parameter is the size of the message sent between tasks, $d(k, n)$ in Equation 2.10. The second parameter, $t_{comm}(P_i, P_j)$, is the time per data size unit required to send one message from processor P_i to processor P_j. Finally, the *Alloc* function in Equation 2.10 performs the allocation of a task, returning a processor number.

$$X(n, P_i) = max_{k \in pred(n)}(F(k) + d(k, n) * t_{comm}(Alloc(k), P_i)) \tag{2.10}$$

However, the calculation of the ready time $X(n, P_i)$ above can not always be fulfilled, since no check if the processor is available is performed, i.e., it is not known whether the processor already have a task scheduled for execution at that point in time. Therefore, once the ready time has been calculated for *all* processors($X(n, P_i)$), the *real* ready time is calculated by considering if each processor is available or not, as performed in Equation 2.11.

$$R(n, P_i) = max(Avail(P_i), X(n, P_i)) \tag{2.11}$$

Then, the processor giving the earliest starting time is calculated for each task, see Equation 2.12 (m is the number of processors).

$$R(n) = min_{j \in \{1,..,m\}} R(n, P_j) \tag{2.12}$$

Once all these calculations have been performed, the task to choose from the ready list can be calculated. We simply choose the task from the ready list with the minimum value for R, and allocate it to the processor for which that minimal value was achieved. Thus, the ERT algorithm is greedy in the sense that is always selects the task which has the minimal ready time.

The complexity of the ERT algorithm is shown in [36] to be $O(mn^2)$, for m processors and n tasks in the task graph.

The ERT algorithm has some heterogeneous features. The communication costs between processors can be set individually, allowing for a somewhat more flexible network. It is however uncertain if the algorithm performs well for real heterogeneous multiprocessor systems, since different processor speeds is not supported.

2.4.3 Graph Theory Oriented Algorithms with Critical Path Scheduling

Another category of static scheduling algorithms is the group of graph theory oriented algorithms. In this class of algorithms we find techniques such as critical path scheduling [60] and several clustering approaches [16, 39].

The critical path scheduling technique identifies the critical path of the task graph, see Section 2.1.2. Then it schedules all task nodes on the critical path on one processor. After the nodes on the critical path have been scheduled onto the same processor, the communication costs between the nodes on the critical path becomes zero. Hence, after the scheduling of the nodes on the critical path, a new critical path will appear in the task graph. The algorithm will then schedule this critical path onto the next processor, and so on.

2.4.4 Orthogonal Considerations

The classification scheme presented in section 2.4.1 does not cover all aspects of scheduling algorithms. There are some features that are orthogonal to the classification scheme. However, these considerations are important in this work and are therefore explained in detail below.

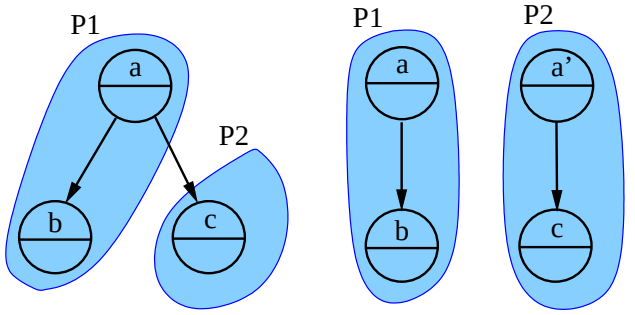
Task Replication

One approach for improving the efficiency of a scheduling algorithm that has increased in popularity over the past decade is to employ task replication as a means of reducing the communication cost [17, 34, 41, 58]. The use of task replication to reduce the total execution time of a parallel task graph is illustrated in Figure 2.5.

For certain applications where the cost of communication is far from negligible, replicating a task to several processors can decrease the execution time significantly [34]. A drawback of using task replication in a scheduling algorithm is that it increases the time complexity of the algorithm. The increase can be substantial. For instance, the CPFD algorithm[34] has a time complexity of $O(n^4)$.

Granularity

An important metric for task graphs with communication costs as defined in 2.1 is the granularity of the graph. The literature also contain variants of the definition of granularity [34, 40, 56]. For instance, another variant is to take the average values for the communication and execution costs. Other authors use the term Communication to Computation Ratio (CCR) instead



(a) Three tasks assigned to two processors without replication.

(b) Three tasks assigned to two processors with task a replicated.

Figure 2.5. Using task replication to reduce total execution time.

of granularity [34]. The granularity g is defined by Equation 2.13, i.e., the maximum execution cost of a node divided by the minimum communication cost of an edge:

$$g = \frac{\max_{n \in V} \tau(v)}{\min_{e \in E} c(e)} \tag{2.13}$$

This definition has some limitations. It does not allow communication costs to be zero (which gives infinite granularity). Therefore, several alternative definitions are available, taking the average values instead of min and max, calculating granularity for each task node of the graph, etc. 'However, in this research problem, the definition works fine since we do not consider communication costs to be zero for any edge. Note that scheduling algorithms might set communication cost to zero but that does not affect the original task graph, for which the granularity is defined.

The granularity factor is an important metric for task graph scheduling. A fine grained task graph, i.e., when the granularity value is low due to large communication costs and small execution costs, the scheduling algorithm must take a large responsibility for preventing communication when possible. One approach might be to apply task replication to prevent communication. Another way of handling the problem is to increase the granularity of the task graph. One such method is called grain packing. Grain packing is a method for increasing the granularity of the task graph by merging tasks [31], see also Section 2.6.

Task Graphs With Fixed Structure

Many scheduling algorithms have certain properties for specific structures of task graphs. For instance, a common structure of a task graph is a task graph that is an out-tree. Out-trees have one node with no predecessor, the successors of the node are all independent, each of them with their own independent successors, and so on. Scheduling and clustering algorithms can for instance take advantage of such task graphs not having join nodes, and thereby perform a better schedule compared to arbitrary task graphs.

2.5 Task Clustering

Task clustering algorithms perform part of the work of a scheduling algorithm. A cluster is a set of tasks, designated to execute on the same processor. The goal of a task clustering algorithm is to reduce the critical path of the scheduling algorithm by explicitly assigning nodes to clusters, reducing the communication costs to zero for edges with both nodes in the same cluster.

The execution order within the cluster does not necessarily need to be determined, except of course that it must fulfill the precedence constraints imposed by the edges of the task graph. The algorithm does not determine when the nodes in the cluster starts to execute. Thus, in order to achieve the same function as a task scheduling algorithm, a task clustering algorithm needs to be followed by a second phase, which usually is a simple list scheduler.

However, some clustering algorithms can constrain the scheduling order by introducing additional data dependency edges in the task graph, giving the scheduling algorithm that follows task clustering an easier problem to solve. One such algorithm is the DSC algorithm, explained in further detail in Section 2.5.3.

2.5.1 TDS Algorithm

The TDS (Task Duplication based Scheduling) algorithm is a *linear clustering* algorithm, with task replication [17] Linear clustering means that the algorithm only assigns *one* predecessor of a node to the cluster containing the node itself. Thus, the clusters form linear paths through the task graph. Due to the linear clustering technique of the TDS algorithm, it needs to be followed by a second scheduling or mapping phase that maps the assignments to physical processors.

The first step in the TDS algorithm is to calculate the earliest starting time (*est*) and the latest allowable starting time (*last*), and some other additional parameters for each node is the task graph. Among these parameters is a

favorite predecessor assignment for each node. The favorite predecessor is the predecessor with the maximum *ect* (earliest completion time) value plus the communication cost, defined in Equation 2.7.

The linear clustering is performed by following the favorite predecessors (*fpred*) of the nodes backward up through the task graph, and assigning the collected tasks among the traversed path to a processor.

When following predecessors up through the task graph, eventually a node which has already been assigned to a processor will be considered. The TDS algorithm will then check if the task is critical or not. A task is x is *critical* for a predecessor task, y if Equation 2.14 is fulfilled. This constraint says that a predecessor task y is critical to a task x if the effect of placing them onto two different processors will invalidate the latest allowable starting time of task x since the communication cost $c_{x,y}$ will have to be considered, increasing the latest allowable starting time ($last(x)$).

$$last(x) - lact(y) < c_{x,y} \tag{2.14}$$

The TDS algorithm will only replicate tasks that are critical, keeping the number of replicated tasks low. If the favorite predecessor has already been assigned to a processor, and it is not critical, the algorithm will follow another predecessor when traversing the task graph upwards.

The computational complexity of the TDS algorithm is $O(n^2)$ for a task graph with n tasks. If a cost relationship between the execution costs of tasks and the communication costs of edges is fulfilled the TDS algorithm produces the optimal schedule on an unlimited number of processors. However, this cost relationship is in practice only fulfilled for coarse grained task graphs. This makes it less suitable to be used directly on the fine grained task graphs produced by the automatic parallelization tool in this research.

2.5.2 The Internalization Algorithm

A task clustering algorithm called internalization is presented in [81]. The internalization algorithm is a task clustering algorithm which traverses all the edges of the task graph and checks if internalizing the two tasks associated with an edge will cause an increase in the total parallel execution time. Internalizing two task means assigning them to the same processor, i.e., putting them into a common cluster. This also means that the communication cost between the two tasks are zero.

The edges are first sorted in descending order of communication cost. Hence, the most costly edge is considered first. The algorithm checks if the parallel time decreases when the edge is internalized. If so, the clustering of

the two nodes is performed, otherwise not. This step is followed by a recalculation of the parallel time, along with calculation of other task attributes, after which the algorithm continues with the next iteration.

The complexity of the internalization algorithm is $O(n^2)$, for a task graph containing n nodes.

2.5.3 The Dominant Sequence Clustering Algorithm

Another task clustering algorithm specially designed for a low time complexity is the *Dominant Sequence Clustering* (DSC) algorithm [88]. Similar to the internalization algorithm it starts by placing each node in its own cluster. It subsequently traverses all nodes in a priority based manner, merging clusters and zeroing the communication label of edges as long as the parallel time of the task graph does not increase. Zeroing an edge means that the clusters where the two nodes resides are merged, hence making the communication cost of the edge reduced to zero, i.e., the same as internalization of two tasks as described in Section 2.5.2 above.

The simplified version of the algorithm (DSCI)is given in Figure 2.6. The first step is to calculate the *blevel* for each node. The *blevel* is the longest path from the node to an exit node, i.e., a node with no successors. Similarly, the *tlevel* is the longest path from a node to a top node, which is a node without any predecessors. This calculation is performed for all entry nodes of the task graph.

algorithm DSCI($G = (V, E, \tau, c) : graph$)
  Calculate blevel for all nodes
  Calculate tlevel for each node n where $pred(n) = \emptyset$
  Assign each node to a cluster
  UEG = V, EG = $\emptyset$
  while UEG $\neq \emptyset$ **do**
    n_f = free node with highest priority from UEG
    Merge n_f with the cluster of one of its predecessor such that $tlevel(n_f)$
    decreases in a maximum degree. If $tlevel(n_f)$ increases, do not perform the merge.
    Update priority values for the successors of n_f
    $UEG = UEG - \{n_f\}$
    $EG = EG + \{n_f\}$
  end while

Figure 2.6. The simplified DSC algorithm, DSCI. UEG is the set of remaining nodes and EG is the set of already completed nodes.

When the initial calculations have been performed, the main loop of the algorithm can start. The first line of the loop identifies a free node with the

highest priority.

A free node is a node for which all its predecessors already have been considered in earlier iterations, i.e n is free iff $k \in EG, \quad \forall k \in pred(n)$. This terminology is the same as is used for list scheduling algorithms, see Section 2.4.2.

The priority which is used for selecting a task in the first step of the loop is defined in Equation 2.15.

$$PRIO(n_f) = tlevel(n_f) + blevel(n_f) \tag{2.15}$$

Once a task node has been chosen, the clusters of the predecessors of the node are considered for merging. The algorithm merges the cluster associated with the chosen node, n_f, with the cluster of the predecessor which will reduce the parallel time to a maximum degree. The parallel time, PT is defined as:

$$PT = max_{v \in V} PRIO(n) \tag{2.16}$$

However, if the merge results in an increase of the parallel time, the merge operation is aborted, leaving the cluster associated with n_f as a unit cluster, and the next iteration is performed.

The merging of two or more clusters means that all the nodes in each of the clusters are put together into the same cluster. Additionally, when the merge is performed, the edges between nodes in the same cluster are zeroed, i.e., their communication cost become zero. The merge operation is also responsible for adding pseudo edges such that each cluster has a determined schedule. Figure 2.7 illustrates the addition of a pseudo edge. In order for the algorithm to determine a schedule a pseudo dependency edge is added between task nodes 2 and 3, forcing the scheduler to schedule task 2 before task 3.

Finally, the algorithm terminates when all tasks have been examined, resulting in a clustered task graph.

In [88] the initial version of the DSC algorithm is presented (also shown in Figure 2.6) and weaknesses of that algorithm are identified, after which an improved algorithm is designed. One weakness identified is that the initial version of the DSC algorithm does not work well for join nodes. A join node is a node with several incoming edges, i.e., several predecessors. The initial DSC only clusters a join node with *one* of the predecessors. However, the optimal solution could include merging several predecessors together with the node.

Also, the initial version was improved to consider partially free nodes as being subject of selection when choosing nodes. A partially free node is a node that has some of its predecessors considered, but not all. The reason for considering these nodes is that if a partially free node that lies on the critical path is not considered before other nodes, the non-critical path nodes could

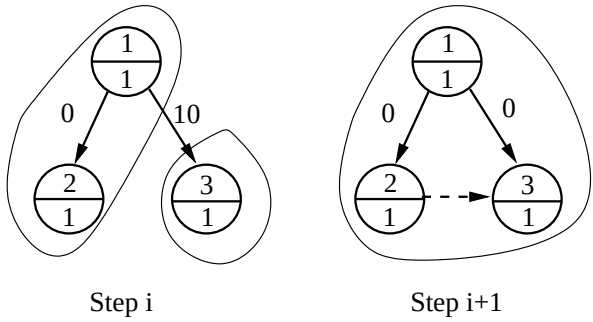


Figure 2.7. The addition of pseudo edges when performing a DSC-merge. By adding an edge from task node 2 to task node 3 the schedule becomes evident: Task 2 is executed before task 3.

be merged such that the critical path is not reduced to a maximum degree, see [88] for details.

One drawback with the DSC algorithm is that the clusters formed by the algorithm do not imply that the nodes can be merged in a *strong* meaning, i.e., merged such that all communication of the merged task is performed before and after the computation of the merged task. By *merging* nodes we normally mean that the execution parts of the nodes to be merged are accumulated into one task, with the all of the communication taking place strictly before and strictly after the execution of the accumulated task. The DSC algorithm does not support this. Instead the communication of data between a task inside a cluster to a task outside the cluster must be performed immediately after the execution of the task residing in the cluster. The clustered task graph in Figure 2.8, also found in [88] shows this problem. The data produced by node 1 needs to be sent immediately to node 3, which belongs to another cluster.

The reason for the *merge* problem being a potential a drawback is that for task graphs with high granularity value, a real *merge* including communication is required to cluster several messages together. For fine-grained task graphs, the communication cost is dominated by the latency. Thus, by merging several messages together large improvements can be made. Related work of task merging is discussed in more detail in the next section.

2.6 Task Merging

Task merging is *stronger* in the way tasks are joined compared to *task clustering*. When tasks are clustered, they are only determined to be executed on

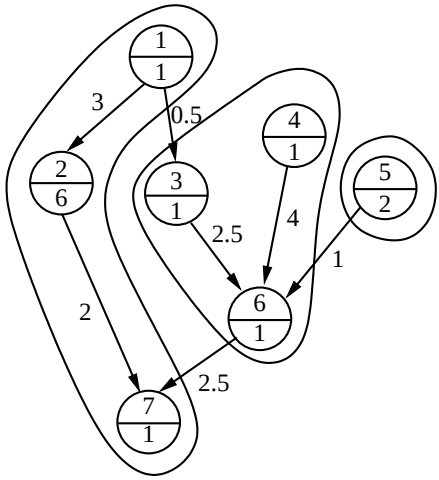


Figure 2.8. A task graph clustered by the DSC algorithm

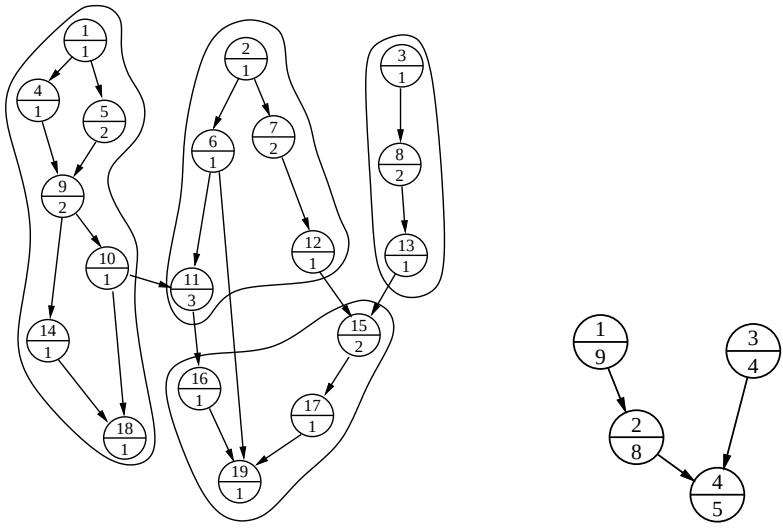
the same processor, which means that the communication cost between tasks belonging to the same cluster are zero. However, the communication of messages between tasks of the cluster and other clusters is still performed at the task level. As soon as each individual task has executed, it individually sends the messages to each of its successor tasks.

Task merging, on the other hand, performs a complete merge of the tasks, by joining the work performed by each individual task into a single work item and composing the in-going messages to the tasks in the cluster into a single message, and the outgoing messages into another single outgoing message. Figure 2.9 shows how a merge is performed. The cluster of tasks is merged into a new task graph that still is a DAG.

The merging strategy can only preserve the DAG properties of the task graph if some constraints are put on which tasks that can be merged. Since the task graph can not contain any cycles, merging of two nodes that introduce a cycle is not allowed. For instance, task a and c in Figure 2.10 is not allowed to be merged, since that will introduce a cycle in the resulting task graph.

2.6.1 The Grain Packing Algorithm

A combined scheduling and task merging technique called *grain-packing* is presented in [31, 32]. The grain packing algorithm is designed to handle fine grained task graphs, i.e., task graphs with high granularity. The grain-packing



(a) The original task graph before the merge has been performed.

(b) The resulting task graph after merge.

Figure 2.9. An example of task merging.

algorithm is a complete scheduling algorithm, i.e., it schedules a fine grained task graph onto a fixed number of processors.

The algorithm is divided into four steps, as explained in [32]:

- **Building a task graph**
  The first stage is to build a fine grained task graph. This approach in [32] builds the task graph at the expression level, as is done in this thesis work.

- **Scheduling**
  The fine grained task graph is then scheduled using a scheduler for a fixed number of processors. In [32] a scheduling algorithm called Duplication Scheduling Heuristic (DSH) is used. The DSH algorithm has a complexity of $O(n^4)$, where n is the number of tasks.

- **Grain-packing**
  After the scheduling algorithm has executed, a grain packing algorithm analyzes the schedule and tries to merge tasks together in order to reduce the parallel time. The grain-packing also includes task replication, i.e., replicating grains from other processors to further reduce execution time.

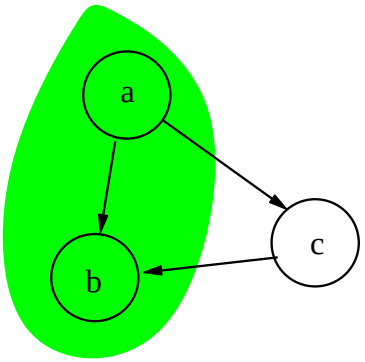


Figure 2.10. Merging of two tasks (here task a and task b) that introduce a cycle is not allowed.

- **Code generation**
  Finally, code generation is performed based on the grains (merged tasks) from the previous step.

The advantage of the grain-packing technique is that since the scheduling algorithm works on the fine grained task graph, all possible kinds of parallelism can be exploited. Thereafter, refinements of the schedule are performed, resulting in a suitable grain size.

One disadvantage is that since the grain-packing is performed after scheduling, the scheduling algorithm works on the large task graph which is only reduced after scheduling [32]. Therefore the quite computationally expensive scheduling algorithm becomes the bottleneck of the scheduling problem. In this thesis we propose to perform task merging prior to scheduling instead to reduce the computational work of the scheduling algorithm.

2.6.2 A Task Merging Algorithm

In [7] a task merging algorithm is presented. The input to the algorithm is a fine grained task graph, from which the algorithm produces a new task graph which has a higher granularity value and fewer tasks. The complexity of the task merging algorithm, or code partitioning algorithm which is the term used in [7], is $O(e \cdot n^3)$ for a task graph with n nodes and e edges.

The basic idea in the algorithm is to repeatedly choose a pair of tasks to merge by using a heuristic. The parallel time, i.e., the length of the critical path, is calculated provided the two tasks are merged. If the parallel time has decreased since the last iteration, the merge is approved and the algorithm

continues by choosing two new nodes using the heuristic. The heuristic is based on a number of criterias. The most important criteria is that only tasks connected by an edge will be subject to a merge operation. This is obvious, since a merge of two tasks connected by an edge will not produce a loss in parallelism in the resulting task graph, since the two tasks are already sequential because of the edge.

Additional criterias are for instance if the edge connecting the two chosen tasks belongs to the critical path of the task graph, or if the merge of the edge connecting the two tasks introduces a cycle in the resulting task graph. Introducing cycles can not be allowed. Therefore a merge of two tasks causing a cycle can not be performed.

2.7 Conclusion

There is much related work in the literature on scheduling and clustering of task graphs for multiprocessor architectures. When considering fine grained task graphs with low granularity values (according to the definition of granularity given in Section 2.4.4), task clustering and task merging algorithms are needed. Ordinary scheduling algorithms designed for coarse grained task graphs does not work well for fine grained task graphs that are targeted in this work. One such coarse grained approach, also targeting simulation code, is described in [84].

Clustering and task merging algorithms often consist of several phases with a normal scheduling algorithm as the final phase. Therefore, scheduling algorithms combined with task clustering or task merging algorithms are needed for scheduling a fine grained task graph for a multi processor architecture.

2.8 Summary

Automatic parallelization deals with the difficult task of trying to generate parallel programs automatically from sequential computer programs. This is performed by analyzing a sequential program and build a data dependency graph of the program called a task graph. The task graph is given costs for communication between tasks and execution costs of the tasks themselves.

Thereafter the multiprocessor scheduling problem deals with how to map the task graph onto a parallel processor architecture by using several techniques. First, a task clustering or task merging algorithm can be applied to make the task graph more suitable and easier to schedule. In this phase the number of processors is typically unlimited.

The second phase is then to schedule the task graph onto a fixed number of processors using any of the large number of scheduling algorithms presented in the literature.

However, for fine grained task graphs most scheduling algorithms fail or perform poorly. This is where this thesis work contributes by providing a method of merging tasks and increasing the granularity in a task graph such that after wards, any of the well-known scheduling algorithms can be used.

Chapter 3

Modeling and Simulation

This chapter describes modeling of complex dynamic systems using equation based languages such as Modelica and how these systems can be simulated.

3.1 The Modelica Modeling Language

This work uses powerful modeling and simulation technology for dynamic and complex physical systems of several application domains. The best representative for this technology is the new modeling language Modelica [49], a modern object oriented equation-based modeling language well suited for modeling of large and complex physical systems using differential and algebraic equations (DAEs).

3.1.1 A First Example

A trivial Modelica model with two variables and two ordinary differential equations (ODE) is given in Figure 3.1. The corresponding mathematical formulation of the equations for this model is presented in Equation 3.1- 3.4.

$$\dot{x}(t) = 4y(t) - x(t) \tag{3.1}$$

$$\dot{y}(t) = -2x(t) \tag{3.2}$$

$$x(0) = 5.2 \tag{3.3}$$

$$y(0) = 0 \tag{3.4}$$

The variables are of the built-in type `Real`. The `x` variable has an optional modification (start=5.2) of the start[1] attribute, setting its initial value to 5.2

[1]Start values for variables in Modelica specify guesses for their initial values when the simulation starts. Here this is at time=0, used to specify the initial conditions

compared to the default value of 0 for Real variables. The `y` variable has its `start` value set to zero. After the variable declarations, an equation section follows, specifying the two equations of the model. The `der` operator specifies the derivative of a variable.

The solution from a simulation of a Modelica model contains a number of functions of time. For each variable, and each derivative, a sequence of values for different time steps is returned from the execution of the simulation. These variables can then be plotted, or processed in other ways. For instance, the value curve of the `x` and `y` variable from simulating the `ODE` model for ten seconds is plotted in Figure 3.2.

```
model ODE
  Real x(start=5.2);
  Real y(start=0);
equation
  der(x)=4*y-x;
  der(y)=-2*x;
end ODE;
```

Figure 3.1. A small ODE example in Modelica.

3.1.2 Basic Features

The basic building block in the Modelica modeling language is the *class*. Everything in Modelica is a class or instance of a class. For example, the model definition above is in fact a so called restricted class using the keyword *model* instead of *class*. Restricted classes are classes with imposed restrictions. For instance, the restricted class *connector* can not contain any equations and the restricted class *package* can only contain other classes and constants. Classes can be instantiated inside other classes, enabling an hierarchical modeling methodology. The end user can build complex models by instantiating classes to create objects inside user defined model definitions and connecting these objects together.

For example, consider the class definition of `TinyCircuit` below. It contains declarations of two instances, named `R1` and `G`. Each instance is also given a corresponding comment, which is used as documentation. Modelica has support for documentation as part of the language (the grammar) on several places in the syntax, e.g. at instance declarations, equations, and enumerations, etc.

```
model TinyCircuit
```

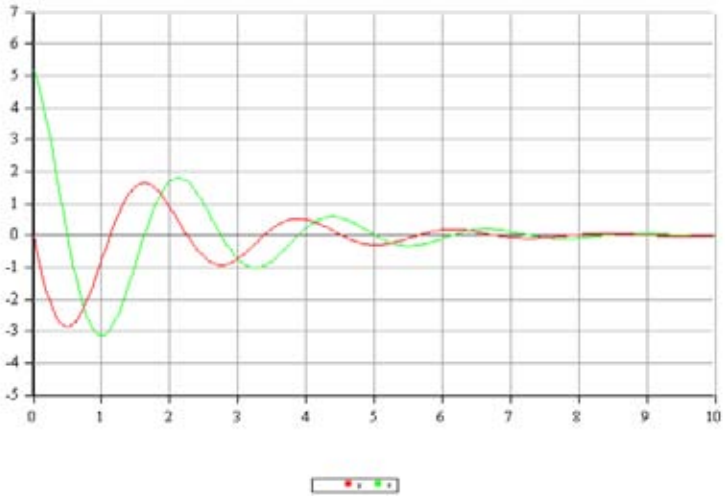


Figure 3.2. The plot of the solution variable x and y after simulating for 10 seconds.

```
   Resistor R1 "The resistance in the Circuit";
   Ground G "The ground element (V=0)";
end TinyCircuit;
```

The `TinyCircuit` model is far from a complete model, since we only created two instances, but did not relate these to each other in any way. Each instance is a variable which in turn can contain other variables and constants (parameters), e.g. a resistor contains current and voltage variables and a resistance R parameter. There is not yet any relation between the variables of instance `R1` and the variables of instance `G`. We therefore make a new attempt at forming a simple circuit model, also using inheritance. A graphical representation of this model is also shown in Figure 3.3.

```
model TinyCircuit2
  extends TinyCircuit;
equation
  connect(R1.n,G.p);
end TinyCircuit2;
```

This model inherits the instances (and equations) from the model `TinyCircuit`. The next part of the model definition is an equation clause, where a `connect`

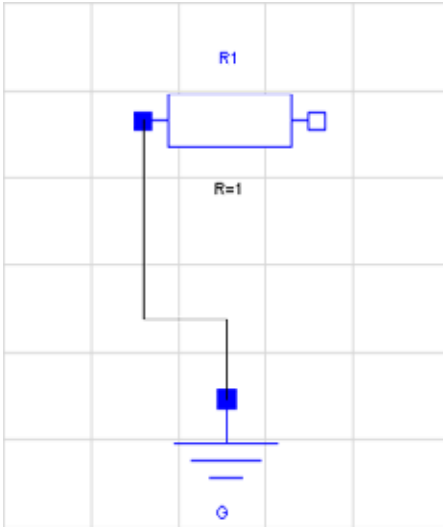


Figure 3.3. The TinyCircuit2 model in a graphical representation.

equation creates coupling equations from the `Resistor` component `R1` with the `Ground` component `G`, by referencing their internal connector components `n` and `p` respectively.

The connectors are instances of a restricted class called *connector*, which together with the `connect` operator constitutes the connection mechanism for Modelica components. Instances of connector classes constitute the interfaces of a component, i.e., how it connects to the outer world. A connector instance contains variables used for communicating with other components. For instance, when building models of electrical components a connector class for electrical properties is needed. The Modelica standard library (MSL) [48] contains two connector classes for simple electrical components (one for positive pins and one for negative pins). The positive pin class definition is:

```
connector PositivePin
   SIunits.Voltage v;
   flow SIunits.Current i;
end PositivePin;
```

The electrical connector contains two variables, the voltage and the current. When two or more connectors are connected using a `connect` equation, the corresponding non-flow variables of the connectors are set equal. But if a

connector variable is prefixed with the `flow` keyword, these connector variables are instead summed to zero. Thus, the voltages are set equal and the currents are summed to zero in electrical connections, corresponding to Kirchhoff's laws.

Since many simple electrical components have two pins, this information is collected into a class called `OnePort`[2], which is a base class for electrical components with two pins:

```
partial model OnePort
   SIunits.Voltage v;
   SIunits.Current i;
   PositivePin p;
   NegativePin n;
equation
   v = p.v - n.v;
   0 = p.i + n.i;
   i=p.i;
end OnePort;
```

The `partial` keyword indicates that the model (class) is an abstract class, i.e., does not have a complete set of equations and variables and therefore can not be simulated by itself since only partial information is given. This can be determined by looking at the variables (`v, i, p.i, p.v, n.i, n.v`) and the equations. There are five variables but only three equations, which makes the system unsolvable (no unique solution can be found). The `OnePort` model contains two variables for keeping the "state" of the electrical component, the current through the component and the voltage drop over it. It also contains two connectors, which are instances of connector classes, one for the positive pin (`p`) and one for the negative pin (`n`).

The `OnePort` base class can be inherited by many electrical components, for instance an inductor, defined in the Modelica Standard Library as:

```
model Inductor
   extends OnePort;
   parameter SIunits.Inductance L=1;
equation
```

[2]The term OnePort is used by specialists in the electrical modeling community to denote components with two physical connection points. This term can be confusing since the ordinary English language meaning of port is as a kind of communication or interaction point, and OnePort electrical components obviously have two ports or interaction points. However, this contradiction might be partially resolved by regarding OnePort as a structured port containing two subports corresponding to the two pins.

```
   L*der(i)=v;
end Inductor;
```

or Resistor, defined as:

```
model Resistor
   extends OnePort;
   parameter SIunits.Resistance R=1;
equation
   v=R*i;
end Resistor;
```

Once the basic electrical components have been described, which already are provided in the Modelica Standard Library, most basic electrical circuits can easily be modeled. For instance, the simple electrical circuit in Figure 3.4 has the following Modelica definition:

```
model DAECircuit
   Resistor R1(R=100);
   Resistor R2(R=470);
   Capacitor C1(C=0.0001);
   Inductor L1(L=0.001);
   Ground Ground;
   SineVoltage V(freqHz=50,V=240);
equation
   connect(V.p, R1.n);
   connect(R1.n, C1.n);
   connect(C1.p, R2.n);
   connect(R2.n, Ground.p);
   connect(L1.p, Ground.p);
   connect(L1.n, R1.p);
   connect(L1.n, R2.p);
   connect(V.n, Ground.p);
end DAECircuit;
```

This model constitutes a complete Modelica model in the sense that it has as many variables as equations and can therefore be simulated.

Arrays

Modelica also has support for multidimensional arrays. It is for example possible to declare an array of ten Real numbers as follows:

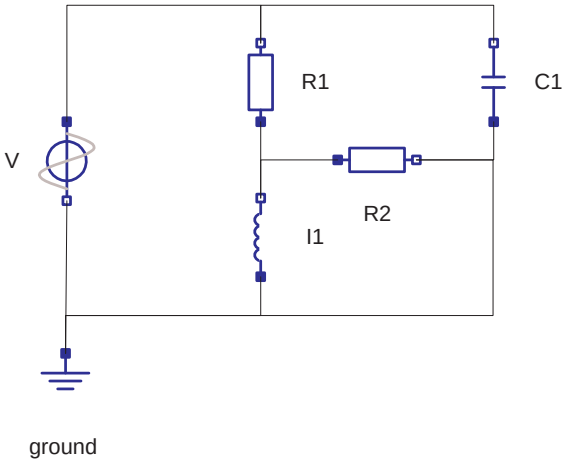


Figure 3.4. An electrical circuit resulting in a DAE problem.

```
model Array1
  Real[10] x=fill(0,10);
end Array1;
```

The `fill` function is a built-in function, here used to create an array of size 10 containing zeros.

However Modelica allows dimensionality on all components, not just the built-in types. It is therefore possible to make *arrays of components* and connect these together using a `for` equation. First, if we consider a model with 5 connected resistors that inherits a `OnePort`, giving the model two electrical connector instances, `p` and `n`, as shown graphically in Figure 3.5, it can be written as:

```
model FiveSerialResistors
  extends OnePort;
  Resistor r1,r2,r3,r4,r5;
equation
connect(p, R1.p)
connect(r1.p, r2.n);
connect(r2.p, r3.n);
connect(r3.p, r4.n);
```

```
connect(r4.p, r5.n);
connect(r5.n, n);
end FiveSerialResistors;
```

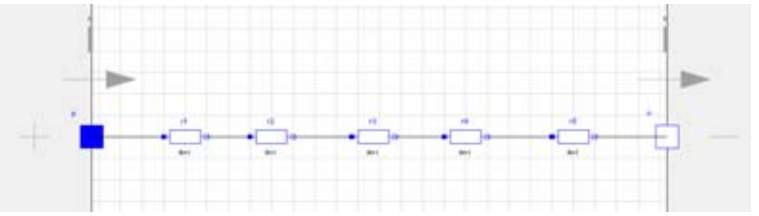


Figure 3.5. An electrical circuit resulting in a DAE problem.

By instead making an array of components and write a `for` equation that expands into corresponding `connect` equations a more flexible model with varying number of resistors is achieved:

```
model SerialResistors
  extends OnePort;
  Resistor Rx[n](each R=1);
  parameter Integer n=10;
equation
  for i in 2:n-1 loop
    connect(Rx[i].p, Rx[i+1].n)
  end for;
  connect(Rx[1].p, p);
  connect(Rx[n].n, n)
end SerialResistors;
```

The third line on the model above has an array of Resistor instances of size `n`, which is a parameter to the model, defined one line 4. This example shows that it is possible to use a variable before it is declared in Modelica. The `Resistor` declaration on line 3 has a modifier (`each R=1`). A modifier in Modelica can be a simple change of value on e.g. a parameter, or it can be more powerful constructs, such as changing a local class definition or changing the type of a declaration. In the modifier to the array of components (`each R=1`) the `each` keyword is used to give each component element of the array the modifier `R=1`. The equation section consist of a `for` loop that loops trough all interleaving connections making a connection between each subsequent pair of

components. The two ends of the chain of components is finally connected to the interface of the component (`n` and `p`).

Creating arrays of components in Modelica is a powerful modeling technique for describing discretized one-dimensional PDEs (Partial Differential Equations) using Modelica by making a discretization of the PDE into an ODE/DAE in its length dimension. This can be used in several different modeling domains, like for instance heat transfer, thermo-fluid and mechanical domains. This modeling capability is important in the context of this thesis, since it often leads to substantial opportunities for parallelization. The modeling technique can also be used for two- and even three dimensional PDEs that are discretized and translated into ODE/DAEs.

Functions

Functions are a fundamental part of the Modelica language. A Modelica function has a similar syntax as a class with input and output formal parameters and a function body starting with the algorithm keyword. A Modelica function can not contain equations and is side-effect free,i.e., a function is always produces the same outputs given the same input argument values. This property is important when performing equation optimizations later on, to be able to move the function call around when sorting equations, etc.

A Modelica function typically looks like this:

```
function scalarProd
  input Real u[:];
  input Real v[size(u,1)];
  output Real res;
protected
  Integer i;
algorithm
  res:=0;
  for i in 1:size(u) loop
    res:=res+u[i]*v[i];
  end for;
end scalarProd;
```

This function, performing a scalar product between two vectors, takes two vectors (`u` and `v`) of the same size as input. The size of the first vector `u` is arbitrary (using `':'` for any size), but the second input parameter has the same size as the first one. This function only gives one parameter as output value of the function, even though Modelica allows several output parameters. The function also has a local variable, defined in a `protected` section. The function

body is defined in an algorithm section. Here it consists of two statements, one assignment and a loop, which itself includes (the accumulative) statement of calculating the scalar product.

Modelica functions is a necessity for modeling in a convenient way. They can be used for table lookup, interpolation of measured data, linear algebra calculations, conversion functions, etc.

It is also possible to have external functions in Modelica. The current specification allows interfacing with Fortran77 and C. To declare an external function in Modelica the `external` clause is used, instead of the usual algorithm section:

```
function dgesv "Solve real system of linear equations "+
               "A*X=B with a B matrix"
  input Real A[:, size(A, 1)];
  input Real B[size(A, 1), :];
  output Real X[size(A, 1), size(B,2)]=B;
  output Integer info;
protected
  Real Awork[size(A, 1), size(A, 1)]=A;
  Integer ipiv[size(A, 1)];

  external "FORTRAN 77" dgesv(size(A,1), size(B,2), Awork,
                              size(A,1), ipiv, X,
                              size(A,1), info);
end dgesv;
```

This function declares the `dgesv` external function from the LAPACK library, which solves a system of linear equations on the form `A*X=B` for a matrix B.

Since Modelica functions are part of almost all models, they are important in the automatic parallelization tool as well.

3.1.3 Advanced Features for Model Re-Use

To enhance the component based modeling approach and to attract end users to re-use already developed classes, the Modelica language has several features for allowing reuse of classes.

- **Inheritance**
  A class can inherit equations and variables from a base class. Local classes are also inherited, making them accessible from the scope of the subclass.

- **Redeclarations**
  The type of an instance can be replaced by a new type using redeclarations. A redeclaration in a class can ,by using a redeclare modifier, be replaced by a new declaration with the same name. This is illustrated in the `circuit` example below.

  Redeclaration allows for later usages of a class to reuse the structure but replacing the type of the component. This is useful to increase the re-use by allowing to replace a component with a compatible (subtype of) one. For instance, by declaring a resistor component in a small circuit replaceable it can later on be redeclared to have the type `TemperatureResistor` which then is a refinement of the `Resistor` model giving a more accurate model. Thus, redeclarations is one way of controlling the level of detail of the model.

- **Modifications**
  A class or an instance of a class can be modified using a list of modifications, changing e.g. parameter values, constants, variables or even local classes in that class (see redeclarations above). However, the most common modification is to change a parameter constant value, like for example the inductance(L) of an electrical inductor.

Below these three language constructs are presented. The first two classes illustrated the inheritance mechanism using the `extends` construct. Both instances and equations are inherited in Modelica.

```
record ColorData
  Real red;
  Real blue;
  Real green;
end ColorData;

class Color
  extends ColorData;  // Standard Inheritance
equation
  red + blue + green = 1;
end Color;

class SmallCircuit
  replaceable Resistor R1;
end SmallCircuit;

class circuit                          // Redeclaration
```

```
  MiniCircuit tempcircuit1(redeclare TempResistor R1);
end circuit;

class SmallCircuit2
  Resistor R1(R=3); // Instantiation with modification
end C;
```

The next two classes show how redeclarations of variables are done. The circuit model redeclares the Resistor R1 component in the `tempcircuit` instance using a `redeclare` modifier. The last class in the example shows a standard parameter modification in Modelica, replacing the default for R by R=3, which is the most common usage of modifiers.

3.2 Modelica Compilation

The Modelica compilation process performed by a Modelica compiler basically consists of two parts, compiling to flat equation form followed by compilation and optimization of the equations. The first part translates the Modelica model code from its textual form to a set of equations and variables. The second stage performs transformations and optimizations on these equations to finally produce efficient code for simulation of the Modelica model. Figure 3.6 presents the different stages of a Modelica compilation.

3.2.1 Compiling to Flat Form

When the model has been described as Modelica source code it can be fed to a Modelica compiler. The Modelica compiler performs type checking, inheritance, instantiation, etc., breaking down the hierarchical object-oriented structure into a flat Modelica class. The flat model contains all variables defined in the model such as state variables, parameters, constants, auxiliary variables, etc., along with all equations of the model. These equations constitute the complete set of equations from all components and their subcomponents, along with equations generated from all the connect equations. The complete set of equations is either a system of Ordinary Differential Equations (ODE) or a system of Differential Algebraic Equations (DAE), depending on the model structure.

An ODE system on explicit form can be expressed as:

$$\dot{X} = f(X, t) \tag{3.5}$$

where X is the vector of all state variables. A DAE system on implicit form

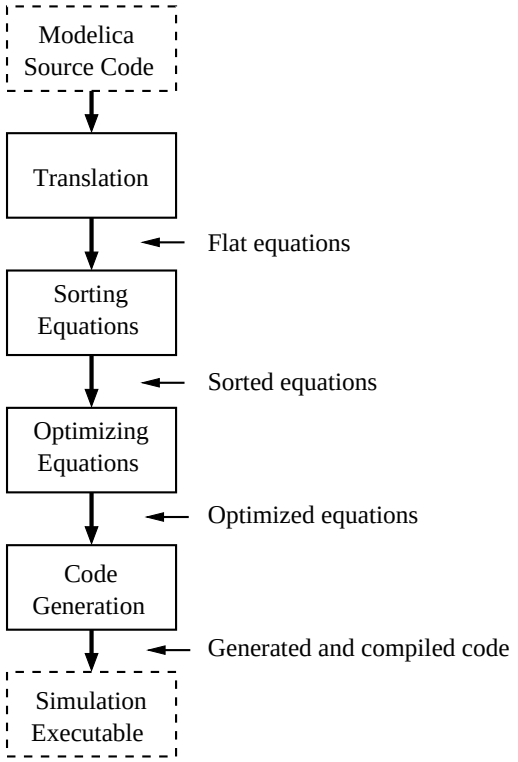


Figure 3.6. Compilation stages from Modelica code to simulation.

is expressed as follows:

$$\begin{aligned} g(\dot{X}, X, Y, t) &= 0 \\ h(X, Y) &= 0 \end{aligned} \tag{3.6}$$

where

- X is the vector of state variables
- $\dot{X}$ is the vector of derivatives of the state variables
- Y is the vector of algebraic variables
- t is the time variable

When the DAE system is on an implicit form the equation system first has to be converted to explicit form by solving for the derivatives and the algebraic

variables, if it is going to be numerically solved by an ODE solver. Alternatively, the DAE system can be solved directly by a DAE solver such as DASSL.

For example, the equations generated from flattening the `DAECircuit` model are shown in Figure 3.7. Equation 1-4 originates from the Resistor instance `R1`, 5-8 from `R2`, etc. Equation 23-25 are generated from the two connect equations connecting `C1`, `R1` and `V` together. The number of equations from the `DAECircuit` are 33, resulting in a differential algebraic system of equations, corresponding to Equation 3.6.

As just mentioned, a DAE system can also be solved by a DAE solver. In that case the model equations are often given on their residual form (i.e., as functions `f` and `g` in Equation 3.6 above. The DAE solver then internally solves all variables for one iteration. In this case, it can also have use for the Jacobian of the model equations as mentioned in Chapter 1.

3.2.2 Compilation of Equations

After the flattening phase of Modelica compilation, several optimizations on the resulting set of equations can be performed. By reducing the number of equations (and variables) in the problem, the execution time of the simulation will be reduced. Modelica compilers generating efficient code, e.g. OpenModelica [6] or Dymola [19], will perform several equation oriented optimizations, like reduction of the size of the equation system by removing algebraic equations, sorting of the equations¸ as well as using other techniques to reduce sizes of equation systems. Some of these different techniques are explained in briefly below.

Simplification of Algebraic Equations

Simplification of algebraic equations involves removing simple equations like $a = b$, where the variable b can be removed from the equation system and all references to it can be replaced by a. These variables are algebraic variables and most of these simple equations are generated from connect equations. Therefore they are quite many in the flat set of equations in a Modelica model, due to the object oriented way of modeling. Thus the size of the equation system can normally be substantially reduced by eliminating these simple equations.

The equations used for simplifications can also have a more complicated structure, like for instance $a = -b$ or $a = 1/b$. Many of the variables computed from those equations are not interesting to store in the simulation result either, making them completely redundant to calculate.

No.	Equation	Origin
1	$R1.v = R1.p.v - R1.n.v$	OnePort
2	$0 = R1.p.i + R1.n.i$	OnePort
3	$R1.i = R1.p.i$	OnePort
4	$R1.R * R1.i = R1.v$	Resistor
5	$R2.v = R2.p.v - R2.n.v$	OnePort
6	$0 = R2.p.i + R2.n.i$	OnePort
7	$R2.i = R2.p.i$	OnePort
8	$R2.R * R2.i = R2.v$	Resistor
9	$C1.v = C1.p.v - C1.n.v$	OnePort
10	$0 = C1.p.i + C1.n.i$	OnePort
11	$C1.i = C1.p.i$	OnePort
12	$C1.i = C1.C * der(C1.v)$	Capacitor
13	$I1.v = I1.p.v - I1.n.v$	OnePort
14	$0 = I1.p.i + I1.n.i$	OnePort
15	$I1.i = I1.p.i$	OnePort
16	$I1.L * der(I1.i) = I1.v$	Inductor
17	$Ground.p.v = 0$	Ground
18	$V.v = V.p.v - V.n.v$	OnePort
19	$0 = V.p.i + V.n.i$	OnePort
20	$V.i = V.p.i$	OnePort
21	$V.src.outPort.signal\_1 = V.src.p\_offset\_1$ $+(iftime < V.src.p\_startTime\_1then0else$ $V.src.p\_amplitude\_1 * sin(2 * V.src.pi$ $*V.src.p\_freqHz\_1 * (time - V.src.p\_startTime\_1)$ $+V.src.p\_phase\_1))$	SineVoltage
22	$V.v = V.src.outPort.signal\_1$	SineVoltage
23	$C1.n.i + R1.n.i + V.p.i = 0$	Connect
24	$R1.n.v = C1.n.v$	Connect
25	$V.p.v = C1.n.v$	Connect
26	$C1.p.i + Ground.p.i + I1.p.i + R2.n.i + V.n.i = 0$	Connect
27	$Ground.p.v = C1.p.v$	Connect
28	$I1.p.v = C1.p.v$	Connect
29	$R2.n.v = C1.p.v$	Connect
30	$V.n.v = C1.p.v$	Connect
31	$I1.n.i + R1.p.i + R2.p.i = 0$	Connect
32	$R1.p.v = I1.n.v$	Connect
33	$R2.p.v = I1.n.v$	Connect

Figure 3.7. The complete set of equations generated from the `DAECircuit` model.

BLT Transformation

The BLT transformation is a sorting technique to transform a system of equations into Block Lower Triangular form, which will identify blocks of inter-

dependent equations that form simultaneous subsystems of equations. This technique will (together with the index reduction technique discussed later in this chapter) transform a DAE system into an explicit ODE part that calculates the derivatives and an algebraic part that calculates algebraic variables.

This BLT sorting is done on a bipartite graph representation of all equations and variables by identifying strongly connected components in that graph. A bipartite graph is a graph that has two kinds/sets of nodes (vertices) and edges only going from a node in the first set to nodes of the second set. The vertices of the bipartite graph in the BLT sorting are of two types: equation vertices and variable vertices. The edges, which are connecting two vertices of different types, connect variables to equations. Each variable in an equation is connected to its equation node. As an example, we will consider a small Modelica model, `BLTExample` below:

```
model BLTExample
  Real s1,s2;
  Real x,y;
equation
 der(s1) = s2*s1;
 der(s2) = s1+x;
 x = 5 + y;
 y = time;
end BLTExample;
```

When performing the BLT sorting it is important to distinguish between what is considered to be known and what is solved for. The sought variables are the derivatives of the state variables and the algebraic variables, whereas the state variables themselves are assumed to be known (their initial values are known from initial conditions and their values at the next time step can be computed by the solver from the current value and the derivative) and so are parameters and constants and the built-in variable `time`.

Therefore, the bipartite graph representation of the equations from the model `BLTExample` becomes as presented in Figure 3.8. The first equation contains the unknown variables `der(s1)` since both `s1` and `s2` are known (they are both states). Therefore there is only one edge connecting equation node 1 with the variable node of `der(s1)`. The second equation has two connecting edges, one to `der(s2)` and the other to `x`. The later is present since `x` is an algebraic variable, it has no derivative operator among the set of equations. The third variable only contains the variable `y` which is also an algebraic variable, whereas the `time` variable is considered known.

The result from the BLT algorithm applied to the equations in the `BLTExample` model becomes in the form of assignments:

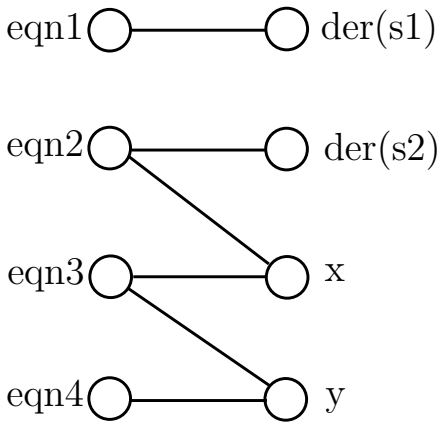


Figure 3.8. The bipartite graph representation of the equations in the `BLTgraphfig` example.

```
der(s1) := s2*s1;
y := time;
x := 5+y;
der(s2) := s1+x;
```

The BLT transformation gives a unique representation of the total system of equations as a number of subsystems of equations [29], where some of the subsystems can be solved analytically while others constitute systems of equations that in some cases can be solved analytically and in other cases need to be solved using numerical solvers.

The BLT algorithm is also responsible of solving the causality problem of Modelica models by sorting the obtained subsystems of equations in data dependency order. Due to the a-causal modeling capability of Modelica, this step is needed. Other languages which only have causal modeling capabilities, i.e., all equations are already written on assignment form, do not need to perform BLT transformation on their equations.

Index Reduction

Index reduction is a technique to reduce the *index* of a system of equations [57]. The index of a DAE corresponds to the maximum number of times an equation has to be differentiated such that the DAE can be written on explicit

form in the DAE system, which is referred to as the *underlying ODE*. Some numerical solvers can handle equation systems of index one or two. However, if the equation system has an index higher than what the chosen numerical differential solver can handle, the index needs to be reduced. For instance, models of mechanical systems typically are index 3 problems. Also if a uniform simulation code where solvers can be switched without recompilation is preferred, the index of the equation system must be reduced to zero, such that simple ODE solvers can be used. There are also other reasons for reducing the index, explained in Section 3.2.3 below.

As an example of the index reduction technique, lets consider a model of a pendulum of length L with a mass m in a gravitational field g parallel to the y axis, see Figure 3.9. If we use a Cartesian coordinate system, with variables x and y, we get the following equations.

$$m\ddot{x} = -xF/L \tag{3.7}$$

$$m\ddot{y} = -yF/L - mg \tag{3.8}$$

$$x^2 + y^2 = L^2 \tag{3.9}$$

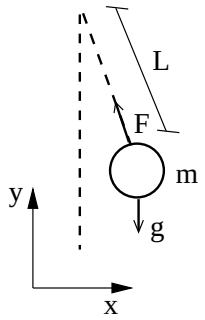


Figure 3.9. A pendulum with mass m and length L

Equations 3.7 and 3.8 constitute the Newtons law for the body with mass m, where F is the force along the pendulums direction. Equation 3.9 gives the constraint on the length of the pendulum.

This system of three equations with three unknowns (x, y and F) is a higher index problem, since in order to solve it we need to differentiate Equation 3.9. By differentiating it once we get the equation

$$x\,\dot{x} + \dot{x}\,x + y\,\dot{y} + \dot{y}\,y = 0 \tag{3.10}$$

which still is not solvable together with the equations 3.7 and 3.8, using an ODE solver. However, if we differentiate it once more (after simplification) we get

$$x\,\ddot{x} + \dot{x}\,\dot{x} + y\,\ddot{y} + \dot{y}\,\dot{y} = 0 \tag{3.11}$$

This equation together with equations 3.7 and 3.8 is a solvable system using an ODE or DAE solver.

The index reduction algorithm is an integrated part of the BLT sorting algorithm. Thus differentiation of equations can be performed while producing the BLT sorting.

Related to the index reduction algorithm is the dummy derivatives method []. When two or more state variables are present in an equation forming an algebraic constraint of the states, and this equation needs to be differentiated, the original algebraic constraint of the states is lost since we now have the differentiated version of the equation. This imposes a problem for the numerical solvers and the phenomenon of *drift* can occur. For instance, considering a simple electrical circuit with two capacitors in parallel will form an equation system where the two voltage drops over the respective capacitor (u_1, u_2 forms an algebraic constraint over the states ($u_1 = u_2$), see Figure 3.10. The index reduction algorithm will then replace this equation with the differentiated one ($\dot{(u_1)} = \dot{(u_2)}$). The numerical solution to this problem will then not constrain the two states u_1 and u_2 to be identical and the two states can drift apart.

The dummy derivatives method will in our circuit example choose one of the derivatives, e.g. $der(u_1)$ as a dummy derivative and transform it into an algebraic variable. Note that this will also change the state variable u_1 to an algebraic variable. As a result, the constraint equation is no longer an algebraic constraint between two states, since one of the states has been transformed into an algebraic variable.

Tearing of Equation Systems

Tearing of equation systems is a technique for optimizing the performance of solving equation systems [22]. The tearing technique breaks a system of simultaneous equations into two or more subsystems by inserting a new dummy variable (or several dummy variables) `v_new` that is assigned to the old value of a selected variable `v`, which is the variable used to break the equation system as a whole. By instead using the new variable in place of the old for the remaining equations of the system, constituting the second part, the complete system is torn apart. The main advantage is that solving the two parts separately and iterate until a stable solution is found can be less computational work compared to solving the complete equation system. Figure 3.11 illustrates how two subsystems are torn apart using a tearing variable, v.

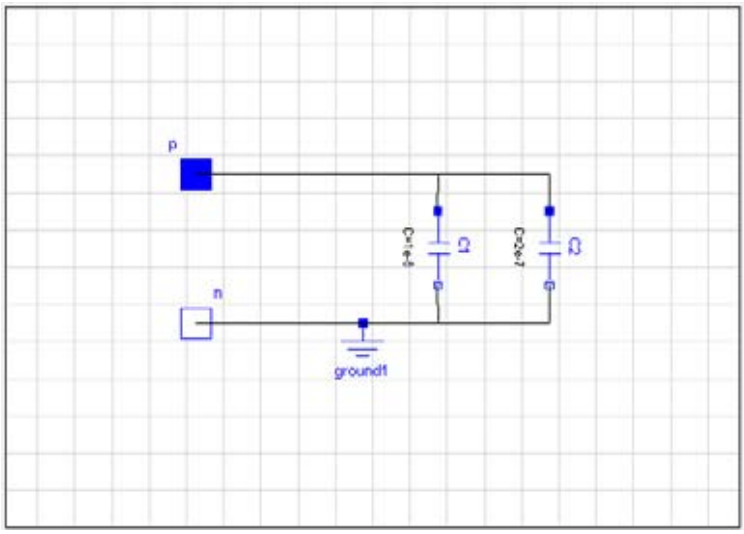


Figure 3.10. Two capacitors in series form an algebraic constraint between states.

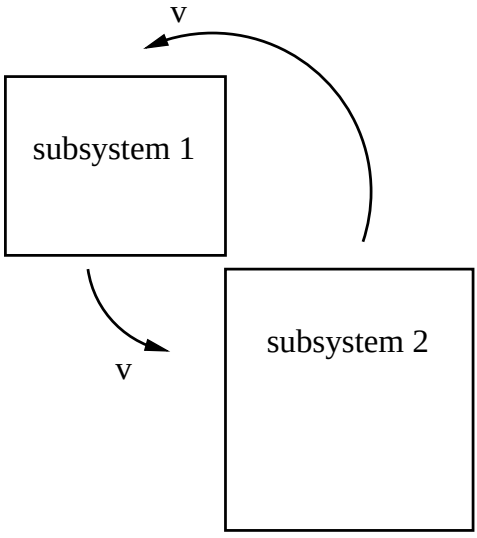


Figure 3.11. The tearing technique applied to a system of equations to break it apart in two subsystems.

However, an alternative to the tearing technique could instead be to use parallel solving techniques for the numerical solution of these equation systems. There exist both parallel implementations of numerical solvers for linear systems of equations and for non-linear systems of equations. Which of this method to use depends on the size of the equation system and on the efficiency of the parallel implementation of the linear systems solver.

Mixed Mode Integration

Mixed mode integration is a method for breaking apart an equation system and using several numeric solvers together instead of one [72]. For the fast dynamics of the equation system, i.e., the equations that are *stiff*, an implicit solver is used. A *stiff* equation system is a system containing both fast moving dynamics as well as slow dynamics, making the equation system much harder to solve numerically, thus smaller step-size and more computationally demanding solvers need to be used. For the slow states of the system, a more efficient explicit solver can be used, with longer step sizes because of the slow dynamics. When using an implicit solver, a non-linear equation system has to be solved in each time step, which is time consuming. However, by only using an implicit solver on the parts that really need it, i.e., on the stiff parts of the system, the non-linear equation system is reduced in size, and a speedup is achieved. This method also has a potential for revealing more parallelism in the code since the parts that are broken apart will become more independent of each other.

Inline Integration

Inline Integration is an optimization method that inserts an inline expansion of the numerical solver into the equation system parts [21]. By doing so, and again performing BLT transformations, etc., a substantial speedup in simulation time can be achieved. As an example, consider a simple equation:

$$der(x) = -x + y; \tag{3.12}$$

If we use the backward difference formula of order one, we get:

$$x_{t+1} = x_t + h * (-x_t + y_t) \tag{3.13}$$

This equation is added to the equation system as an algebraic variable, i.e., x is no longer a state variable in the new DAE system.

The inline integration method has successfully been combined with mixed mode integration to further reduce simulation time [72]. There are cases when inline integration does not work well. For instance, for stiff systems where

small step sizes and/or more robust solvers are required, the inline integration method does not always work.

3.2.3 Code Generation

When all desired optimizations have been performed on the system of equations, code is finally generated. The generated code corresponds to the calculation of the right hand side of Equation 3.5, i.e., the calculation of f. For the DAE case, i.e., equations on the form as given in Equation 3.6, the subsystems of equations that are on explicit form are generated in the same way as for the ODE case, with assignment statements of arithmetic expressions to variables. But for the subsystems of equations which are on implicit form this does not work. Instead, code that solves the subsystems numerically is generated and/or linked with a numerical solver.

In order to freely choose between different solvers, ranging from ODE solvers like explicit or implicit Euler to advanced multi-step solvers like DASSL, code for calculating the underlying ODE is generated. This means that the simulation code will contain a function in e.g. C or Fortran for calculating the derivatives given the state variables and algebraic variables. As explained above, this function is not always purely "mathematical" and can contain subsystems of equations that are solved using numerical solving techniques. The index of the model equations have also been reduced to zero such that an ODE solver can be used. This means that the capabilities in more advanced solvers like DASSL of solving e.g. index one problems is not used to its full extent. However, the advantage of reducing the index to fully exploit the choice of numerical solver is often more important. But it is even more important for other reasons.

For hardware in the loop real time simulations it is of uttermost importance to be able to use a simple fixed step ODE solver. In real time simulations the solver must be able to calculate new values for the states within a fixed deadline. Therefore, a fixed step size solver must be used, since multi-step solver often can not guarantee a deadline of the calculations of the states. By reducing the index in combination with other equation optimization techniques such as tearing and inline integration, etc., the performance of the simulation code also increases. This makes the approach more advantageous for hardware-in-the-loop real time simulations.

3.3 Simulation

As already mentioned, the simulation process for Modelica models, based on a DAE formalism, involves numerical solution of differential-algebraic equa-

tions. Even though certain simple differential equations could be solved symbolically, such simple equations seldom occur in real applications. Therefore, a generic method using numerical solvers for differential equation solution is used. There exist many different solvers for numerical solution of differential-algebraic equations, ranging from the simple explicit Euler to advanced solvers having features like variable step size, etc.

The explicit Euler method for the numerical solution to an initial value problem calculates time varying variables at time $t + h$ by using the variable values and their derivatives at time t. The time increment h is called the step size, see Figure 3.12. The explicit Euler uses the Equation 3.5 directly to calculate the values at the next time step. Equation 3.5 is presented again below for convenience for the reader.

$$\dot{X} = f(X, t) \tag{3.14}$$

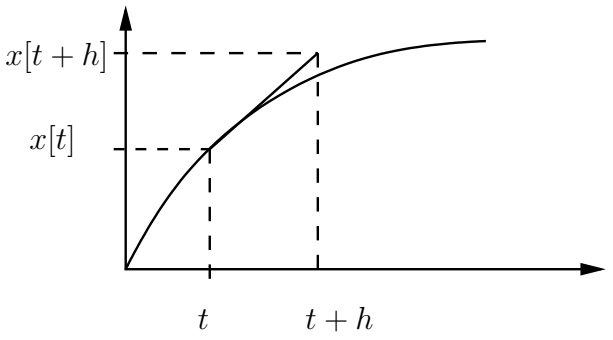


Figure 3.12. Solution of differential equation using explicit Euler.

Below follows pseudo code for the explicit Euler method. The simulation time is incremented by the fixed step value `h` until the stop time of the simulation is reached:

```
for ( t = start ; t <= stop; t += h) {
   X[t+h] = X[t] + h*f(X,t);
}
```

The *implicit Euler* approach instead calculates the next step by an equation which involves solving a linear equation for X_{t+1}, i.e., the function call f contains time $t + 1$.

Numerical solvers can take the value of the state variables at several points in time as input to the numerical solution. If there are more than one point

in the solution scheme, the numerical solver is called a *multi-step* solver, i.e., at each calculation it take values from several earlier steps into consideration.

Numerical solvers have different stability criteria and different error estimates, depending on their approximation scheme. For example, the simple explicit Euler method is not very stable and has an error estimate of $O(h)$.

Figure 3.13 presents some of the most common solvers, including error estimates.

Solver	*Approximation*	*Error*
Explicit Euler	$x_{n+1} = x_n + hf(\dot{x_n}, x_n, t)$	$O(h)$
Implicit Euler	$x_{n+1} = x_n + hf(\dot{x_{n+1}}, x_{n+1}, t)$	$O(h^2)$
Runge Kutta	$x_{n+1} = x_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4, t)$ $O(h^4)$ $k_1 = hf(\dot{x_n}, x_n, t)$ $k_2 = hf(\dot{x_n} + \frac{k_1}{2}, x_n + \frac{h}{2}, t)$ $k_3 = hf(\dot{x_n} + \frac{k_2}{2}, x_n + \frac{h}{2}, t)$ $k_4 = hf(\dot{x_n} + k_3, x_n + h, t)$	
DASSL	(Uses Backward Difference Formula, BDF)	-

Figure 3.13. A few common numerical solvers for differential equations.

3.4 Simulation Result

The simulation results in a number of functions of time, each function being represented by a data set, i.e., its value at a number of points. These functions represent the numerical solution to the (time varying) variables of the simulation model. These sets are typically used for analysis of the model, by for instance visualization like 2D or 3D plots, or perhaps by filtering or processing the data by e.g. Fourier analysis, low pass filtering, etc.

For large simulation models the amount of data generated by the simulation can be huge. Storing all simulation data from the models might not be possible or even needed. The Modelica compiler will limit the number of needed variables by performing alias analysis to detect which variables are equal. Furthermore, it can also be sufficient to only store the state variables and their derivatives from a simulation to reduce the number of variables in the simulation result. However,the size of the simulation results might still be a problem. For such occasions, lossless compression techniques can be employed to compact the simulation result to a manageable size [23]. This is especially important in the context of automatic parallelization of large models.

3.5 Summary

Modeling and simulation has been used for decades to gain a better understanding of physical systems by building mathematical models of these systems and then performing simulations on these models. New modeling languages such as Modelica advances the modeling capabilities by introduction of object orientation, a-casual modeling, efficient compilation techniques and fast simulation. This expands the modeling and simulation area and opens up for new kinds of modeling needs, such as full system simulations.

The Modelica modeling language enables a modeler to graphically compose highly complex models by composition of predefined components defined in standard libraries, or to write or components as a library developer. In either case, the language uses object oriented features such as inheritance, class parameterization and object instantiation to enable component reuse to speed up the modeling process. Thus, the modeling process is substantially faster in a language such as Modelica, since a modeler does not need to write all model code from scratch.

Once the model is finished, the process of performing simulations and validating the model starts. From the Modelica code a flat set of equations is generated. These equations are translated into efficient C code which is connected to a numerical solver for differential equations. This code is then executed to perform the simulation of the model. After that follows the process of analyzing the data from the simulations, perhaps using visualizations such as 2 dimensional plots or 3D animations.

Most of this chapter relates to this research work. The Modelica language is itself extending modeling and simulation to areas such as full system simulation, where a complete system is simulated, often consisting of several modeling domains such as e.g. electrical, mechanical and thermal. Due to the large size of such full system models it is is especially relevant to achieve faster simulations.

Many of the presented equation optimization techniques also help in speeding up the execution of simulations and some of the techniques, such as inline-integration, tearing, and in fact also the basic BLT sorting help us to identify and increase the amount of parallelism the resulting simulation code.

Chapter 4

DAGS - a Task Graph Scheduling Tool in Mathematica

This chapter present DAGS, a task graph package for the computer algebra and programming language Mathematica [86]. This tool has been used as an experimental framework for investigating scheduling, clustering and task merging algorithms. Through this thesis work, several prototypes have been made. After the first prototype tool, DSBPart, presented in detail in chapter 5, the implementation of the DAGS tool started. Later on, the prototypes implemented in the DAGS tool was implemented in OpenModelica in the ModPar tool, which is presented in chapter 6. The major reason for presenting this prototype in this chapter is that several implementation details that also affect the ModPar tool is presented in this chapter.

4.1 Introduction

Throughout this thesis work there has been a need for experimentation on scheduling and clustering algorithms on task graphs. Therefore an experimental framework where such algorithms could be implemented in a fast and efficient manner was desired. For this purpose the computer algebra system Mathematica, which also is a very excellent functional programming language was used. Mathematica provides an interactive and incremental functional programming environment which enables fast prototyping of algorithms like scheduling and clustering algorithms for task graphs, much like what is pro-

vided in other programming environments for functional languages such as lisp or Haskell [].

The user interface in Mathematica is a hierarchical structured document called notebook consisting of *cells*, where program code can be mixed with text and graphics. Cells can be *evaluated* which means that the program code in the cell is sent to the computational kernel in Mathematica for evaluation, like issuing a command in any shell based computational tool (like Matlab or Maple). The mixture of program code, text and graphics in a notebook makes Mathematica a very powerful literate programming environment. Figure 4.1 shows the Mathematica notebook containing the program code and examples for the DAGS package. The figure shows a test section of the document for the implementation of the TDS [17] algorithm in the framework. The first program code cell in the `Testing` section calls the function `BuildDag` which takes a list of edges. Each edge is represented as a list of two task identifiers. The second cell evaluates and returns the variable `Nodes` to verify that the `BuildDag` function succeeded and built the task graph with in this case ten nodes, 1 to 10.

Then follows two cells for setting up the execution cost and communication cost for the TDS algorithm, and after that the TDS algorithm is called.

The final cell shows which processors each task is assigned to using the function `TDSAlloc`.

4.2 Graph Representation

There already exist a graph package in Mathematica called Combinatorica [62] that allows several different graph representations such as adjacency list and incidence matrices. For our purposes an adjacency list representation would be preferable especially for task merging algorithms, see Chapter 7. The reason for this is that when having to delete and add edges and nodes the adjacency list representation of a graph is best suited.

However, the Combinatorica package uses a standard Mathematica list for each adjacency list and such list must be copied every time an element is added, which makes this implementation quite expensive for large graphs. Therefore there was a need for a more efficient implementation using linked lists in Mathematica.

Below are some Mathematica code for linked lists and standard lists (represented as arrays in Mathematica). First a standard Mathematica list:

```
lst={1,2,3,4};
```

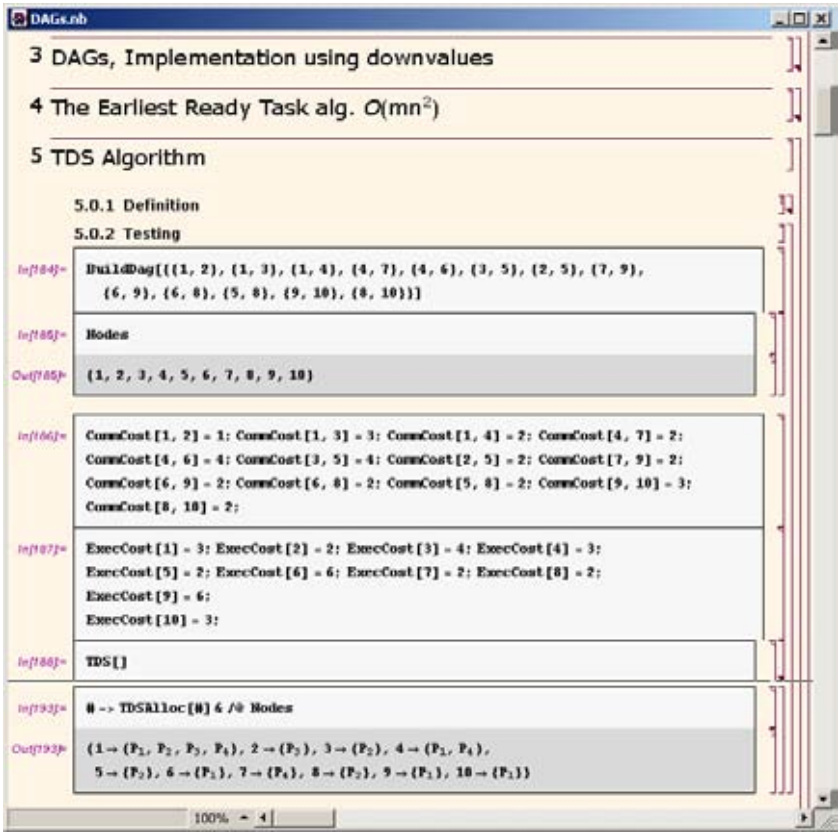


Figure 4.1. The Mathematica notebook for the DAGs package.

If we want to add an element to the list we use Append which copies the entire list:

```
lst2=Append[lst,5];
```

As an alternative we can use linked lists, first creating an empty list.

```
lst={};
```

and adding an element by:

```
lst={1,lst};
```

Lets add one more and then traverse the list to extract the elements using the built in function Flatten:

```
lst={2,lst};
elts=Flatten[lst];
```

`elts` will then have the value

```
{1,2}
```

We have implemented a graph representation using linked lists as adjacency lists and giving each vertex of the graph a unique integer as identifier. Attributes of the graph as set by defining functions on these identifiers. For instance, to set the execution cost of task 1 to 4.5 we evaluate:

```
ExecCost[1]=4.5;
```

giving the function `ExecCost` a value for task 1. The advantage of using functions as attributes is that it is efficient to retrieve the values later on in Mathematica and that we can easily issue default values. For instance, if we would like to set all execution costs of all nodes in the graph to one we can evaluate:

```
ExecCost[_]=1;
```

The underscore here means that any argument to the function will be matched, thus always returning cost one. However, the function definition for argument 1 is still valid, returning the cost 4.5 for task 1. This is achieved by Mathematica having overloading of functions. A function can thus have several definitions,

and when evaluating a function call in Mathematica the function with the most specific matching pattern will be used. Thus, for our execution cost example above if the execution cost for the Integer value 1 (i.e., task 1 of the task graph) is evaluated the result will be the first function definition which has the most specific pattern that matches.

4.3 Implementation

The implementation consist of several parts, the graph primitives, scheduling algorithms, clustering algorithms and the task merging algorithm presented in Chapter 7.

4.3.1 Graph Primitives

The implementation contains the graph representation itself, along with primitives for retrieving information about the graph. Table 4.2 gives a list of these primitives.

Name	*Description*
`AddEdge[{`*a*`,`*b*`}]`	Add an edge from node *a* to node *b*
`AllEdges[]`	Return a list of all edges
`BuildDag[`*lst*`]`	Builds a new graph given a list of edges
`Children[`*node*`]`	Return a list of all immediate successors of *node*
`ClearDag[]`	Clears the graph definition removing all edges and nodes
`EdgeQ[{`*a*`,`*b*`}]`	Returns `True` if edge exists, otherwise `False`
`Nodes`	Contains a list of all nodes of the graph
`Parents[`*node*`]`	Return a list of all immediate predecessors of *node*
`RemoveEdge[{`*a*`,`*b*`}]`	Remove edge from graph

Figure 4.2. Graph primitives in the DAGs package.

4.3.2 Scheduling Algorithms

In the DAGs package we have implemented several scheduling algorithms, such as the TDS algorithm and the ERT algorithm. For the ERT algorithm we also made a simple Gannt chart layout of the schedule, see Figure 4.3.

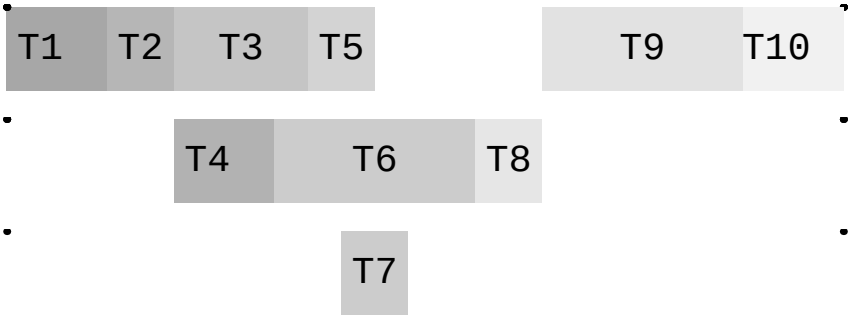


Figure 4.3. DAGs notebook showing a Gannt schedule from the ERT algorithm.

Such graphical objects can easily be created in Mathematica by using a set of predefined graphical objects, such as lines, circles and polygons.

4.3.3 Clustering Algorithms

The DAGs package also includes an implementation of the Dominant Sequence Clustering (DSC) algorithm, see Section 2.5.3. [1] The implementation contains both the simplified version of the DSC algorithm, which is called using the function `DSCI`, and the final version as described in Yang's thesis [88], called using the function `DSCFinal`.

Figure 4.4 shows the notebook cells for testing of the `DSCFinal` function using the test graph from Yang's thesis.

4.3.4 The Task Merging Algorithm

The perhaps largest individual part of the DAGs package is the experimental implementation of the task merging algorithm using graph rewrite systems, see Chapter 7. The method was first implemented in this framework and thereafter translated into C++ for use in the ModPar tool, see Chapter 6.

[1] We made some experimental attempts to introduce task replication into the DSC algorithm, but these attempts were abandoned since this could not be done in an intuitive and controlled way. I.e., without affecting the complexity of the algorithm and without having a balanced task replication scheme.

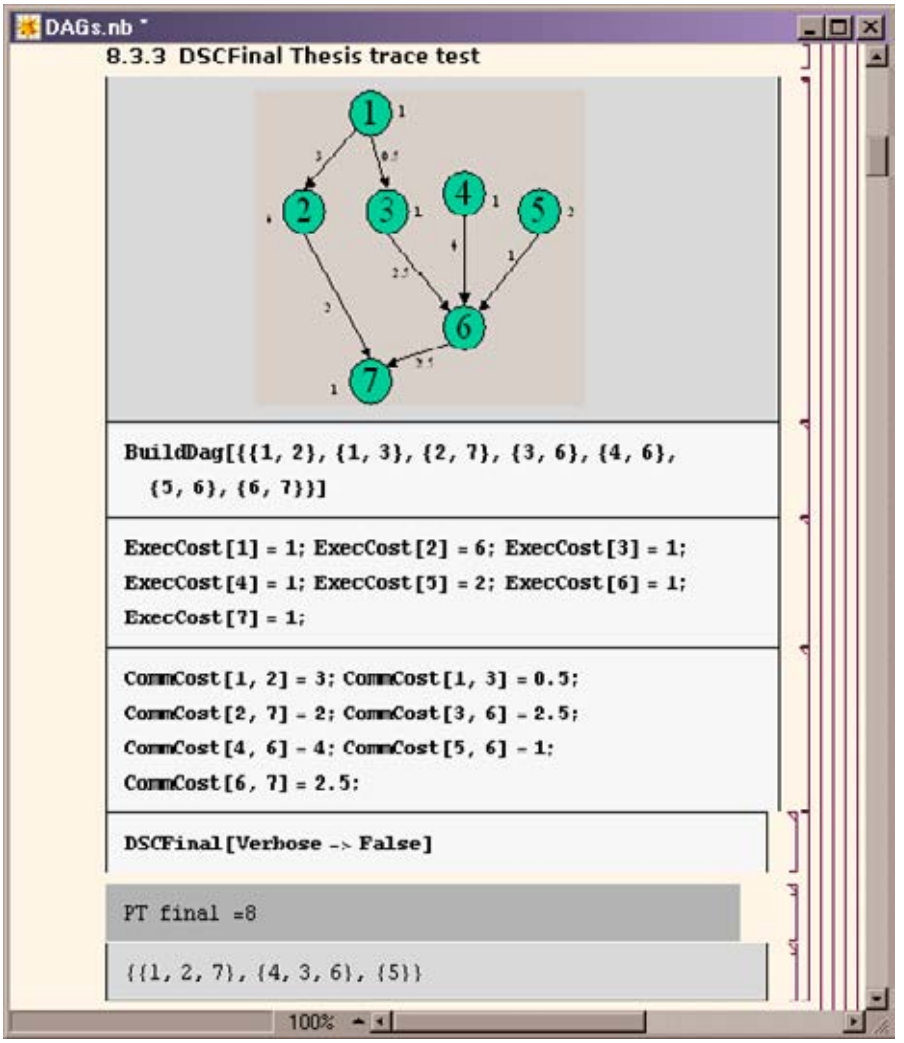


Figure 4.4. The notebook cells for testing the DSC algorithm.

4.3.5 Loading and Saving graphs

The implementation also contains a set of interfacing functions for loading and saving graphs in different formats. The main usage of these functions is to be able to import existing graphs and also to export graphs to graph visualization

tools. Figure 4.5 shows the available commands for importing and exporting of graphs in the DAGs package.

One import function that is available is importing from the Standard Task Graph Set (STG) [77], which consist of a large set of randomized task graphs of different sizes. It also includes a few task graphs from real application examples. The STG comes in two variants, one with communication costs and one without.

Name	*Description*
`GenerateDOT[`*filename*`]`	Output task in Graphviz format
`GenerateVCG[`*filename*`]`	Output task in VCG format
`ReadSTG[`*filename*`]`	Read the graph using STG format from file
`ShowGraph[`*filename*`]`	Shows a graphical representation of the graph using the Graphviz toolset

Figure 4.5. Graph primitives in the DAGs package.

4.3.6 Miscellaneous Functions

There are also some additional miscellaneous functions available in the DAGs package. Some of these are needed for the scheduling and clustering algorithms, like the `tlevel` function whereas others are used for the testing of algorithms, such as the butterfly task graph building function `BuildButterFlyDAG`. Figure 4.6 shows a summary of these functions.

Name	*Description*
`BuildButterFlyDAG[`*n*`]`	Generate a task graph for a butterfly calculation of order *n*
`CalcualteGranularity[]`	Calculates the granularity of a task graph, using `CommCost`, `ExecCost`, `B` and `L` attributes
`tlevel[`*n*`]`	Calculate the top level of a node

Figure 4.6. Graph primitives in the DAGs package.

Lets look at the definition of the `BuildButterFlyDAG` function.

```
BuildButterFlyDAG[i_Integer] :=
  If[i == 1,
    Return[{{1, 3}, {1, 4}, {2, 3}, {2, 4}}],
    Module[{b1, b2},
      b1 = BuildButterFlyDAG[i - 1];
```

```
    b2 = BuildButterFlyDAG[i - 1];
    Return[MergeButterFly[i, b1, b2]]
    ]
  ]
```

The function is recursive building a butterfly task graph of order n by combining two butterfly task graphs of order $n-1$ in the `MergeButterFly` function. The base case is a butterfly of order 1 which consists of the graph shown in Figure 4.7(a). For instance, the butterfly task graph of order two is constructed by combining two task graphs of order 1 and add some more nodes for the next level of the butterfly graph, along with connecting edges between the new level and the two sub-task graphs of the underlying level (of order 1). All this is performed in the `MergeButterFly` function, defined as:

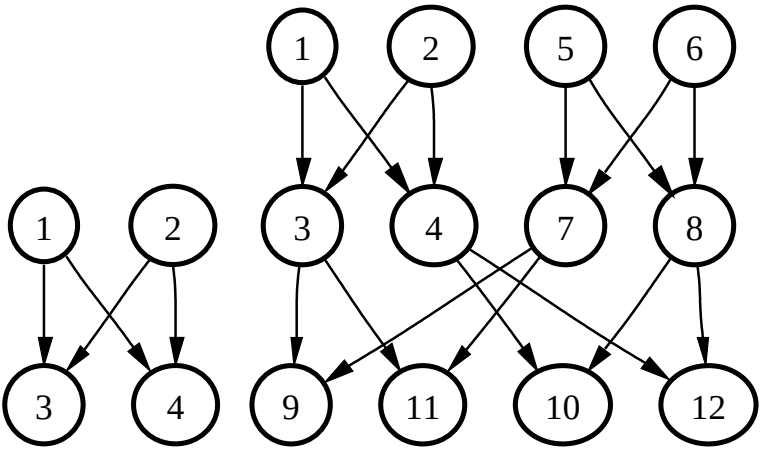
```
MergeButterFly[i_, b1_, b2_] :=
  Module[{b1n = b1, b2n = Max[b1] + b2, b3,
          b1top, b2top, edges1, edges2},
    b3 = Range[(i)2^(i) + 1, (i)2^(i) + 2^i];
    b1top = Take[Sort[Union[Flatten[b1n]]], -2^(i - 1)];
    b2top = Take[Sort[Union[Flatten[b2n]]], -2^(i - 1)];
    edges1 = Thread[List[Join[b1top, b2top], b3]];
    edges2 = Thread[List[Join[b2top, b1top], b3]];
    Return[Join[b1n, b2n, edges1, edges2]];
    ]
```

The result of the `MergeButterFlyDAG` is a list of edges that can be used to construct the graph using the `BuildDag` function.

4.4 Results

We have implemented an efficient task graph framework that can interactively and efficiently handle large task graphs in the Mathematica framework. Figure 4.8 shows the execution time for building graphs of different sizes (executed on a laptop PC with 2GHz Pentium Mobile processor), clearly demonstrating that the complexity of building a graph is linear in time. Mathematica provides an interactive environment where incremental programming can speed up prototyping of algorithms and computer program as well as providing good documentation support using literate programming.

Out task graph tool uses an efficient graph representation based on linked lists, the so called adjacency list graph representation. It also allows attributes



(a) A butterfly task graph of order 1

(b) A butterfly task graph of order 2

Figure 4.7. Butterfly task graphs generated using the DAGs package.

of tasks to easily be defined using the powerful function definition and pattern matching concept of Mathematica, allowing the setting of attributes for individual nodes or even groups of nodes, etc.

Number of edges	*Execution time*
100	0.01 s
1000	0.06 s
10000	0.531 s
100000	4.28 s
1000000	42.6 s

Figure 4.8. Execution times for building graphs using the `BuildDag` function on a 2GHz Pentium Mobile PC.

4.5 Summary

In this chapter we have presented a graph toolkit for task graphs and scheduling and clustering algorithms working on such graphs implemented in the Mathematica tool. This toolkit enables us to fast prototyping and experimentation but still being able to write highly efficient (for interpreted code)

algorithms in a functional programming language.

We have implemented several scheduling and clustering algorithms in this framework and also have several possibilities of importing and exporting task graphs to formats like STG, VCG and Graphviz.

The most important experimentation and prototyping usage has been done when developing the task merging algorithms presented in this thesis. These algorithms were first developed in this framework.

Chapter 5

DSBPart - An early parallelization Tool Prototype

DSBPart is our first prototype developed for automatic parallelization of Modelica simulations. It started as a master thesis project and was continued in this PhD project. This chapter describes the tool and the parallelization approach taken. It also presents results and lessons learned during its development. Parts of this chapter have been published e.g. in [4].

5.1 Introduction

Earlier work on parallelization of simulation code from mathematical models started at the Programming Environments Laboratory (PELAB) at Linköping University for models in a language called ObjectMath [47, 83, 52]. One of the findings from this research was that all potential parallelism initially should be considered, resulting on the building task graphs at the sub-equation level [2]. The ObjectMath language effort was after its fourth generation transferred into the joint effort of the next generation modeling language Modelica. The Modelica language is based on experience and ideas from a large number of earlier languages, including ObjectMath, Omola, Dymola, etc.

The automatic parallelization of Modelica models started as a master thesis work in 1999 by an effort to automatically parallelize the simulation code from the commercial tool Dymola [19]. This prototype was named DSBPart which

is an abbreviation for DSBlock[1]partitioning. This work was never finished and the prototype served as a starting point for this PhD work.

5.2 Overview

Figure 5.1 gives an overview of how the parallelization tool DSBPart is used. It also shows the normal compilation of Modelica models to sequential simulation code. The input to the parallelization tool is a sequential program consisting of automatically generated C-code with macro calls, the so called DSBlock code. The format of this code is presented in detail in section 5.3. Internally, the tool builds a task graph, applies a scheduling algorithm and generates code for parallel execution. These different phases of the parallelization tool are shown in Figure 5.2. The code is compiled and linked with platform independent message passing libraries and numerical solver libraries. The executable can then be executed on a parallel computer.

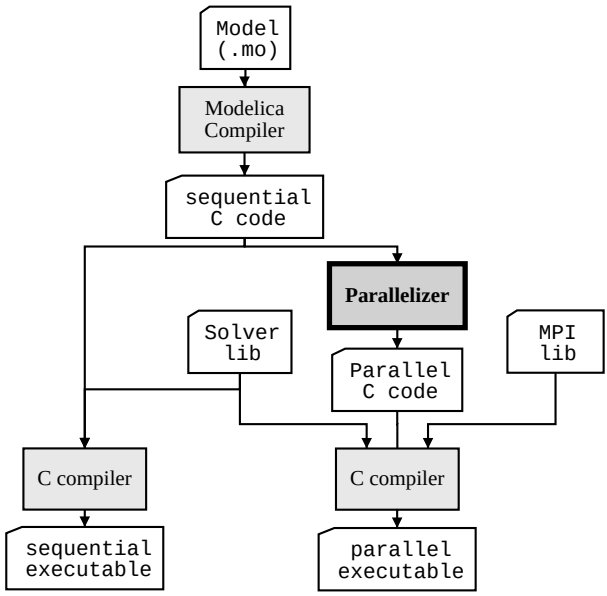


Figure 5.1. An overview of the automatic parallelization tool and its environment.

[1]DSBlock is the intermediary C code format originally developed at DLR used by Dymola at that time.

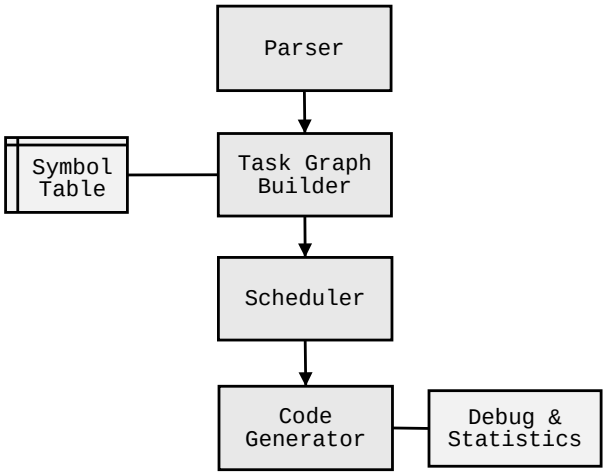


Figure 5.2. The internal architecture of the DSBPart tool.

The following sections present the different parts of the tool in some detail.

5.3 Input Format

The input to the tool is the sequential simulation code generated from a Modelica compiler, in our case Dymola [19] producing the DSBlock code format. The simulation code performs calculation of the right hand sides, i.e., the function f in Equation 3.5 or solving $\dot{X}$ om the equation system in Equation 3.6 on page 61. The code is generated C-code with macros. The syntax of the C-code is a limited subset of the C language. Therefore, writing a parser for parsing the C-code is less complicated than writing a complete parser for the C programming language. However, each macro must also be parsed by the tool.

The subset of the C language that is needed is:

- **Expressions**
  Most parts of the generated simulation code consist of expressions built of arithmetic operations on constants and variables. The code typically also contains function calls, e.g. the standard math functions like `sin`, `cos` and `exp` are frequently used.

- **Statements**
  The statements found in the code are mostly assignment statements,

where scalar variables or array elements with a constant index are assigned large expressions. However, some of the macro calls need also be treated as statements. For instance the macro call for solving a linear system of equations (`SolveLinearSystemOfEquations`) is treated as a statement, even if it after macro expansion corresponds to a complete block of statements.

- **Declarations**
  The code can also contain declarations of temporary variables, used in the expressions and as targets for the assignment statements.

- **Blocks**
  In some cases, the code contains blocks of statements, i.e., a new scope is opened with the '{' character and closed with the '}' character, with a sequence of statements in between. The reason for having such local blocks is for instance to be able to use local variables.

- **Miscellaneous**
  The simulation code can also contain some miscellaneous C language constructs, like `if` statements.

The parallelization tool uses Bison [18] to generate a parser. The lexical analyzer is generated by the Flex tool [61]. The parser calls a set of functions for building the task graph, described in the next section.

5.4 Building Task Graphs

A fine grained task graph is built, while parsing the input file. For each arithmetic expression, function call, macro statement, etc. a task node is created. Data dependence edges are created between pairs of two tasks from each definition (i.e., assignment) of a variable in one task to the corresponding use of the variable in another task.

As an example, we use a simple model with only one variable:

```
model SmallODE
  parameter Real a=3.5;
  parameter Real b=2.3;
  Real x;
equation
  der(x)=-a*x+b/(time+1);
end SmallODE;
```

Parts of the generated simulation code in DSBLock format for the `SmallODE` example appears as follows:

```
#define der_x Derivative(0)
#define x State(0)
#define a Parameter(0)
#define b Parameter(1)
...
 der_x = a * x + divmacro(b,"b",Time+1,"Time+1");
...
```

From this code a task graph as shown in Figure 5.3 is built.

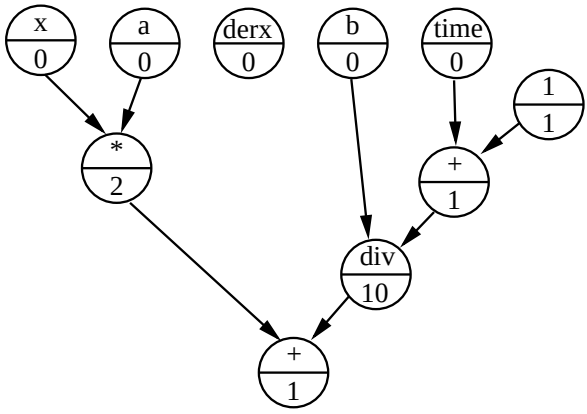


Figure 5.3. The task graph produced from the simulation code for the `SmallODE` example, on page 90.

First, when the definitions of the variables in the code, (`der_x, a` and `b`) are parsed, a task for each variable definition is created. These tasks are definition nodes, hence their execution cost is zero. A symbol table keeps track of the tasks that define the value of a given symbol. For instance, the variable name `der_x` points to the define task for the variable `der_x`, see Figure 5.3.

When the statement (that assigns to the variable `der_x`) is parsed, task nodes for the division macro, the two additions and the multiplication are created. For instance, when the multiplication task is created, the two operands are looked up (i.e., the definition tasks for `a` and `x` are accessed) and edges between the operand tasks and the multiplication task are created. The symbol table entry to the variable `der_x` is updated, so that subsequent reads of the same variable will connect data dependence edges to the new task instead of

the definition task of the der_x variable. For scalar values the communication costs associated with the edges are set to the cost of sending one scalar value between processors. In the example we use the cost of 100 units, e.g could be 100 microseconds, for a communication of one variable.

5.4.1 Second Level Task Graph

The task graph built while parsing is not suitable as input to a scheduling algorithm. There are several reasons for this. First, many scheduling algorithms assumes a single entry, single exit task graph. This means that the task graph should only have one entry node (a node without any predecessors) and one exit node (a node without any successors).

Second, since a lot of definition nodes are created, one for each variable defined in the simulation code, and these nodes have no computational cost, they could preferably be joined into one task. This task could be the single entry task of the task graph.

Third, some constructs in the simulation code must be sequentialized and performed atomically as one unit of execution without being divided. Modelica when-statements is an example of such constructs. The following model illustrates the problem:

```
model DiscreteWhen
   discrete Real a(start=1.0);
   discrete Integer b;
   Real x(start=5);
equation
   der(x) = -x;
   b = integer(x);
   when (b==2) then
     a=2.3;
     reinit(x,4);
   end when;
end DiscreteWhen;
```

The DiscreteWhen model has two discrete-time variables a and b and one continuous-time variable x. A discrete-time variable only changes value at certain points in time, at events, whereas continuous-time variables may change at any point in time. The when equation is a discrete event handling construct in the Modelica language. It triggers at a specific event, specified by the code b==2, i.e., when the discrete variable b equals two. At the event, two instantaneous equations become active, resulting in the execution of two

corresponding statements to solve these equations. The first one sets the discrete variable `a` to the value 2.3, and the second one reinitializes the state variable `x` with the new value 4. The generated simulation code for the when equation has the following structure[2]:

```
beginwhenBlock
whenModelicaOld(b0_0 == 2, 0)
  a0_0 = 2.3;
endwhenModelica()
endwhenBlock
```

To solve these three problems related to the task graph built from parsing the simulation code a second task graph is built, where each node in the new task graph can contain one or several tasks from the first task graph. Figure 5.4 illustrates the relationship between the two task graphs. The first task graph contains all arithmetic operations, function calls, etc., and the second one is built by clustering together tasks from the first task graph. This is an important reason for why the second task graph is built. For some scheduling algorithms to work well, the granularity of the task graph, defined in Equation 2.13, must have a low value. This is for instance true if the scheduling algorithms only considers linear clustering, i.e. never scheduling several immediate predecessors of a node on the same processor. This can be achieved by running grain packing or clustering algorithms on the original task graph, resulting in a new task graph which is coarser than the original one.

5.4.2 Implicit Dependencies

The simulation code also contains implicit dependencies, not visible by parsing the simulation code with its macros unexpanded. Such code can for instance be initialization macro calls for matrices and vectors used by code sections for solving a system of equations. This means that information about these special macros, and their implicit dependencies, must be known by the tool, and that additional pseudo dependencies must be added to the task graph.

An example of implicit dependencies is given in Figure 5.5. The code fragment solves a non-linear equation system. In this case the implicit dependencies makes the whole code fragment sequential. For instance, the first macro (`NonLinearSystemOfEquations`) declares several variables used in the macros that follow and must hence be first in the code fragment. The sequentialization of the code fragment is also motivated by the opening of a new scope with the '{' character, which forces the tasks inside the scope to be executed atomically.

[2]The when equation is split internally by Dymola and the reinit equation is treated elsewhere in the code. Therefore the reinit code is not present in the generated code.

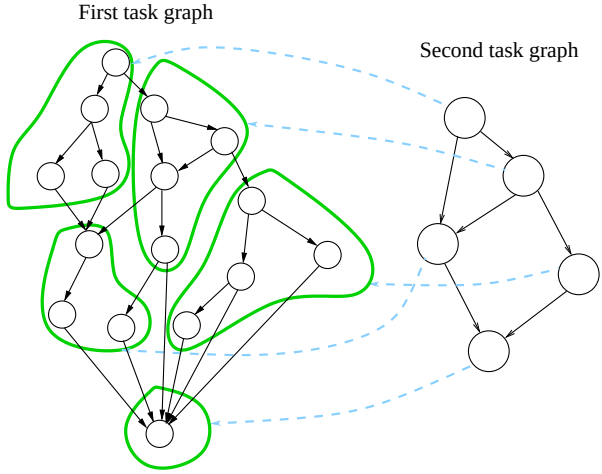


Figure 5.4. The two task graphs used in the tool.

5.5 The Full Task Duplication Method

The parallelization method implemented in the DSBPart tool is called the Full Task Duplication Method, FTD [3]. It is especially suited for task graphs with high granularity numbers, in the range $100 - 1000$. This approach means that clusters are built around each leaf node of the task graph, i.e., a task without any successors, by collecting all predecessors of the leaf node. Hence, the resulting clusters contain all tasks needed by the computation of the leaf task node. Each cluster contains a tree traversal of the task graph originating from the leaf node, following all edges upwards in the tasks, as depicted in Figure 5.6. This simple approach also corresponds to performing a data partitioning of the simulation code to several processors, i.e. a well known parallelization technique. However, it still used the complete data dependency graph (the task graph) to perform this partitioning.

When the clustering has been made, a second phase limits the number of clusters until it matches the number of processors. This merge strategy is performed in three steps.

- First the maximal cluster size is determined. This value is a measure of the speedup that can be achieved by parallelizing the code using the FTD approach, assuming that as many processors that are needed are available.
- Secondly, clusters are merged in a load balancing manner by repeatedly

```
...
{ /* Non-linear system of equations to solve. */
const char*const varnames_[]={"Pipe.Ploss[1]"};
NonLinearSystemOfEquations(Jacobian__, residue__, x__, 1, 1, 1,
  154);
SetInitVector(x__, 1, Pipe_Ploss_1, Remember_(Pipe_Ploss_1, 0));
Residues;
SetVector(residue__, 1, Pipe_mdot_2-
  ThermoFluid_BaseClasses_CommonFunctions_ThermoRoot(
     divmacro(50*Pipe_Ploss_1*Pipe_mdot0*Pipe_mdot0,
       "50*Pipe.Ploss[1]*Pipe.mdot0*Pipe.mdot0",Pipe_dp0,
       "Pipe.dp0"),
  Pipe_mdint*Pipe_mdint));

{ /* No analytic Jacobian available*/
SolveNonLinearSystemOfEquations(Jacobian__, residue__, x__);
Pipe_Ploss_1 = GetVector(x__, 1);
EndNonLinearSystemOfEquations(residue__, x__);
 /* End of Non-Linear Equation Block */ }
...
```

Figure 5.5. Simulation code fragment with implicit dependencies, e.g. between `SetInitVector` and `Residues`.

merging clusters as long as the size does not exceed the maximum cluster size. This phase substantially reduces the number of clusters to a reasonable value, making the next phase less time consuming.

- Finally, clusters are merged until the processor requirement is met by merging the two clusters with the largest number of common nodes. This approach is greedy, since it will try to minimize the maximum cluster size.

Figure 5.7 shows an algorithmic description of the FDT method. Step 0 in the algorithm creates the clusters by collecting the predecessors of each exit node. Step 1 through 3 correspond to the three steps described above. The clusters are described in the algorithm as list data structures where each element is a set of nodes. The input to the algorithm is a task graph as defined earlier in 2.1, and the number of processors. The algorithm returns a list of sets mergedClLst, where each element in a set contains the nodes to be executed on one processor.

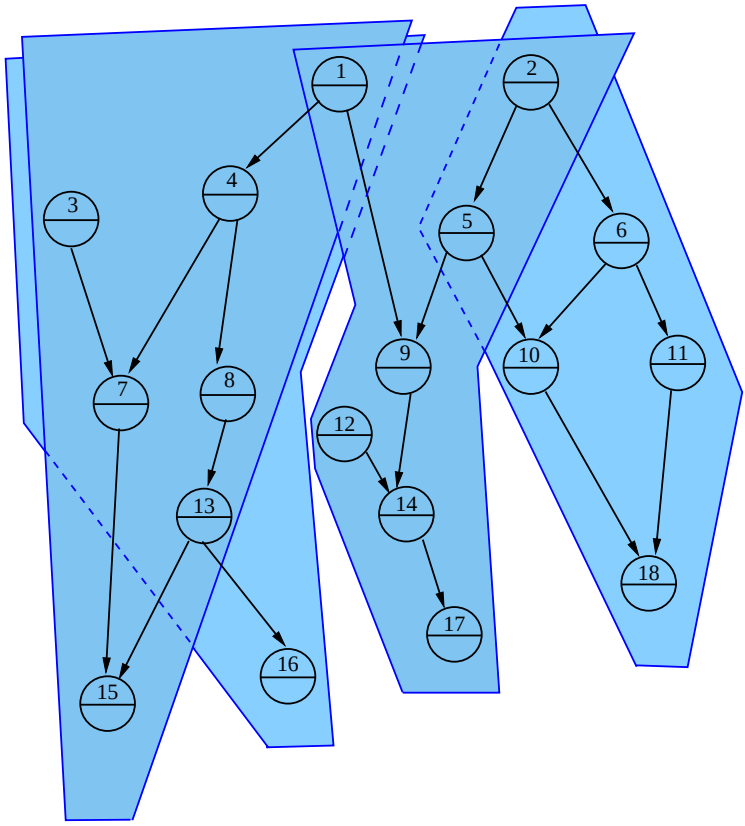


Figure 5.6. Applying full duplication (FTD) to a task graph.

Since the FTD algorithm duplicates all necessary tasks (which is done in the first step when collecting all predecessors for a task), there will be no communication between slave processors *during* the computation of the right hand side. Therefore, the communication that occurs is only between slave processors and the master processor. The master processor will send the state variables to each slave processor, either by broadcast or by individual message sending. After the slaves have finished their execution they will send individual messages back to the master processor, which will update the variables before the numerical solver is executed again.

With this simplified message sending strategy a simple but yet accurate cost model can be used, see Equation 5.1. Here cl_{max} is the maximum execution

algorithm FTD(G: Graph (V, E, c, τ), N:Integer)
clLst, mergedClLst : *list of* $S \subseteq V$
maxSize : *Integer*

clLst:=emptyList()
mergedClLst:=emptyList()
0. $\forall n \in V \mid pred(n) := \emptyset$ *do*
clLst:=addElt({n, predecessors(n)},cl)
1. $maxSize := max_{\forall S \in cl}(\sum_{n \in S} \tau(n))$
2. mergedClLst = addElt($(clLst(1) \cup \ldots \cup clLst(i))$
$, mergedClLst) \mid \sum_{n \in clLst(1) \cup \ldots \cup clLst(i)} \tau(n) <$ maxSize
while $length(mergedClLst) > N$ **do**
3.find $S_1, S_2 \in mergedClLst \mid \sum_{n \in S_1 \cup S_2} \tau(n) = min(\sum_{v \in S_i \cup S_j} \tau(v))$
$\forall i, j \in \{1 \ldots, length(mergedClLst)\}$
$mergedClLst := delElt(S_1, mergedClLst)$
$mergedClLst := delElt(S_2, mergedClLst)$
$mergedClLst := delElt(S_1 \cup S_2, mergedClLst)$
end while

Figure 5.7. An algorithmic description of the FTD Method.

cost of a cluster, L is the latency of the communication network, and B is the bandwidth of the network. The variable n is the maximum size of the messages needed to be sent, thus giving an overestimation of the total cost.

$$C_p = cl_{max} + 2 * (L + n * B) \tag{5.1}$$

5.6 Conclusions

Despite some results, the DSBPart prototype was eventually abandoned for several reasons. The most important reason was that the parser became more and more complicated. In order to capture all variables and data necessary to build the correct task graphs, eventually a complete C-parser would have to be written. Also, new versions of Dymola meant changes in the generated C-code, which meant that the DSBPart tool had to be updated as well.

Another reason for abandoning the prototype was that the OpenModelica environment was emerging as a viable alternative, which would allow the parallelizer to also have more control over the equation optimizations and the solver code. This was not possible using Dymola, where the compiler was only available as a binary executable.

A third reason for building a new prototype was that the old one was not flexible enough and based on low level C-programming. With a new prototype

one could use C++ standard libraries to speed up the development by not having to write everything from scratch. The following prototype, presented in the next chapter, uses both the C++ standard template library and the BOOST library for standard data structures and algorithms.

5.7 Results

Section 9.3 on page 150 presents some of the speedup figures produced by the simulation of models produced by this tool. These are also presented in [3]. The approach produced speedups for some examples using the FTD method, whereas other examples did not produce any speedups due to limited amount of parallelism in the simulation code. One conclusion made was that better clustering algorithms that use task replication to increase the amount of parallelism and limit the cost of communication was required.

5.8 Summary

The DSBPart parallelization prototype translates the C-code generated from the Dymola tool into parallel C-code using MPI. It is the first prototype and method made for parallelization of simulation code from Modelica models. It has a simple clustering algorithm called Full Task Duplication (FTD). A few models could be parallelized using the tool.

Finally, the DSBPart tool was abandoned and a new tool was developed as an integral part of the OpenModelica environment. This tool is presented in the next chapter.

Chapter 6

ModPar - an Automatic Parallelization Tool

ModPar is the name of the main automatic parallelization tool for Modelica models that has been developed in this thesis work. It is a part of the back-end of the compiler in the OpenModelica framework. This chapter describes this framework in general and the ModPar module in particular.

6.1 Research Background

When the DSBPart prototype had evolved to a stage where it could translate some Modelica models to parallel code it became evident that a more flexible parallelization tool was needed. The parallelization tool should for instance also have control over the numerical solution routines and this was not possible in the DSBPart tool. Moreover, the parallelization tool also needed control over the equation optimization routines to fully exploit different options. The OpenModelica compiler provided all this features and provided an open source research compiler framework. Therefore it was quite natural to re-implement the parallelization prototype in the OpenModelica compiler.

6.2 Implementation Background

The DSBPart prototype described in the previous chapter evolved enough to be able to parallelize some examples, running simulations on Linux cluster computers [3].

After a few years of development the Open Modelica compiler was complete enough to make it possible to perform our parallelization research in the OpenModelica framework, with the advantage of having full access to the source code including numerical solvers, as well as to all the details of the equation optimizations performed on the flat set of equations. This has resulted in the development of the ModPar parallelization tool.

Initially, an early version of the the ModPar tool was integrated with the ModSimPack tool, which parse a flat Modelica file and built up a graph representation of the equations and variables, etc. The ModSimPack tool was a prototype developed in the equation debugging PhD thesis project [10], implemented using C++ and a computer algebra package called SymbolicC++. However, in the spring of 2004 it became evident that both the ModSimPack module itself as well as the SymbolicC++ library contained both severe bugs (memory leaks) and inefficient algorithms. These defects made the translation of larger simulation models intractable.

It was also quite obvious that the symbolic transformations and the equation sorting, etc. required a much more tighter integration with the OpenModelica front-end. Therefore we decided to instead implement the missing parts including the ModPar tool directly as a module in the OpenModelica compiler. This also had the advantage of not having to write yet another parser for flat Modelica.

6.3 The OpenModelica Framework

The OpenModelica framework is a set of tools for modeling, compilation and simulation of Modelica models [55, 6]. It consists of several separate tools that communicate through files and through a CORBA/socket interface.

This framework is entirely developed under Open Source, using the new BSD license.

6.3.1 Overview

Figure 6.1 gives an overview of the OpenModelica framework. The OpenModelica model compiler (omc) translates Modelica models into an executable file, which when executed performs the simulation of the model. The compiler is internally divided into the omc front-end and the omc back-end. The front-end translates Modelica .mo files into an intermediary format called flat Modelica.

Flat Modelica is basically a simplified Modelica format containing declarations of variables of built-in type, equations, algorithm sections and functions. It corresponds to the Hybrid DAE [27] that constitutes the model equations for the complete model. In order to produce this format the omc compiler flattens

the inheritance structure, elaborates model components, and generates simple equations from connect equations.

In the next step of compilation, OpenModelica sorts the equations, tries to reduce the size of the equation system by equation optimizations such as elimination of simple variables. It also performs index reduction to eliminate certain problems during numerical solution. Finally it generates C-code from the sorted and optimized equations. The C-code is linked together with a numerical solver.

The last part of the toolset mentioned here is the interactive shell tool. It is a simple shell-like interpreter that is directly linked to the omc, which can also be executed in an interpretive mode. This allows us to let users interact with Modelica models, simulations, Modelica functions, etc. in an interactive manner, basically providing the same functionality as other interpreted computation engines, such as e.g. Matlab.

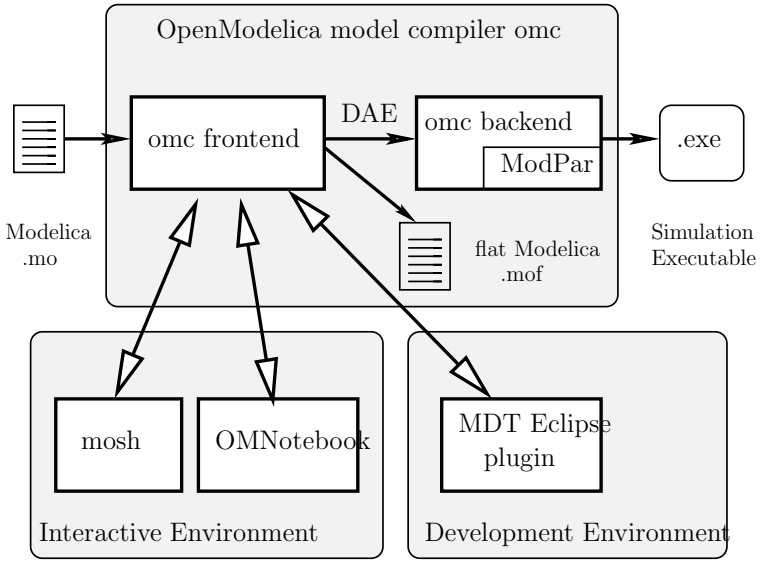


Figure 6.1. The OpenModelica framework.

6.3.2 Modelica Semantics

The omc tool translates Modelica code (files with extension .mo) into flat Modelica (a file with extension .mof). It can itself be seen as a compiler

even though it is only a front-end in the OpenModelica model compiler, see Figure 6.1. The omc compiler is written in a language called RML [63], which uses Natural Semantics to describe the semantic rules of the compiler.

Natural Semantics is a popular formalism for formally describing compiler semantics, such that for instance proofs of properties of languages can be made. RML, which is inspired from standard ML [75] and Natural Semantics [30], has successfully combined the strength of Natural Semantics with highly optimized and efficient compilation techniques to enable compiler writers to use Natural Semantics for real applications, such as generating efficient compilers from formal specifications of real programming languages.

Natural Semantics in RML consists of *relations*, which can be thought of approximately as functions. Each relation contains a sequence of *rules*. Each rule contains a sequence of *premises* and a *conclusion* which can be viewed as producing the return value of the relation.

The rules are matched in a sequential manner, starting with the first rule in the relation. If the pattern of the rule matches the input to the relation the premises of the rule are computed. If all premises of the rule successfully evaluates, the conclusion is returned. However, if any of the premises do not match, the rule fails and the next rule in the sequence is tried.

As an example we will now show a small interpreter in RML for a small calculator language. First we define the data types, i.e., the abstract syntax, that represents expressions in the language.

```
datatype Exp =
         ADD of Exp * Exp
         | MULT of Exp * Exp
         | CONST of real
```

RML borrows the data type constructor syntax and semantics from the Standard ML language. Here we defined a data type `Exp` that can either be a constant of the built in RML-type `real`, an addition of two `Exp` values (the `ADD` node), or a multiplication of two `Exp` values (the `MULT` node).

We now define a small evaluator relation that evaluates values of this data type, i.e., calculates the real values of expressions.

```
relation eval: Exp => real =

  rule eval(e1) => c1 &
       eval(e2) => c2 &
       real_add (c1,c2) => sum
       ----------------------
       eval(ADD(e1,e2)) => sum
```

```
  rule eval(e1) => c1 &
       eval(e2) => c2 &
       real_mult(c1,c2) => prod
       ------------------------
       eval(MULT(e1,e2)) => prod

  axiom eval(CONST(c)) => c
end
```

The relation `eval` consists of three rules. The first rule matches expressions having the `ADD` node. The premises are that given an `ADD` node with the two subexpressions `e1` and `e2` and the premises that they recursively evaluates to the values `c1` and `c2` respectively and having the final premise that the addition of `c1` and `c2` gives the variable `sum`, the conclusion of the rule is this `sum`. Analogously, the second rule deals with the `MULT` type.

The last rule is an *axiom* which means that it has no premises and will therefore always apply as soon as the matching criteria are fulfilled.

The whole OpenModelica compiler is written in RML, resulting in about 72 000 lines of RML code. The author has spent quite some time in implementing the omc tool writing RML code, and has thus become an experienced RML programmer and specification writer. We present some conclusions regarding the properties of the RML language and its compiler in Chapter 11. Many Modelica language constructs that were needed to express Modelica models relevant for parallelization have been implemented by the author.

The main relation in the omc front-end is presented below:

```
relation main =
  rule  Parser.parse f => p &
        SCode.elaborate(p) => p2 &
        Inst.instantiate(p2) => d &
        DAE.dump d
        ----------
        main([f])
end
```

Its first task is to parse the Modelica program text given as input. This is performed by the `parse` relation in the `Parser` module. The actual parser is an external function to RML, which is implemented using the ANTLR parser generation tool and has a parse tree walker that builds the RML data objects, i.e., the Abstract Syntax Tree (AST). The AST is returned from the `parse` relation call and put in the variable `p`.

The next step in the Modelica compilation is to transform the AST to a canonical form, removing redundant information, simplifying the tree structure, etc. This is done in the OpenModelica `SCode` module by the `elaborate` relation. The result from this relation is then sent to the symbolic instantiation process in the `Inst` module.

Symbolic instantiation in Modelica is the process of flattening the inheritance structure symbolically, instantiating components, and generating equations from connect equations. All this results in a flat Modelica class, defined in the `DAE` module.

6.3.3 Modelica Equations

The most important back-end part of the OpenModelica compiler is the DAELow module. It translates the equations and variables (defined in the DAE module) into C-code which is linked together with a numerical solver giving an executable program for running the simulation of the model.

To reduce the execution time of the simulation, a set of optimization techniques are used, see Section 3.2. Currently, only removal of simple equations and BLT sorting combined with index reduction are implemented in the OpenModelica back-end.

Finally code generation is performed. It is here that the ModPar tool is plugged in, as an alternative code generator that also analyzes the systems of equations in order to parallelize them.

6.3.4 Interactive Environment

The last tool to mention in the OpenModelica framework is the interactive shell tool. It provides an interactive shell where users can perform calculations using Modelica expressions and call Modelica functions, built-in or user defined. The environment also makes it possible to create model definitions, to retrieve model definitions, or change them through an API interface suitable for metaprogramming.

An example session in the interactive environment is given below:

```
>>> a:=1:10
{1,2,3,4,5,6,7,8,9,10}
>>> b:= a * 2
{2,4,6,8,10,12,14,16,18,20}
>>> a*b
770
>>> model test Real x[3]=fill(1,3); end test;
Ok
```

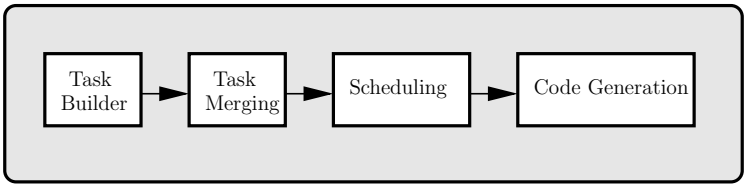


Figure 6.2. The ModPar internal modules.

```
>>> instantiateModel(test)
" class test
Real x[1];
Real x[2];
Real x[3];
equation
  x[1] = 1.0;
  x[2] = 1.0;
  x[3] = 1.0;
end test;
"
>>>
```

6.4 The ModPar Parallelization Tool

As previously mentioned, the ModPar automatic parallelization tool is an optional part of the back-end of the OpenModelica compiler. The internal structure of ModPar consist of a Task Graph building module, the task merging module, a scheduling module, and a code generation module, see Figure 6.2

6.4.1 Building Task graphs

Two task graphs are built from the sorted equations. The first is the fine grained task graph corresponding to the expressions of those equation that could be converted to assignment statements, or larger tasks for solving of simultaneous sub-systems of equations.

The second task graph is the task graph used for merging of tasks. It is initially a copy of the first fine-grained task graph, but will after task merging become a more coarse grained graph. Both of these task graphs must be kept throughout the parallelization process. The first fine-grained task graph is

used in the final code generation, whereas the second merged task graph is used in the scheduling phase to determine on which processors and in which order the tasks will execute.

6.4.2 ModPar Task Graph Implementation Using Boost

The task graph implementation in ModPar is based on the C++ Boost Graph Template library (BGL) [8]. The BGL has an extensive use of C++ templates to be able to change the graph data structures, as well as graph attributes, etc. The task graph used in the implementation uses adjacency list representations of a graph, enabling fast insertion and deletion of edges and nodes of the graph. This is essential for the task merging algorithm which relies on fast implementation of such operations.

The two task graphs are defined as a C++ template, using boost:

```
typedef boost::adjacency_list<boost::listS, boost::listS,
                              boost::bidirectionalS,
                        VertexProperty, EdgeProperty> TaskGraph;
```

This will define a type that can be used for a graph implementation using adjacency lists. There are two lists for adjacent nodes, one list for the immediate predecessors and one list for the immediate successors of a node. It will also use the types `VertexProperty` and `EdgeProperty` for the different properties of vertices and edges of a task graph. These types are defined as:

```
using namespace boost;
typedef property<vertex_name_t,string,
         property<vertex_execcost_t,float,
          property<vertex_unique_id_t, int,
           property<vertex_index_t,int,
            property<vertex_color_t, default_color_type,
             property<vertex_resultname_t, string,
              property<vertex_tasktype_t, TaskType,
               property<vertex_malleable_t, bool>
              >
             >
            >
           >
          >
        > VertexProperty;
typedef property<edge_weight_t, int,
```

```
        property<edge_result_set_t, ResultSet >
      > EdgeProperty;
```

The `VertexProperty` type associates the properties

- A name - of type `string`.
- An execution cost - of type `float`.
- A unique id - of type `int`.
- A vertex index - of type `int`. This attribute is needed for certain graph algorithms in boost.
- A vertex color - of type `boost::default_color_type`. This attribute is also needed for certain graph algorithms in boost.
- A result name - a `string` containing the result of the task.
- A task type - An enumeration of task types, used for code generation.
- A malleable flag - A flag indicating if a task is malleable or not.

The `EdgePropery` only contain two attributes. An edge weight of type `int` for the communication cost and a `ResultSet` attribute giving the set of all results from a task. The reason for this resultset is to be able to perform merging of several tasks without replicating data in the communication packets.

Fine Grained Task Graph

The task graph is built from the sorted equations. For each arithmetic expression, function call, subsystem of equations, etc. a task is created. A data dependence edge is created between two tasks from a definition (i.e., assignment) of a variable in one task to the corresponding use of the variable in another task.

As an example, we use the `SmallODE` example introduced in Section 5.4:

```
model SmallODE
  parameter Real a=3.5;
  parameter Real b=2.3;
  Real x;
equation
  der(x)=-a*x+b/(time+1);
end SmallODE;
```

From this code a task graph is built as shown in Figure 5.3. First, the definitions of the variables in the code, (`der_x, a` and `b`) are added, creating a task for each variable definition. These tasks are definition nodes, therefore their execution cost is zero. A symbol table keeps track of the tasks that define the value of a given symbol. For instance, the variable name `der_x` points to the define task for the variable `der_x`, see Figure 5.3.

When the equations are traversed, tasks for the division, the two additions and the multiplication are created in a similar manner as done by the DSPPart tool described in Chapter 5. For instance, when the multiplication task is created, the two operands are looked up (i.e., the definition tasks for `a` and `x` are accessed) and edges between the operand tasks and the multiplication task are created. The symbol table entry to the variable `der_x` is updated, so that subsequent reads of the same variable will connect data dependence edges to the new task instead of the definition task of the `der_x` variable. For scalar values the communication costs associated with the edges are set to the cost of sending one scalar value between processors. In the example we use a cost of 100 units, e.g could be 100 microseconds, for a communication of one variable.

Merged Task Graph

From the task graph corresponding to the sorted equations we build a second task graph. This is required to be able to later merge tasks but still having the information of the original task. However, many scheduling algorithms assume a single entry, single exit task graph. This means that the task graph should only have one entry node (a node without any predecessors) and one exit node (a node without any successors). The second task graph is therefore built having a single entry node and a single exit node. However, the entry and exit nodes are not considered during task merging, since they should remain in the resulting graph.

When the task graph has been built, each task has to be assigned an execution cost and each edge a communication cost. One approach of estimating the different costs is to use profiling. Since the simulation code is executed repeatedly each time step of the numerical solver, with almost the same execution time, a simple profiler can be used to measure the execution costs.

However, a simpler approach where each different task type is estimated by hand could be sufficient. It is more important to give a good estimate of the *relation* between communication cost and execution cost, since this is the main factor that affects the possible speedup when executing on a parallel computer. Once this relation has been measured with enough precision, the other costs are worth considering.

6.4.3 Communication Cost

The communication costs can be determined by measuring the time it takes to send differently sized datasets between two processors on the targeted multiprocessor architecture. Typically, for small sizes of data, the affecting parameter is mostly the latency of the parallel computer, see Section 2.2.2. Therefore, for the fine grained task graphs produced in our parallelization tool, the latency is the most important parameter.

Such measurements are commonly used to benchmark different communication APIs on different machines. Figure 6.3 gives the latency and bandwidth measured (by each specific vendor) for different multiprocessor architectures [1] [70, 51]. The values in Figure 6.3 are measured at the MPI software level (except for Firewire) , thus including the overhead of calling the API functions when sending a message.

	Bandwidth (Mbyte/sec)	Latency μs
Scali MPI (SCI network)	199.2	4.5
GM (Myrinet network)	245	7
SHMEM (SGI Origin 3800)	1600	0.2-0.5
Firewire (at hardware level)	50	125

Figure 6.3. Bandwidth and latency figures for a few multiprocessor architectures and interconnects.

6.4.4 Execution Cost

The execution cost of the tasks in the task graph can either be estimated given the architecture specification of the targeted platform, or determined by measuring the time by profiling the simulation code. The method used depends on what accuracy is needed. When using estimates of the execution cost instead of actual measurements, effects from the cache is often neglected, giving large errors in the approximation.

On Pentium-based processors there is a special instruction that counts the number of cycles elapsed since the last reboot of the processor. A single assembler instruction put inside a function is therefore sufficient for measuring high resolution time. The code in Figure 6.4 illustrates how this method can be used. One problem is that the compiler might optimize the code, moving parts of the computation that should be measured outside the two calls to the measuring function (`rdtsc` in Figure 6.4). This can be solved by turning off

[1] The SGI computer is a shared memory machine, thus the figures denote writing data to shared memory.

the optimizations responsible for moving the code. However, that introduces errors in the measurements which could be large.

```
__inline__ unsigned long long int rdtsc()
{
  unsigned long long int x;
  __asm__ volatile (".byte 0x0f, 0x31" : "=A" (x));
  return x;
}

int main(int argc, char **argv)
{
 long int start,end;

  start = rdtsc();
  myfunc(argc,argv); // function to measure
  end = rdtsc();
  return end-start;
}
```

Figure 6.4. Pentium assembler code for high resolution timing.

There are certain tasks that are harder to measure. For instance, the code for solving of a non-linear system of equations is based on a fixed point iteration. Thus, the execution cost for the corresponding task graph can not be estimated well enough, since the number of iterations depends on the input values of the involved variables. For these tasks, a less precise estimation is given.

For other tasks, corresponding to code solving a linear system of equations, the cost estimate can be obtained from the number of involved variables. Hence, the cost of for instance solving a linear system of equations involving ten variables can be estimated by a function call $f(10)$, where f is defined in (6.1). If the standard LaPack [1] function for solving a linear system of equations is used (`xGESV`), the function described in Equation 6.1 can be used, where C_1 and C_2 are constants that can be determined by for instance profiling as above.

$$f(n) = C_1 \cdot n^3 + C_2 \tag{6.1}$$

6.4.5 Task Merging

Once the task graph has been built from the sorted and solved equations the task merging takes place. The next chapter explains the task merging process in detail, for now it is sufficient to now that task merging is primarily performed to increase the task graph granularity, such that the preceding scheduling phase can achieve a better schedule.

6.4.6 Task Scheduling

Once the task merging has been performed, resulting in a coarse grained task graph, task scheduling is initiated. The task scheduling algorithm schedules the task graph given a specific parallel architecture platform with P processors. The ModPar tool uses the TDS scheduling algorithm, which can optimally schedule coarse grained task graphs given some constraints. Unfortunately it can not build a schedule for a fixed number of processors, therefore it is followed by a simple load balancing scheduling algorithm to reduce the schedule to a fixed number of processors.

The TDS algorithm also uses a restricted form of task replication to decrease the number of sent communication packets. However, the TDS algorithm does not use the cost model with two parameters for the communication cost (bandwidth and latency), see Chapter 7. Instead it uses the delay model, presented in Chapter 2.

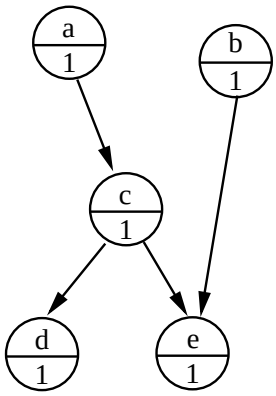
The scheduling algorithm is implemented as one C++ class, making it easy to extend the ModPar tool with new scheduling algorithms.

6.4.7 Code Generation

After scheduling comes the task of generating code for running on a parallel machine. Since we are using a data dependence graph (task graph) as the underlying data structure for our parallelization, a message passing programming model is suitable. Programming models (typically libraries) with message passing can be implemented on both distributed shared memory machines as well as shared memory machines. For instance, MPI (Message Passing Interface) implementations exists for both types of architectures.

Code generation is performed for one processor at a time by traversing the tasks assigned to the processor. Once a task needs data, due to a data dependence in the task graph, from a task not assigned to the same processor, an explicit MPI receive call is inserted to receive this data from a processor executing this task. The same goes for a task having a successor task not executing on the same processor. Then a MPI send call is inserted to send the data to the processor(s) responsible of the execution of that task.

For example, consider the merged task graph in Figure 6.5. Since the task graph is merged, there can be several scalar values communicated along the edges of the task graph, forming larger data packets. For example, when generating code for task c of the merged task graph, all immediate predecessors (task a) is executed on the same processor, so there is no need to generate receive commands. Then follows the code generation for the task c itself, which internally can consist of several tasks. These internal tasks are traversed in a top level order to ensure that all data dependencies are fulfilled and code is generated for each subtask. Then follows the code generation of send commands to successor tasks that are executing on other processors, in this case task e.



(a) Merged task graph

P0	*a c d*
P1	*b e*

(b) Schedule of task graph for two processors

Figure 6.5. Code generation on a merged task graph.

Thus, the code generated for processor 0 would have the following structure:

```
/* Code for task a */
...
/* Code for task c */
...
sendbuf[0]=d1;
senbbuf[1]=d2;
...
sendbuf[n]=dn;
MPI_Send(buf,n,1); // Send data to processor 1
```

```
/* Code for task d */
...
```

where `d1, d2,...,dn` are the scalars associated with the communication edge (c,e). Similarly, the code generated for processor 1 would look like:

```
/* Code for task b */
...
MPI_Recv(recvbuf,n,0); // Recv data from processor 0
/* Code for task e */
...
```

The code generation algorithm generates one function for each processor, called `proc0, proc1, ... procn` for n processors. The main function of the parallel simulation code will then determine the processor rank by using the `MPI_Comm_Rank` function and call the proper function. Processor zero will also execute the numerical solver, i.e., the main loop of the ODE/DAE solver, so the code looks a little bit different in this case.

Once the code generation is done, the automatic parallelization tool has completed its task of parallelizing the simulation code. The code must then be compiled and transferred to the parallel computer for execution.

6.5 Summary

The ModPar parallelization tool is a part of the OpenModelica framework for automatic parallelization of Modelica simulations. The parallelization scheme is intended to parallelize the computations of solving the system of equations (right hand side calculations of the underlying ODE/DAE, but currently not parallelize the central solver algorithm) and execute these in parallel. This is performed by building a data dependency graph of these calculations, a so called *task graph* and thereafter merge tasks in this graph using a proposed task merging algorithm with the goal of increasing the granularity of the task graph. The new task graph, resulting from the task merging process, is then scheduled using conventional multiprocessor scheduling algorithms to achieve faster execution of the simulation code on a parallel computer. After scheduling, code generation is performed using MPI (Message Passing Interface), making it possible to execute the simulation code on a variety of parallel computer architectures.

Chapter 7

Task Merging

This chapter presents techniques of merging tasks in a task graph and our contributions made in that area. Merging of tasks is performed to increase the granularity of a task graph, obtaining fewer tasks with increased computation cost making it more suitable for scheduling by better balancing the ratio between communication cost and computation cost of the task graph. This work has partly been published in [5].

7.1 Increasing granularity

The goal of a task merging algorithm is to increase the granularity of a task graph. This is performed by merging tasks to increase the computational work of the tasks compared to the communication cost between different tasks of the task graph. When tasks are merged their computation costs are added, thereby increasing the execution cost while the communication costs remain unchanged. However, by merging tasks we also reduce the degree of parallelism in the task graph. At the extreme, if we merge all tasks into a single tasks, there is no parallelism left in the task graph. Thus the task merging problem must be balanced against the amount of parallelism of the task graph. Thus the task merging problem can be viewed as a max-min problem. We want to maximize the amount of parallelism in the task graph while minimizing the cost of communication. Or actually we want to minimize the inverse of the granularity, resulting in large execution costs and low communication costs.

7.2 Cost Model for Task Merging

As a cost model for the execution and communication of the task graph we use a variant of the cost model from the LogP parallel programming model, see Section 2.2.2 on page 29. We use the latency, l, and the bandwidth, B, from the LogP model to measure the cost of communication. Equation 7.1 gives the cost for sending n bytes of data from processor i to processor j.

$$C_{i,j} = l + \frac{n_{i,j}}{B} \tag{7.1}$$

This model is more advanced compared to the delay model since it considers the latency metric, which is independent of message size sent. The latency corresponds to the initialization cost of sending a message between two processors. The latency varies substantially between parallel processor architectures but typically lies in the proximity of about 10 μs. The latency can be estimated by the time the message is occupying the communication link, i.e., the amount of time the message takes to travel from one processor to another. This measure is directly proportional to the length of the actual physical link on which the message is transmitted. This length can of course vary between different pairs of processors on different types of architectures, but in this context a uniform latency cost is provided which is the same for communication between any pair of processors.

The second term of the communication cost is the bandwidth. It is proportional to the actual message size and measure how long the message takes to communicate over the network. The bandwidth is proportional to the communication speed of the physical link between the processors. It typically measures how many bytes can be transmitted over the network per second. This can of course also vary between different pairs of processors but here is given a uniform value for a given architecture.

When having these two terms as the cost of communication between processors it can be beneficial to merge data packets sent between processors. For instance, if task a sends different data to both b and c in Figure 7.1 below, the messages could be merged to a single message provided that task b and c are merged. This would reduce the overall cost of outgoing communication for task a by l, since only one message is sent. If we would have had a simpler cost model, such as the delay model, this merge would perhaps not have been worthwhile.

7.3 Graph Rewrite Systems

A Graph Rewrite System (GRS) is an universal way of transforming an arbitrary graph by adding or removing edges and vertices by a set of transformation

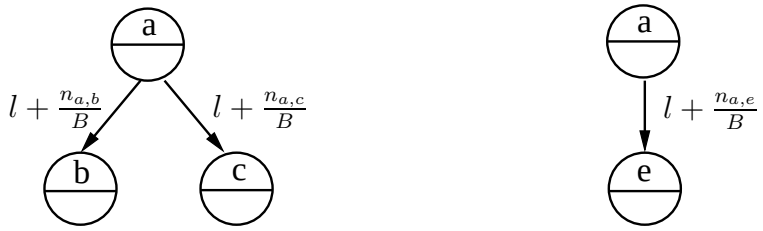


Figure 7.1. Task b and c could be merged into task e to the right where $n_{a,e} = n_{a,b} + n_{a,c}$, resulting in one message sent.

rules. Each transformation consists of three or four parts.

- A *subgraph* pattern which will be mapped on the targeted graph to find matching subgraphs.
- A *resulting* subgraph, also called *redex* which is the resulting subgraph of the matched subgraph of the transformation by adding or removing edges or vertices from the subgraph.
- A *condition* that must be fulfilled in order for the transformation to be applicable.
- An *invariant* statement, indicating what remains unaffected by the transformation, e.g. certain attributes of edges and vertices.

A GRS typically consists of several such transformation rules which can both add and remove edges and vertices. Such a system can be applied to graphs until no more transformations can be performed, causing the GRS to *terminate*. The termination of a GRS is a desirable property for most practical applications of graph rewrite systems. For instance, by having one rule that adds a vertice and a second rule that removes the same vertice we get a GRS that will never terminate.

A second property of a GRS is its *confluence*. A GRS is *confluent* if the ordering of the transformations is not dependent on the resulting graph. This means that regardless of in which order the transformations are applied the resulting transformed graph will be the same. This property is desirable to ensure that the transformation rules can indeed be applied in any order as long as each individual condition of the transformations is fulfilled.

If a GRS is not confluent it can in some cases be made confluent by e.g. giving priorities to the transformation rules or by dividing the GRS into several consecutive graph rewrite systems which are applied in a sequence.

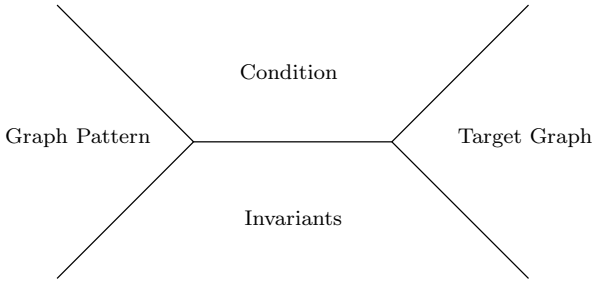


Figure 7.2. The X-notation for a transformation rule in a GRS.

7.3.1 The X-notation

There are several notations for graph rewrite systems available in the literature []. One of these is the *X-notation*. It describes the four parts of a graph transformation rule in a graphic form using the `X` character, as shown in Figure 7.2. To the left is the pattern subgraph which will be matched against subgraphs of the targeted graph of the GRS. To the right is the resulting subgraph, the redex. Above is the condition expression for the transformation and below are the invariants of the rule.

In this thesis we will use the X-notation for describing our graph rewrite rules used for task merging.

7.4 Task Merging using GRS

One contribution of this thesis is the idea of merging tasks in task graphs using a graph rewrite system formulation. In the following we will present a set of task merging transformations that together constitutes a GRS. This task merging system will then be used on fine grained task graphs from simulation code to transform them into more coarse grained task graphs that hopefully still reveal much parallelism. We call this method ATMM (Aronssons Task Merging Method).

The main idea behind our task merging GRS is to merge tasks together as described earlier in this chapter with the goal of increasing the granularity of the task graph. A condition for performing the merge is that the (top) level, see Equation 2.3 of the involved tasks of the transformation are not increased. With this simple strategy we can construct a set of rules that will merge tasks together resulting in larger execution costs of tasks in the task graph and larger data sizes of the communication between tasks. The following section presents

a first attempt at constructing a task merging algorithm using graph rewrite systems.

7.4.1 A First Attempt

As a basic task merging algorithm using GRS we construct three simple rules, to deal with different patterns of a task graph. Each rule is given an explanatory name for reference later on. The three basic rules are presented in the three following sections.

Merging of Single Child Nodes

A first and quite simple rule to have in a task merging GRS is to allow merging of tasks when this does not decrease the amount of parallelism of the task graph. This will occur if the task graph contains two tasks where the first only has the second as a single immediate successor and the second only has the first as single immediate predecessor. In this case no parallelism is lost by merging these two tasks. Also the transformation will increase the granularity for the resulting task and perhaps also for the complete task graph. The transformation will also reduce the number of tasks in the task graph, which is beneficial from a computational complexity point of view.

The transformation rule is depicted in Figure 7.3 and is named *singlechildmerge*. The dotted ingoing edges on each node, e.g. p, mean that the pattern allows the p task to have an arbitrary number of immediate predecessors (zero or more). This is also true for the immediate successors of task c in Figure 7.3. The resulting task graph will have a task p' as the result of merging the tasks p and c. Therefore the execution cost $\tau(p')$ of p' can be calculated as $\tau(p') = \tau(p) + \tau(c)$. The top level attribute for all predecessor tasks of p remain unchanged by this merge, hence they are stated as invariants of the rule, see Figure 7.3.

Merging of Join Nodes

A join node, e.g. at the left of Figure 7.4, can be merged with all its immediate predecessors (i.e., parents) given that the level of the join node does not increase after the merge. The new merged task will then contain the computational work of the join node itself and all predecessor tasks of the join node. This is the *mergeallparents* rule which is shown in Figure 7.4.

The condition can be formulated as

$$tlevel(c) > \max tlevel(p_i) + \sum_{p_i \in pred(c)} (\tau(p_i)) + \tau(c) \tag{7.2}$$

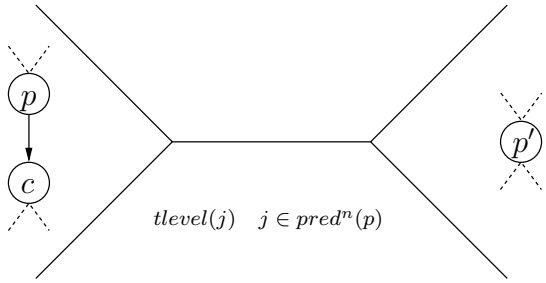


Figure 7.3. The *singlechildmerge* rule. The invariants states that all predecessors of p have the same top level after the merge.

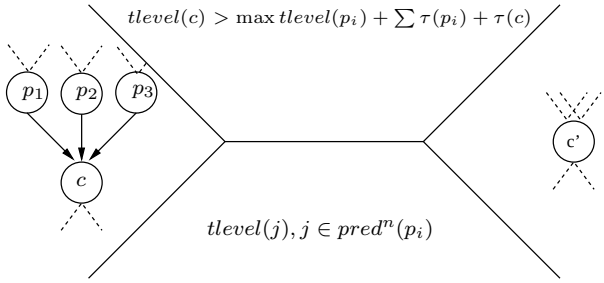


Figure 7.4. The *mergeallparents* rule. The condition becomes true if $tlevel(c') < tlevel(c)$, invariants of the rule are the top level value of all predecessors of p_i.

which means that the merged node c' must get a lower top level than the join node c after the merge. The resulting merged task c' will have an execution cost of $\tau(c') = \sum \tau(p_i) + \tau(c)$, hence giving the condition in Equation 7.2. This condition will typically be fulfilled if the latency, $l >> \tau(n)$ for all the involved tasks n in the pattern.

Moreover, this transformation rule will decrease the number of nodes of the graph and increase the granularity for the c' task and potentially for the complete graph as well. Similarly, as for the *singlechildmerge*, all immediate predecessor tasks of each predecessor p_i of c will have their tlevel values unchanged by the merge, i.e., they are invariants of the rule, see Figure 7.4.

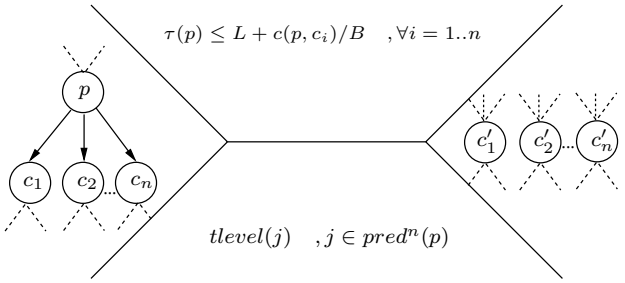


Figure 7.5. The *duplicateparentmerge* rule.

Merging of Split Nodes

In order to merge tasks for a split node, e.g. p, depicted at the left of Figure 7.5, we need to perform task replication. The split node p can be replicated into as many copies as there are immediate successors of the split node and then each individual replica can be merged with one of the immediate successors. This rule is called *duplicateparentmerge* and is depicted in Figure 7.5.

The condition of merging a replica of the split node p into each immediate successor c_i is that the execution cost for the split node must be less than the communication cost between the split node and each immediate successor of the split node. The resulting tasks are the merged immediate successors c'_i to the split node p, with the execution costs $\tau(c'_i) = \tau(p_i) + \tau(p)$.

An Example

To illustrate how the task merging algorithm works we consider the task graph example given in Figure 2.1 in Chapter 2, also shown in Figure 7.6(a). Figure 7.6 shows a possible task merging transformation of this graph. Figure 7.6(a) shows the input task graph before transformations are applied. The first rule that is applied in this case is the *singlechildmerge* rule, merging tasks 4 with 7 and task 3 with 6, resulting in the task graph in Figure 7.6(b). Since this rule has no condition these transformations will be performed regardless of the sizes of the bandwidth and latency parameters.

After this step the *duplicateparentmerge* rule is applied, first duplicating task 1 into task 2 and task (4,6) respectively. Then the rule is applied again, replicating task (1,2) into task (4,7) and task 5, resulting in the task graph shown in Figure 7.6(c). Finally, the *mergeallparents* rule is applied, merging task (1,2,5) and task (1,3,6) with task 8 into a single task, resulting in the task graph depicted in Figure 7.6(d).

This particular merge also illustrates that the task merging algorithm needs to keep track of which subtasks are contained in a task, to be able to merge them correctly. Otherwise, this particular merge would have counted the replicas of task 1 twice resulting in an erroneous execution cost of the merged task. Similarly, the algorithm also must keep track of the messages sent between tasks. Due to a merge, the same message might otherwise be counted several times, also giving an erroneous cost.

For this example with the given communication cost, all conditions for the transformations are fulfilled. This results in an almost complete reduction of the task graph. However, with different parameters of the communication cost the result would have been different, since some of the conditions of the transformations would have been false.

Limitations

Even though the example above could be reduced to two nodes when the communication cost was as high as given, there are other cases where the three task merging transformations given above are not enough. For instance, consider the graph in Figure 7.7. The rule *duplicateparentmerge* will fail for task 1 since task 4 has an other immediate predecessor (task 2). Similarly, *mergeallparents* will fail for task 4 since task 1 and task 2 has another immediate successor, task 3 and task 5 respectively. Obviously, these cases can occur quite often and therefore we need to enhance our rules to make them more applicable.

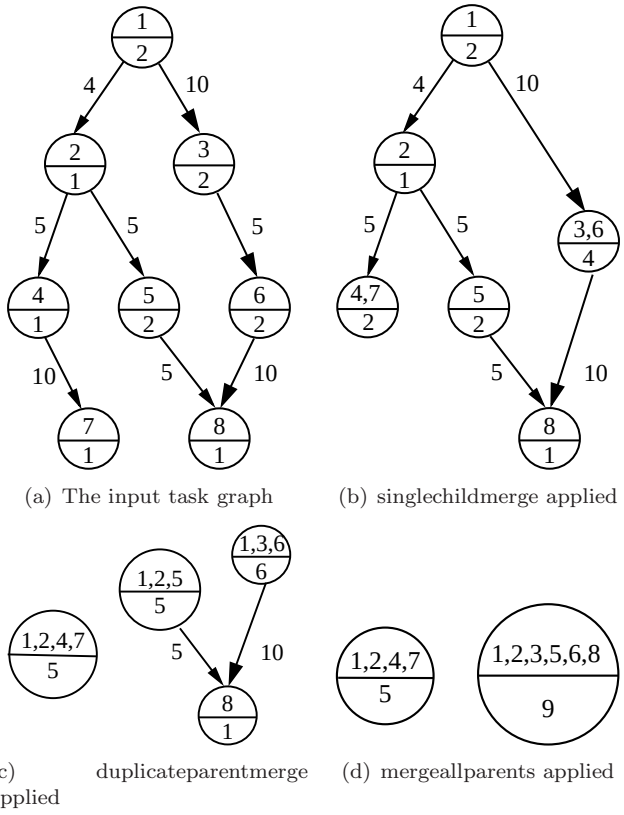
7.4.2 Improving the Rules

The *duplicateparentmerge* rule and the *mergeallparents* rule can be improved in several ways. The first improvement for both of these rules is to allow other immediate predecessors as discussed above. We then need to enhance the conditions of the rules to make sure that these immediate predecessors will not increase their top level value by the merge. These extra conditions will increase the execution cost of applying the transformation since the extra conditions must be checked. However, to be able to cope with real application examples, these improvements are necessary.

Below follows improvements of the above mentioned rules.

Improved Rule for Join Nodes (mergeallparents2)

The *mergeallparents* rule can be improved by allowing the immediate predecessors to the join node c, see Figure 7.8, to have *other* immediate successors besides c. The communication to these other immediate successors will as a result of the merge be delayed to after the computation of the merged task.



(a) The input task graph (b) singlechildmerge applied

(c) duplicateparentmerge applied (d) mergeallparents applied

Figure 7.6. A series of task merging transformations applied to a small task graph.

Thus, the condition is expanded to check that these other immediate successors top levels are not increased by the merge. This is the second line of the condition in Figure 7.8. Typically this can be true if these other tasks have a higher top level due to some other unrelated tasks in the task graph.

Yet another improvement is to relax the condition that other immediate successors to p_i tasks should not increase its top level. If the immediate successor nodes to some p_i will increase their tlevel they can not be merged into the merged task. We can then improve the rule by dividing all immediate predecessors p_i into two disjunct sets, those that fulfill the condition ($c \in C$) and can be merged and those that do not fulfill the condition ($c \notin C$). The merge can then only be performed with all immediate predecessors if the im-

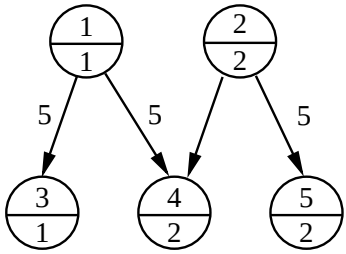


Figure 7.7. A small example where *duplicateparentmerge* and *mergeallparents* will fail. Duplication of task 1 is prevented by task 2 and merging of tasks 1,2 and 4 will fail because of task 3 and 5.

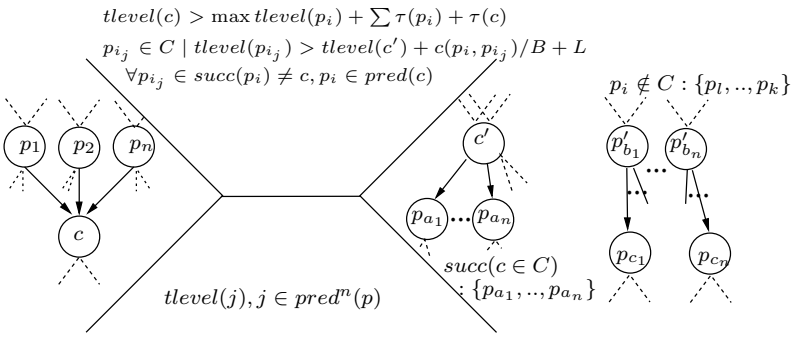


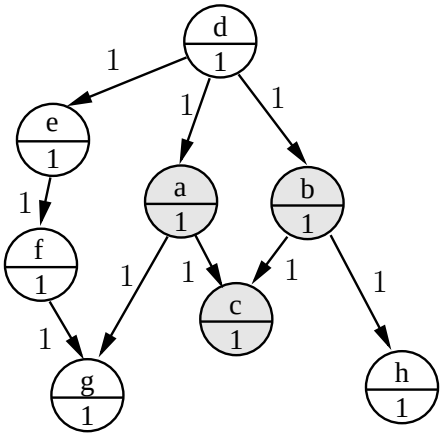
Figure 7.8. The *mergeallparents2* rule.

mediate predecessors for which the condition fails is also replicated. These enhancements to the old rule are shown in Figure 7.8. This new rule is named *mergeallparents2*.

As an example, consider the task graph in Figure 7.9. We set bandwidth, $B = 1$ and latency, $L = 9$ and try to apply the rule to task c. The cost of communicating from task b to task c becomes $c_{b,c} = L + B * 1 = 9 + 1 = 10$. The top level value of all tasks in the task graph are presented in Figure 7.9(b).

The first condition, $tlevel(c) > \max\{tlevel(a), tlevel(b)\} + \tau(a) + \tau(b) + \tau(c)$ becomes $22 > 11 + 1 + 1 + 1$. Thus the first condition is true.

The top level of the newly created task c' is given an upper bound by $tlevel(c') \leq \max\{tlevel(a), tlevel(b)\} + \tau(a) + \tau(b) + \tau(c) = 11 + 3$ which is then used to check the conditions of the *other immediate successor* tasks g and h. The reason this being an upper bound of the top level for the new task c'



(a) Task graph

Task id	*top level*
a	11
b	11
c	22
d	0
e	11
f	22
g	33
h	22

(b) Top level for all tasks

Figure 7.9. Example of merging parents applying the improved rule called mergeallparents2.

is if two immediate predecessors of c have a common immediate predecessor, since task a and b has the common immediate predecessor d, the top level can be reduced even further for the new task c' since the communication messages then are combined.

The reason for not calculating a tighter (or even exact) bound for the new task c' is that such calculations are too computationally expensive. The same relaxation of the upper bound also occurs on the first condition of the rule due to the same reason. However, the exact calculation could be seen as yet another improvement of the rule, even though experiments have shown that this improvement is not necessary in practice, since the condition will be fulfilled in most cases anyway.

For task g we get $tlevel(g) > tlevel(c') + 1/B + L$ which gives $33 > 14 + 10$, for bandwidth, $B = 1$ and latency, $L = 9$. Thus the task g has other immediate successors *not* affected by the merge, since the condition is true, giving task $a \in C$.

For task h we get $tlevel(h) > tlevel(c') + 1/B + L$ giving $22 > 14 + 10$ which is not true, giving task $b \notin C$.

After the transformation has been performed we get the task graph pre-

sented in Figure 7.10. The three tasks a, b, c are merged and since the condition failed for task b it is replicated to prevent an increase of the top level for task h.

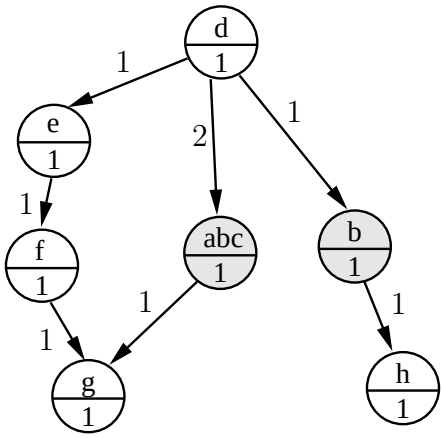


Figure 7.10. The resulting task graph after *mergallparents2* has been applied to task c.

Improved Rule for Split Nodes (duplicateparentmerge2)

The *duplicateparentmerge* rule for split nodes can be improved in a similar manner. First we allow other immediate predecessors to the immediate successor tasks of the split node p, which will increase the potential of successfully matching the pattern. In this case if the merge is successful the top level could be increased for these nodes since the execution cost of the split node $\tau(p)$ must be taken into consideration.

But as for the *mergeallparents2* rule we can further improve this by again dividing these *other immediate successor* nodes into two disjunct sets, those that fulfill the condition of not increasing the top level and those that do not fulfill this condition.

Those that fulfill the condition will be merged into the split node p and those that do not fulfill the condition will be left as is, i.e., still having the split node p as a immediate predecessor. This also implies that the split node itself must be kept if the set of tasks not fulfilling the condition is nonempty. The enhanced rule is named *duplicateparentmerge2* and is shown in Figure 7.11.

The condition of the new rule consists of the condition from the *duplicateparentmerge* rule that the top level should not be increased by merg-

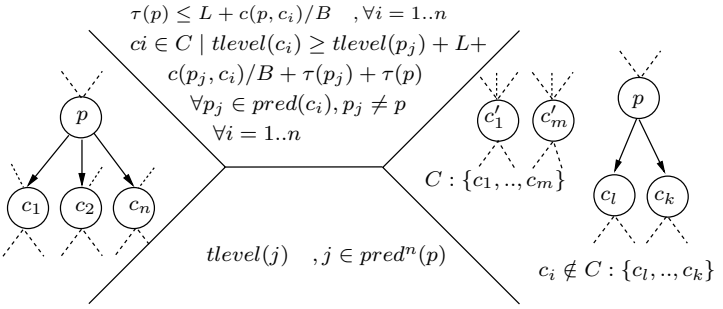


Figure 7.11. The *duplicateparentmerge2* rule.

ing *all* immediate predecessors with the task p. This gives the condition $\tau(p) \leq L + c(p, c_i)/B) \quad , \forall c_i \in succ(p)$.

The second condition states that *other immediate predecessors* (p_j) of the successors of p (c_i) should not increase their top level, giving the condition $tlevel(c_i) > tlevel(p_j) + L + c(p_j, .c_i)/B + \tau(p)$.

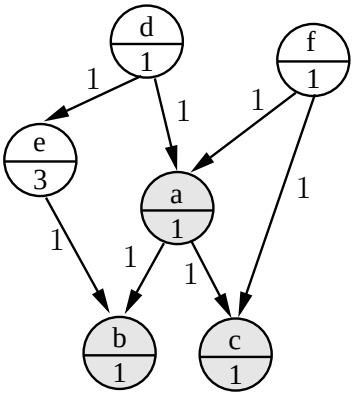
As an example, we illustrate the rule by applying it to the task graph in Figure 7.12. The first condition, $\tau(a) < L + c(a, b)/B$ and $\tau(a) < L + c(a, c)/B$ holds.

The second condition will then check the *other successors* of task b and c. For task b we get $tlevel(b) > tlevel(e) + L + c(e, b)/B + \tau(e) + \tau(a)$ which does not hold for $B = 1$ and $L = 9$. Thus, task a can not be replicated and merged into task b since that would increase its top level (going through task e).

For task c we get $tlevel(c) > tlevel(f) + L + c(f, c)/B + \tau(f) + \tau(a)$ which holds for $B = 1$ and $L = 9$. Therefore task a can be replicated and merged together with task c. The resulting task after the merge is depicted in Figure 7.13.

7.5 Extending for Malleable Tasks

A malleable task is as explained in Section 2.1 a task that can be executed on a single processor one or on several processors. The execution cost for such a task is therefore a decreasing function of the number of processors it will execute on, assuming that the computation to communication ratio on the multiprocessors allows speedup in that case (sometimes execution is slower using more processors). The actual number of processors it will execute on can be determined both at static scheduling time, as a part of the scheduling problem, or even at runtime by a dynamic scheduler. It would however not



(a) Task graph

Task id	*top level*
a	11
b	24
c	22
d	0
e	11
f	0

(b) Top level for all tasks

Figure 7.12. Example for replicating parent and merging with the improved *duplicateparentmerge2* rule.

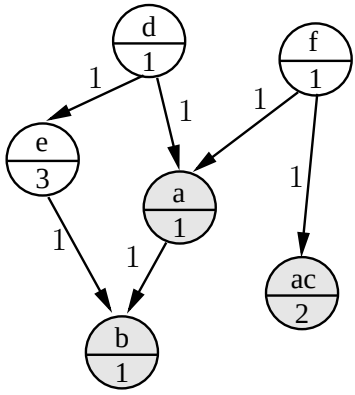


Figure 7.13. The resulting task graph after *duplicateparentmerge2* has been applied to task a.

make much sense to determine this at task merging time, since the approach of task merging is independent of the scheduling problem, i.e., no scheduling

decisions are made during the task merging process.

To include malleable tasks in the task merging process we first need to recognize which tasks are malleable. This can be achieved with an attribute indicating whether a task is malleable or not. The task merging rules will then be enhanced with conditions concerning malleable tasks. These conditions are based on a few simple rules.

- A malleable task should not be allowed to be replicated. This is justified by two reasons. First, a malleable task is probably never a small task, i.e., a task with low execution cost compared to the communication costs of the task graph, therefore making it too costly to replicate. Secondly, replicating a malleable task would lead to a substantial increase in the number of processors required. Since the task merging approach is performed prior to scheduling, there is no information available on how many processors are available. Therefore the decision to replicate a malleable task can not be fully made.

- Two malleable tasks should not be merged. This makes sense since we do not know how many processors each of the two malleable tasks will require. It is not certain that they both require the same number of processors. Therefore, merging two malleable tasks together is not feasible, mostly because the scheduling algorithm in that case still would need to be able to determine the number of processors for each of the two internally merged tasks anyway.

With these rules as a baseline, the inclusion of malleable tasks into the task merging rules are quite straightforward. The *singlechildmerge* rule will have the additional condition that not both tasks should be malleable, giving the extra condition $\neg malleable(c) \wedge malleable(p) \vee malleable(c) \wedge \neg malleable(p)$, where malleable is a function returning true for malleable tasks and false for non-malleable tasks.

The *mergeallparents2* rule can also allow some of the tasks involved to be malleable. The problem in this case is to calculate the top level of the malleable task, since the execution time is dependent on how many processors gets allocated to the task. There are two possibilities for solving this problem. The first one is to make a conservative assumption and use the execution cost for executing the task on a single processor. However, this can in some cases prevent the rule from being applied. The alternative is to before the merge determine how many processors each malleable task should execute on and then use the estimated cost for the determined number of processors when calculating the top level value. In this case we get a more accurate execution cost but the downside is that we need to know the number of processors allocated

for each task, which can only be known at the scheduling time. Thus, giving a dependency between the task merging phase and the scheduling phase.

The same problem arises when we look at the *duplicateparentmerge2* merge rule. We can not exactly determine the execution cost of the malleable task without knowing how many processors it will execute on. For this case we also have to prevent malleable tasks from being replicated, i.e., the parents of the targeted child in this rule can not be malleable. It would make no sense to allow a malleable task to be replicated. The only exception would be if the malleable task had the fastest execution time for executing on a single processor, i.e., the task is so small that it executes fastest on one processor. Then it could be considered for replication, since the effect of the replication is then more limited. But a task can also have the fastest execution time for one processor but still be relatively large. Therefore it might be needed to instead use a absolute threshold for when such a malleable task might be replicated or not.

The extension to the task merging method to malleable tasks is feasible by slightly extending the conditions of the rewrite rules. Malleable tasks is a way of extending static task graphs to include more dynamic information. For instance, a simple loop in a Fortran program that can easily be parallelized can not be represented using task graphs in its present form. However, the same loop can be treated as a malleable task, allowing programs containing such loops to be parallelized by using task graphs. These programs can of course also gain from having a task merging phase when they are parallelized.

7.6 Termination

One of the most important and desired properties of a graph rewrite system is if it terminates for any given input graph. If the task merging graph rewrite system would not terminate it would be of small or no practical use. Hence this property is of uttermost importance for the task merging GRS.

Fortunately, our task merging GRS does terminate for any input graphs. This can easily be shown since no rule of the task merging GRS will (in total) increase the number of vertices or edges of the task graph. Thus every application of a rule will decrease the size of the graph monotonically. Therefore, the task merging GRS will always terminate for all input task graphs. In the worst case the GRS will terminate when there is only one task left in the task graph, or only tasks without any edges in the task graph.

Even if the termination is guaranteed it might be useful to interrupt the task merging algorithm before it terminates. This can be done since each transformation step results in a valid task graph. In other words the task merging algorithm is incremental by nature, in each step making a small im-

provement by increasing the granularity of the task graph. This property can be of practical importance since it allows a tool to interrupt the task merging if it takes too long time, but still being able to continue with task scheduling.

7.7 Confluence

The confluence property of a graph rewrite system is important if the rules should be allowed to be applied in any order. As presented before, a graph rewrite system is confluent if the order of application of transformations does not affect the result.

To formally prove that the task merging GRS is confluent can be a tedious and complex task, and is outside the scope of this thesis work. One problem of proving this property is that one task merging transformation in a totally different part of a task graph can for instance potentially change the condition whether a task belongs to the set C or not in the *duplicateparentmerge2* rule. Thus, affecting if that particular task should be replicated and merged into the successor task or not. It can perhaps be feasible to show that the order would not matter if the conditions of *other immediate predecessor* tasks were neglected, i.e., considering the rules in the first attempt. However, to prove this for the complete set of enhanced rules is not an easy task.

However, the confluence property can be established to a certain degree by empirically studies on a large set of examples. For this purpose we use the standard task graph set (STG), which consists of a large number of randomly generated task graphs. We applied the task merging graph rewrite system to the STG using different priorities on the transformation rules, to force the graph rewrite system to apply the task merging rules in different order for different executions of the algorithm. The output is then compared between executions using different priorities to establish the confluence property. Since we have two sets of rules, the rules in the first attempt and the improved rules, the confluence property is investigated for both of these graph rewrite systems.

We ran the task merging algorithm using two different priorities on the two systems. For the simple version we ran:

1. *DuplicateParentMerge* > *MergeAllParents* > *SingleChildMerge* meaning that *DuplicateParentMerge* has the highest priority.

2. *MergeAllParents* > *SingleChildMerge* > *DuplicateParentMerge* meaning that *MergeAllParents2* has the highest priority.

And for the enhanced rules:

1. *DuplicateParentMerge2* > *MergeAllParents2* > *SingleChildMerge* meaning that *DuplicateParentMerge2* has the highest priority.

2. *MergeAllParents2* > *SingleChildMerge* > *DuplicateParentMerge2* meaning that *MergeAllParents2* has the highest priority.

The resulting numbers of nodes after the merge and the parallel time, see Section 2.1.2, before and after the merge are presented in Figure 7.14 for the first priority order and in Figure 7.15 the second priority order[1]. The two results are identical. In these measurements we used $B = 1$ and the latency varied between 10 and 1000. Also, the execution costs typically varies between 1 and 100 while the execution cost (data size) is set to 1 since they are not given in the STG files. For the STG measurements all measured data are the same between the two different priority orders. From these results it is tempting to draw the conclusion that the task merging graph rewrite system is confluent.

7.7.1 Non Confluence of the Enhanced Task Merging System

However, when looking at more fine grained task graphs with a large number of reductions taking place, it is evident that this is not the case. If we consider an example generated from Modelica code using the `PreLoad` example from the Modelica Standard Library, where we have removed the source and the sink nodes, the two priority orders give different results for the enhanced rules. Thus, this model serves as a counter-proof of confluence of the enhanced task merging system.

7.8 Results

We have proposed a new way of merging tasks in a task graph with the objective of increasing the granularity of the task graph without increasing the parallel time of the task graph, called ATMM (Aronssons Task Merging Method). This approach uses a graph rewrite system consisting of a few simple rules for when tasks can be merged together, potentially increasing the granularity of the task graph. The results are very promising, especially for task graphs that are very fine grained. This becomes evident when such task graphs are scheduled or clustered using standard scheduling and clustering algorithms. By combining ATMM with such standard schedulers a substantial speedup of the generated schedule is achieved. Results from such measurements are presented below.

[1]These measurements mentioned here are only a subset of the measurements performed. for a complete list, see appendix A.

B and L	*Name*	*PT before*	*PT after*	*Node Reduction*
$B = 1, L = 10$	rand0161.stg	520	366	0
	rand0167.stg	252	252	0
	rand0106.stg	293	241	0
	rand0122.stg	577	432	1
	rand0053.stg	165	130	0
	rand0163.stg	809	496	1
	rand0166.stg	93	59	11
$B = 1, L = 100$	rand0161.stg	2860	1130	6
	rand0167.stg	1062	926	0
	rand0106.stg	1100	736	3
	rand0122.stg	3187	1365	7
	rand0053.stg	975	655	2
	rand0163.stg	4679	1928	3
	rand0166.stg	543	170	48
$B = 1, L = 1000$	rand0161.stg	26260	8341	11
	rand0167.stg	9162	6075	3
	rand0106.stg	9200	4112	2
	rand0122.stg	29287	11136	17
	rand0053.stg	9075	4039	14
	rand0163.stg	43379	14193	15
	rand0166.stg	5043	116	55

Figure 7.14. First priority order for enhanced task merging on STG (Standard Task Graph Set) subset. PT is the parallel time of the task graph.

The computational complexity of ATMM has also been investigated. Attempts to improve the complexity have been made and results from these attempts are also presented below.

Another result of ATMM is that the method can easily be included in existing scheduling and parallelization frameworks, since the input to the merging algorithm is a task graph and the result is another more coarse grained task graph. Thus, the merging algorithm can be plugged in before the scheduling phase in any tool, providing better scheduling results for fine grained task graphs. This can also increase the efficiency of such tools, since it allows the tools to initially build task graphs at a finer granularity level (which is substantially reduced to more coarse grained), thus potentially increasing the amount of parallelism in the task graphs resulting in shorter schedules.

B and L	*Name*	*PT before*	*PT after*	*Node Reduction*
$B = 1, L = 10$	rand0161.stg	520	366	0
	rand0167.stg	252	252	0
	rand0106.stg	293	241	0
	rand0122.stg	577	432	1
	rand0053.stg	165	130	0
	rand0163.stg	809	496	1
	rand0166.stg	93	59	11
$B = 1, L = 100$	rand0161.stg	2860	1130	6
	rand0167.stg	1062	926	0
	rand0106.stg	1100	736	3
	rand0122.stg	3187	1365	7
	rand0053.stg	975	655	2
	rand0163.stg	4679	1928	3
	rand0166.stg	543	170	48
$B = 1, L = 1000$	rand0161.stg	26260	8341	11
	rand0167.stg	9162	6075	3
	rand0106.stg	9200	4112	2
	rand0122.stg	29287	11136	17
	rand0053.stg	9075	4039	14
	rand0163.stg	43379	14193	15
	rand0166.stg	5043	116	55

Figure 7.15. Second priority order for enhanced task merging on STG subset.PT is the parallel time of the task graph.

7.8.1 Increasing Granularity

ATMM is quite successful at increasing the granularity of task graphs. Table 7.1 below gives the increase in granularity for a subset of task graphs from the Standard Task Graph Set. The first ten lines presents results on task graphs when $B = 1$ and $L = 10$. For the next ten lines the latency has been increased to 100, and the final ten lines have a latency of 1000. In all cases the granularity of the task graph increases, due to merging of tasks.

B and L	*File*	*Granularity before*	*Granularity after*
$B = 1, L = 10$	rand0161.stg	0.704545	0.932727
	rand0167.stg	0.72	0.830909
	rand0106.stg	0.879091	1.09727
continued on next page			

continued from previous page			
B and L	*File*	*Granularity before*	*Granularity after*
	rand0122.stg	0.74	1.08081
	rand0053.stg	0.501818	0.692727
	rand0163.stg	0.714545	0.991736
	rand0166.stg	0.715015	0.993007
	rand0163.stg	0.714545	0.991736
	rand0152.stg	0.812672	1.10655
	rand0157.stg	0.805455	1.01364
$B = 1, L = 100$	rand0161.stg	0.0767327	0.194727
	rand0167.stg	0.0784158	0.141881
	rand0106.stg	0.0957426	0.212814
	rand0122.stg	0.0805941	0.221197
	rand0053.stg	0.0546535	0.141844
	rand0163.stg	0.0778218	0.148909
	rand0166.stg	0.077873	0.271915
	rand0163.stg	0.0778218	0.148909
	rand0152.stg	0.0885089	0.248086
	rand0157.stg	0.0877228	0.199312
$B = 1, L = 1000$	rand0161.stg	0.00774226	0.0507571
	rand0167.stg	0.00791209	0.0453149
	rand0106.stg	0.00966034	0.0527736
	and0122.stg	0.00813187	0.0472764
	rand0053.stg	0.00551449	0.0372748
	rand0163.stg	0.00785215	0.03327
	rand0166.stg	0.00785731	0.0338485
	rand0163.stg	0.00785215	0.03327
	rand0152.stg	0.00893046	0.0723083
	rand0157.stg	0.00885115	0.041282

Table 7.1. Granularity measures on task graphs from STG.

A more interesting test of the task merging approach is to instead use task graphs from simulation code. Table 7.2 presents some task graphs generated from simulation code from Modelica models. Some of the task graphs are built in the earlier prototype of the parallelization tool, see Chapter 6.

B and L	*Model*	*Granularity before*	*Granularity after*
$B = 1, L = 100$	PressureWave	0.000990	0.106
	PreLoad	0.00990	0.147277
continued on next page			

continued from previous page			
B and L	*Model*	*Granularity before*	*Granularity after*
$B = 1, L = 1000$	PressureWave	0.0000990	0.0562
	PreLoad	-	-

Table 7.2. Granularity measures on task graphs from Modelica models.

7.8.2 Decrease of Parallel Time

A second way to measure how successful the task merging approach is can be to measure the parallel time of a task graph when it has been *clustered* using the DSC [88] algorithm, and compare that to a task graph that has first been merged using our task merging method. This will indicate whether the task merging can further improve the clustering work. For these measurements we have used two task graphs, one from the simulation code for a Modelica model and the other from a butterfly calculation task graph. Table 7.3 presents the results from these measurements. Both examples gain from first applying ATMM as a complement to the DSC algorithm. For higher latency values the gain is larger than for lower values. But even for the case when the latency is one, task merging still improves the results.

B and L	*Model*	*PT with DSC*	*PT with TM + DSC*
$B = 1, L = 1$	PreLoad	31	24
$B = 1, L = 10$	PreLoad	148	72
$B = 1, L = 100$	PreLoad	1318	242
$B = 1, L = 1000$	PreLoad	13018	94
$B = 1, L = 1$	Butterfly FFT	33	25
$B = 1, L = 10$	Butterfly FFT	105	45
$B = 1, L = 100$	Butterfly FFT	825	148
$B = 1, L = 1000$	Butterfly FFT	8025	1132

Table 7.3. Granularity measures on task graphs from two applications. PT is the parallel time of the task graph.

7.8.3 Complexity

The computational complexity of a graph rewrite system depends on the rewrite rules and on the termination of the rewrite process. For instance, if only one transformation is made before the GRS terminates, the amount of computational work required would be $O(n)$ for a graph containing n nodes. This is because in the worst case we will have to go through all nodes before finding the node for which the transformation is performed.

But, for a GRS like ATMM, where there is always a reduction of tasks in the task graph, the worst case complexity is higher. In the worst case, ATMM will succeed in making a total reduction of the task graph from n tasks to a single task. This will require $O(n+(n-1)+(n-2)+...+1)$ computing steps, giving a worst case computational complexity of $O(n^2)$.

In practice however, this is seldom the case. The GRS will terminate before a complete reduction is made. This will of course depend on the granularity of the input task graph and of the latency and bandwidth parameters. For instance, for a coarse grained task graph, ATMM will terminate after a single traversal, giving a linear computational complexity. But of course the complexity also depends on the number of edges of the task graph. If there are many edges, each task will have many immediate predecessors and successors, giving a more costly condition calculation.

7.9 Discussion and Future Directions

The task merging approach presented in this chapter is a new contribution in the area of task clustering and task scheduling, called ATMM. It is of special importance for automatic parallelization where the task graph granularity is not directly controlled by an experienced programmer but instead is influenced by the tool itself. Theoretical aspects on the graph rewrite system of task merging such as termination and confluence builds a ground for the work, which is strengthened by practical experience in the automatic parallelization tool for Modelica simulation models.

There are several directions for future work within this area. There are certainly large possibilities to reduce the computational complexity of the task merging algorithm. One could investigate if it might be enough to traverse the task graph only once when applying the pattern matching. How to implement such a traversal but still guarantee the same reduction as for the unoptimized task merging approach is an unanswered question.

Other possibilities are to implement heuristics as conditions with the aim of reducing the complex condition evaluations to a simpler form. This could substantially reduce computational complexity in practice and would with a careful design of the heuristics still produce task graphs with a proper balance between granularity and parallelism.

Yet another way to decrease the execution time of ATMM can be to parallelize the task merging algorithm itself. It makes sense to use the same parallel computer for compiling the code in the parallelizing compiler as when executing the compiled code. Such a parallelization would for instance try to perform the subgraph pattern matching and calculation of conditions in parallel. The actual transformation would be performed sequentially. Such

an implementation could for instance distribute the nodes and edges over the processors to perform the matching and transformation. The tricky part is how to communicate and/or migrate nodes and edges between processors.

The idea of using the computational power of the parallel computer also for compilation can of course also be expanded into other parts of the parallelization tool, or in our application also the Modelica compiler itself. However, most of the compilation process is sequential by nature and not much can be gained by such a parallelization in general. But there might be cases where a parallel implementation might speed up compilation.

The task merging approach can be further evaluated in the context of malleable tasks if it is implemented in an automatic parallelizing compiler for a wide spread scientific computing programming language such as Fortran. There are several such research compilers available, some of them can be found at [14]. The task merging approach could then be evaluated for a much larger set of application examples, including for instance dynamic scheduling of malleable tasks.

7.10 Summary

In this chapter we have presented a new approach for merging nodes in a task graph to increase the granularity of the task graph, called ATMM. The approach is based on graph rewrite systems, where a set of transformation rules are applied to a graph until the conditions of the rules are no longer fulfilled, and the graph rewrite system terminates. This approach gives an easy-to-understand and flexible solution to the problem of increasing the granularity of a task graph by merging tasks. The approach also includes task replication which has been shown to substantially improve the results of scheduling algorithms [41, 58].

ATMM has been tested on several examples including task graphs generated from simulation code produced from Modelica models as well as random tasks from the STG (Standard Task Graph Set).

Chapter 8

Applications

This chapter presents a few application examples written in the Modelica modeling language that have been parallelized using OpenModelica.

8.1 Thermal Conduction

Thermal conduction deals with transportation of thermal energy through conduction. This assumes that the atoms or molecules are in a fixed solid form and are not moving around (like in a fluid or gas). Thus thermal conduction can be used when modeling the temperature in a metal rod or a metal plate.

The heat conduction equation for three dimensions is

$$\frac{\delta T}{\delta t} = a\left(\frac{\delta^2 T}{\delta x^2} + \frac{\delta^2 T}{\delta y^2} + \frac{\delta^2 T}{\delta z^2}\right) \tag{8.1}$$

where $T(x, y, z, t)$ is the temperature at position (x, y, z) at time t.

In this example we model the temperature in a thermal plate, which gives us the heat equation for two dimensions, the x and y directions, resulting in the PDE equation in Equation 8.1 below.

$$\frac{\delta T}{\delta t} = a\left(\frac{\delta^2 T}{\delta x^2} + \frac{\delta^2 T}{\delta y^2}\right) \tag{8.2}$$

Since standard Modelica cannot yet describe Partial Differential Equations (for ongoing work to introduce PDEs in Modelica, see [69]), the model has to be discretized by hand in the space dimensions, resulting in the following

Modelica code:

```
model HeatedPlate2D
    parameter Integer n=8;
    parameter Real L=1;
    parameter Real k=0.001;
    Real u[n,n](start=fill(20,n,n));
    Real h=L*L/(n*n);
equation
   // Boundary conditions
   for y in 1:n loop
     der(u[n,y])=-0.167;
     u[1,y]=80;
   end for;
   for x in 2:n-1 loop
     der(u[x,n]) = -0.001*u[x,n];
      u[x,1]=40+20*cos(2*Modelica.Constants.PI/n*x);
   end for;

   // Heat equation. (central difference)
   for x in 2:n-1 loop
     for y in 2:n-1 loop
     der(u[x,y]) = (k*(u[x,y-1]-2*u[x,y]+u[x,y+1]))/h
      + (k*(u[x-1,y]-2*u[x,y]+u[x+1,y]))/h;
     end for;
   end for;
end HeatedPlate2D;
```

The model has the following boundary conditions:

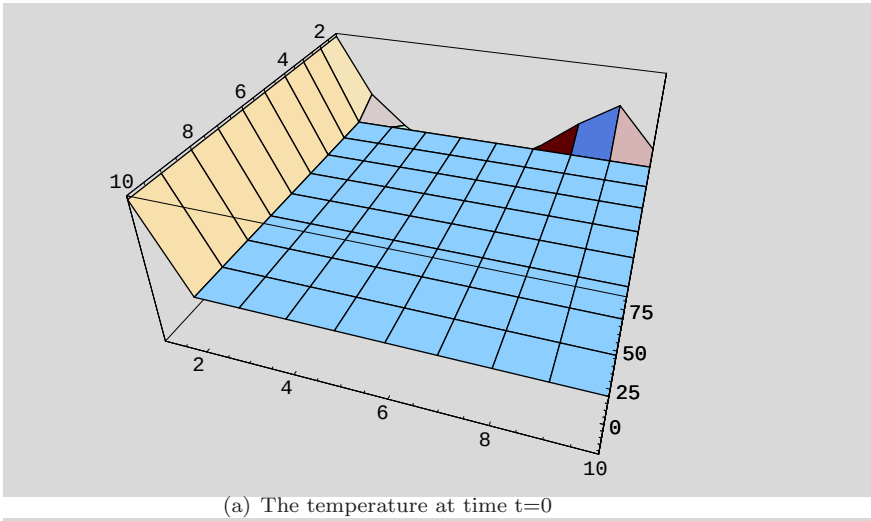
- The east boundary has a constant temperature of 80 degrees, e.g. given from a heating element, i.e. `u[1,y]=80`.
- The west boundary has a partially insulated boundary with a constant decrease of temperature by 0.167 degrees per second, i.e. `der(u[n,y])=-0.167`.
- The north boundary is also partially insulated but the decrease in temperature is dependent on the absolute temperature, i.e. `der(u[x,n])=-0.001*u[x,n]`
- The south boundary has a constant temperature that is varying across the boundary as a *cos* wave.

These boundary conditions are not chosen to be physically realistic, instead they illustrate different usage of the Modelica language.

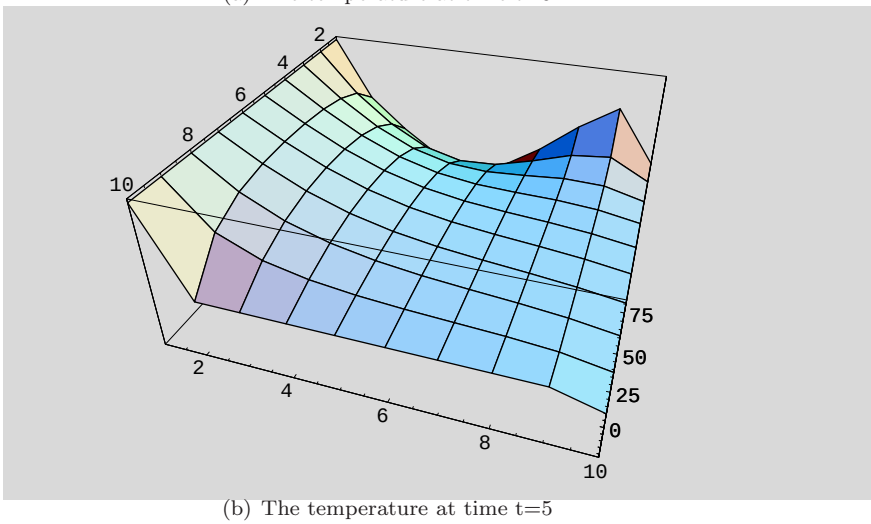
If we simulate the model and plot the temperature of the plate at different points using the MathModelica tool [43], we get the result depicted in Figure 8.1. In Figure 8.1(a) the temperature is shown at the start of the simulation ($t = 0$) and in Figure 8.1(b) the temperature is shown when $t = 5$. The plots are done using Mathematica.

8.1.1 Parallelization

When investigating the equations of the heated plate model it becomes evident that the amount of computation per state variable is not sufficiently large to give any speedup with the current parallelization scheme. The reason for this is that it costs more to distribute the state variables to the slave processors and send the results back compared to performing the actual computation. However, since this example is so well structured it can easily be parallelized using a distributed solver instead. A distributed solver will for each processor have a solver that works on a subset of all state variables. In that case the only amount of data that must be sent between processors are the values of the state variables of the *boundaries* of each state variable set. For example, the PVODE solver [11] (parallel ode solver) could be used for this.



(a) The temperature at time t=0



(b) The temperature at time t=5

Figure 8.1. Simulation of a heated plate in two dimensions

.

8.2 Thermofluid Pipe

There has been developed a Modelica package for thermo fluid models called `ThermoFluid` [79]. It uses lumped control volumes, e.g. with behavioral equations describing a complete volume, or one dimensional discretized models,

where the control volume is discretized along one dimension. The later ones are actually PDE models which are discretized using the finite volume method [79] and these models are of particular interest to parallelize, since the amount of computational work (i.e. the size of the problem) performed in the simulation can easily be controlled.

The model used for benchmarking is a system of pipes connected to a steam source, where steam is fed in to the system, and a steam sink, where the steam leaves the system. The pipe system consists of a series of pipes of different dimensions. The model describes how a pressure wave of steam is transported through the system, leading to the sink. Below is the Modelica text for the model.

```
model PressureWave
  extends ThermoFluid.Icons.Images.Demo;
  Components.Air.Reservoirs.AirSourceD_pT
    Source(pdrop(steadyFlowInit=false),
           dp0=1,
           mdot0=4.0,
           A = 0.00785,
           T0 = 330.0);
  Components.Air.Pipes.PipeDD Pipe(n=50,
                              geo(L=1.0),
                              char(dp0 = 1000),
                              generateEventForReversal=false,
                              init(steadyPressure=false,
                              p0=linspace(1.2e5,1.2e5,50))) ;
  Components.Air.Reservoirs.AirResD_pT Sink;
equation
  connect(Source.b, Pipe.a);
  connect(Pipe.b, Sink.a);
end PressureWave;
```

The `Pipe` component is discretized into `n` pipe elements, which provides an easy way of controlling the size of the model.

8.3 Simple Flexible Shaft

A shaft transporting torque from a motor to other mechanical components in a system, like a propeller or a wheel, must in some cases be modeled in substantial detail. If the shaft is exposed to large forces it might be needed to model the entire shaft not as a rigid body propagating torque but instead as

a flexible body that changes shape when large loads are put on it. Models of such flexible shafts can be made at different levels of detail. It can for example be modeled as a one dimensional discretization of the shaft or in a full three dimensional PDE formulation of the shaft. In this application example we consider modeling a shaft using a one-dimensional discretization scheme.

To model a shaft with flexibility in one dimension is quite easy using Modelica and the arrays-of-components modeling technique. First we create the model component which will be instantiated as an array element in our shaft model.

```
model ShaftElement "Element of a flexible 1-dimensional shaft"
   import Modelica.Mechanics.Rotational.Inertia;
   import Modelica.Mechanics.Rotational.Interfaces;

   extends Interfaces.TwoFlanges;

   Inertia inertia1;
   NonLinearSpringDamper springDamper1(c=5,d=0.11);
equation
   connect(inertia1.flange_b, springDamper1.flange_a);
   connect(inertia1.flange_a,flange_a);
   connect(springDamper1.flange_b,flange_b);
end ShaftElement;
```

In this model we have used the Modelica Standard library and its package for one dimensional rotational mechanics, the `Rotational` package. The `ShaftElement` model inherits the `TwoFlanges` interface, giving it two connectors for transporting rotational energy, `flange_a` and `flange_b`. These connectors are defined as (both `Flange_a` and `Flange_b` have the same definition):

```
connector Flange_a
  SI.Angle phi "Absolute rotation angle of flange";
  SI.Torque tau "Cut torque in the flange";
end Flange_a;
```

The `SI` package is explicitly imported with the following renaming import statement:

```
import SI=Modelica.SIunits;
```

Thus, the torque and the rotation angle is connected from each shaft element to the nearby shaft element using Modelica connectors. The internal model for each shaft element consist of a load component using the `Inertia` model. The elasticity of the flexible shaft is modeled using a `SpringDamper` model which we have replaced with a non-linear equation, giving a model called `NonLinearSpringDamper`. There are many different formulations of a non-linear spring. In this example we simply add a (non-physical) term to get a non-linear equation in the model.

To use this shaft element in a model of a complete shaft one needs to instantiate an array of `ShaftElement`:s and write a for loop (repetitive equation) to connect these elements to the nearby elements:

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

Finally, a small test model to test the model:

```
model ShaftTest
  FlexibleShaft shaft(n=5);
  Modelica.Mechanics.Rotational.Torque src;
  Modelica.Blocks.Sources.Step c;
equation
  connect(shaft.flange_a,src.flange_b);
  connect(c.outPort,src.inPort);
end ShaftTest;
```

This model is depicted in Figure 8.2 below.

We use a step source as input to a torque generator which is then connected to one end of the shaft. If we simulate this model using $n = 30$ and two different parameter values for each spring damper of the shaft we get the plots shown in Figure 8.3.

The figure clearly shows that the harder damping that is put on each element of the flexible shaft, the sooner the system will reach a stable state.

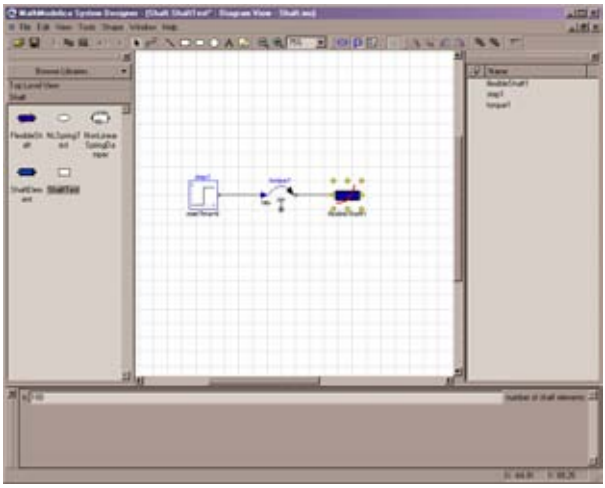
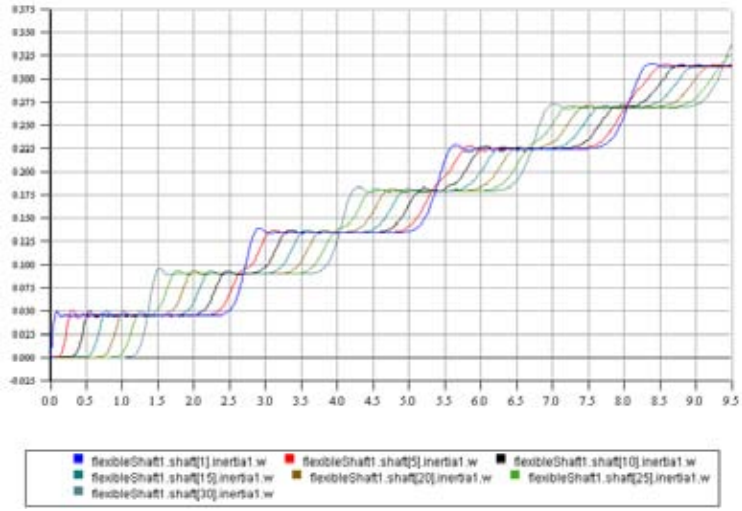


Figure 8.2. The ShaftTest model in the MathModelica model editor.

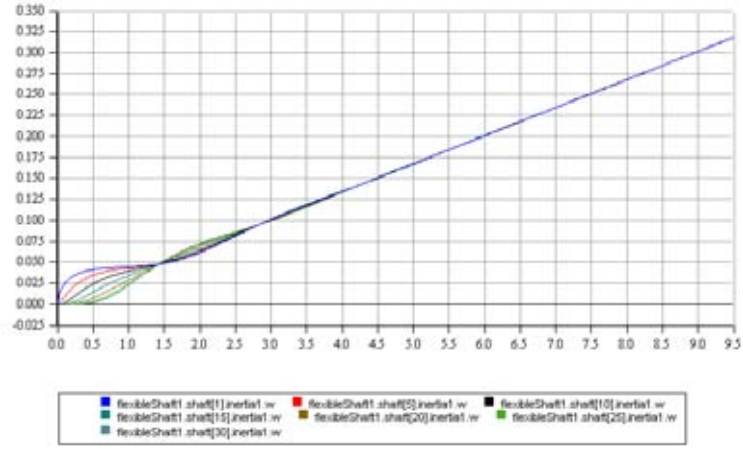
Typically, real application models of flexible shafts would have different spring constants and inertias on different parts of the shaft, depending on its geometry and material composition. The shaft would also probably consist of more internal components that just inertias and springs and dampers, like for example models of roller bearings or friction bearings, etc.

8.4 Summary

In this chapter we have presented a few application examples which could benefit from parallelization. All of them include a discretization of equation using array of components in Modelica, even though the parallelization scheme itself does not rely on that. Larger and more complex models however can as of today not fully be translated by the OpenModelica compiler and can for that reason not be parallelized by this research tool. Such models can for instance be a complete plant model of a process industry, or detailed mechanical system such as a car with engine model, electronic control system, and multi-body suspension.



(a) Plot of torques on flange_b for each shaft element with $c = 500$ and $d = 1.5$



(b) Plot of torques on flange_b for each shaft element with $c = 500$ and $d = 150$

Figure 8.3. Plots of torques on the shaft

Chapter 9

Application Results

This chapter gives results from using the different parallelization techniques for parallelization of the application examples.

9.1 Task Merging

The proposed task merging algorithm in Chapter 7 has been evaluated in two ways. It first have been tested in the DAGS framework, presented in chapter 4, using both task graph built from Modelica models and from tasks graphs of the Standard Task Graph Set [77]. The results from some of these examples can be found in Appendix A. The main conclusion from these tests is that provided that the latency parameter is large in relation to the bandwidth i.e., a fine grained task graph, the task merging algorithm succeed quite well in increasing the granularity (by merging tasks) for increasing values of the parallel execution time. The results are better the larger latency is chosen, as expected.

Moreover, the task merging algorithm has also been implemented in the ModPar tool, presented in Chapter 6, as part of the automatic parallelization scheme. These results are presented below as measurements of speedup of execution times.

9.2 Parallelization of Modelica Simulations

Parallelization and performance measurements of simulation of application models has been done using both the DSBPart and the ModPar tools. Results are presented below in the following two sections.

9.3 DSBPart Experiments

In this section we present results gained from using the DSBPart tool and early experiments made on task graphs produced by this tool.

9.3.1 Results From the TDS Algorithm

Due to the linear clustering technique used by the TDS algorithm, combined with the fine grained task graphs produced by our tool, the TDS algorithm does not work well for the fine task graph generated in our tool (as for fine grained task graphs in general). As an example, we will use the small task graph shown in Figure 9.1. For that task graph, the TDS algorithm produces the values found in Figure 9.2. The table shows the earliest completion time (ect) of the final node (i.e. the exit node of the DAG) for a set of different communication cost values of the task graph. The earliest completion time is a measure of the parallel time, provided that the number of processors required by the algorithm is available. However, in practice the number of processors required by the TDS algorithm is usually too large. For instance, the robot example requires 173 processors when applying the TDS algorithm.

The table shows that in order for the TDS algorithm to produce a schedule that according to the delay model has a computed speedup > 1, the communication cost must be around 10 or less in comparison to the computational size of the tasks. For the task graph produced by the `PreLoad` example, the tasks are almost exclusively arithmetic expressions, thus the communication cost of sending a scalar value should be only at most ten times more expensive compared to performing an arithmetic operation on two scalar variables. This is a far more demanding latency requirement than what most real multi processor architectures can deliver today.

We also ran the TDS algorithm on a larger example, the simulation code from the robot example in the Modelica Standard Library, with inline integration and mixed mode integration, see section 3.2. The result for that example is shown in Figure 9.3, using the same set of communication costs. For this example, the results are a bit better. Computed speedup > 1 according to the delay model is achieved if the communication cost is around 500 or less. One reason for this improvement could be that the simulation code from the robot example contains larger tasks, for instance to solve systems of equations, thereby increasing the average granularity.

In the above results we have not looked at fixing the number of processors to a specific value. One reason for this assumption is that by allowing a unlimited number of processors, we can compare the result with other clustering algorithms like for instance the DSC algorithm. This assumption will produce the best possible results from the TDS approach, i.e. a lower time bound.

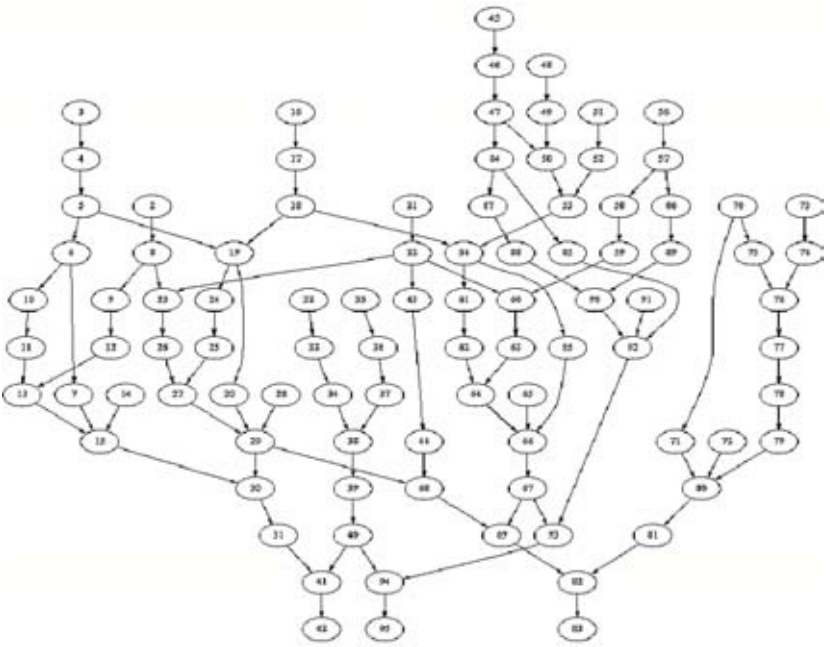


Figure 9.1. The task graph built from the code produced from the `PreLoad` example in the mechanics part of Modelica Standard Library.

Total sequential execution time	Number of nodes
100	221

(a) Total sequential execution cost and graph size.

c	1000	500	100	10	1
ect	5008	2508	508	58	16

(b) Parallel computation time (using TDS with unlimited number of processors), i.e. the ect value of the exit node of the DAG, for different values of node-to-node communication cost c.

Figure 9.2. Results for the TDS algorithm on the `PreLoad` example with varying communication cost.

9.3.2 Results From using the FTD Method

Figure 9.4 gives some computed theoretical speedup figures using the Full Task Duplication Method for a discretized thermofluid pipe. Figure 9.6 contains the

Total sequential execution time	Number of nodes
8369	6301

(a) Graph size and total sequential execution cost.

c	1000	500	100	10	1
ect	9401	6901	4901	4451	4406

(b) Parallel computation time(using TDS with unlimited number of processors), i.e. the ect value of the exit node of the DAG, for different values of node-to-node communication cost c.

Figure 9.3. Results of the TDS algorithm on the robot example, using mixed mode and inline integration with varying communication cost.

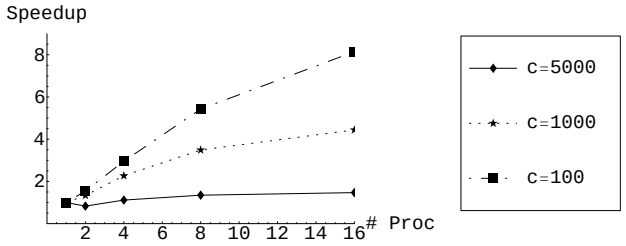
same measurements for the robot example. Since the FTD method does not involve any communication at all during the time between the computation of the states, the parallel time can easily be calculated using Equation 5.1. This equation is used to calculate the cost for the FTD method for various values of bandwidth and latency. However, since the latency cost is the most dominant one, we simplify the two values into a single communication overhead, c, with varying values. This simplification also makes the FTD method correspond better to the delay model.

The parallel simulation code from the discretized thermofluid pipe has also been executed on a PC-cluster with a SCI network as the communication device. Figure 9.5 gives the measured speedup when executing on the PC-cluster. The measurements on execution time differ from the computed theoretical speedup figures given in Figure 9.4 in several ways.

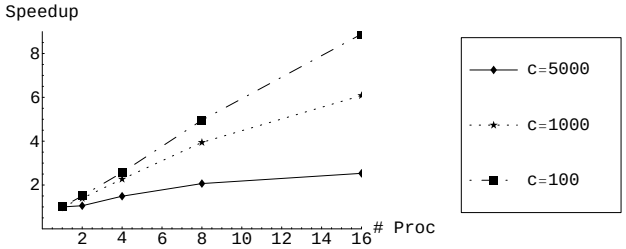
First, the achieved speedup values are lower in all three cases, compared with the most expensive communication cost used in the computed theoretical case ($c = 1000$). Thus, the actual cost of communicating is higher than 1000. The fact that the cost has been simplified from two parameters, i.e. the bandwidth and latency, to one combined parameter also affects the results.

Second, all curves have a tendency of degraded speedup as the number of processors increase. The figures shows a degradation after about 8 processors. This effect is due to the parallel communication and computation model used in this work, the delay model described in Section 2.2.4. The delay model does not cover all costs of communication, e.g. the gap cost (see Section 2.2.2) is not taken into consideration. Therefore, when the number of processors increase the master processor must spend more time communicating messages to slave processors, thus reducing the speedup.

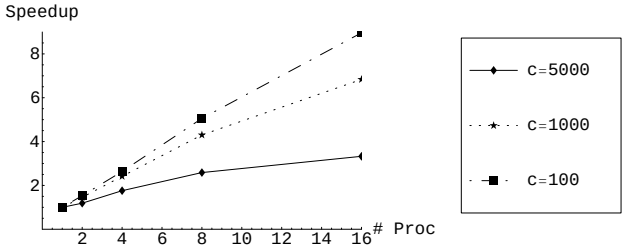
The FTD method has also been tried on the robot example, both using mixed mode and inline integration and without, see Figure 9.6. When using



(a) Thermofluid pipe with 50 discretization points.



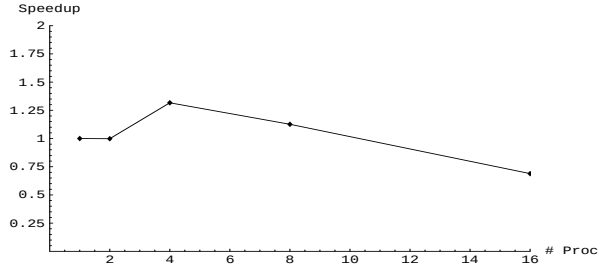
(b) Thermofluid pipe with 100 discretization points.



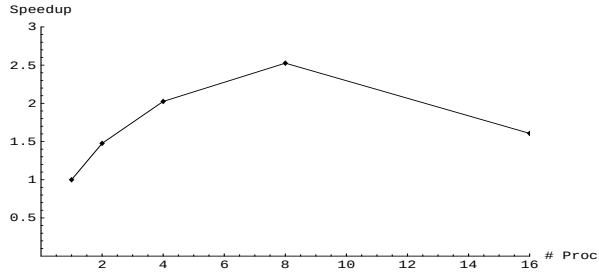
(c) Thermofluid pipe with 150 discretization points.

Figure 9.4. Computed speedup figures for different communication costs c using the FTD method on the Thermofluid pipe model.

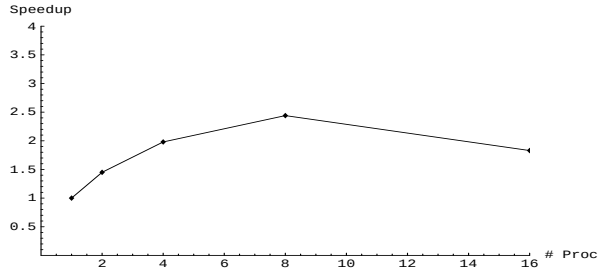
mixed mode and inline integration, the amount of parallelism clearly increases, since the robot example only gives a two processor assignment when not using mixed mode and inline integration, compared to up to nine processors when using these optimization techniques. However, the speedups in both cases are



(a) Thermofluid pipe with 50 discretization points.



(b) Thermofluid pipe with 100 discretization points.

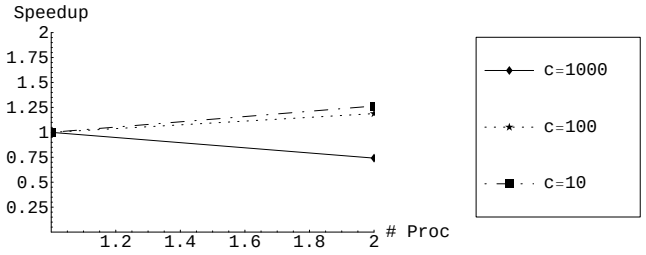


(c) Thermofluid pipe with 150 discretization points.

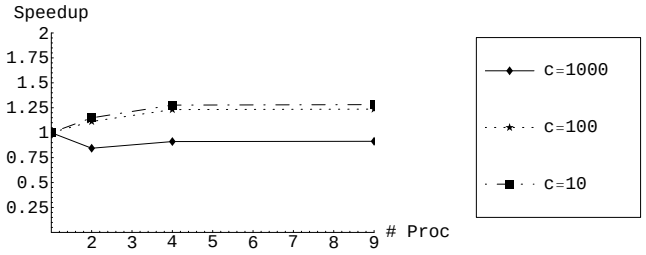
Figure 9.5. Measured speedup figures when executing on a PC-cluster with SCI network interface using the FTD method on the Thermofluid pipe model.

almost none.

Since the robot example is the most realistic example among the examples studied in this thesis, it substantially influences the interpretation of the results. Therefore, a preliminary conclusion that can be drawn is that the FTD method works well for some nice structured examples such as discretized flow



(a) Mechanical robot model with a standard solver



(b) Mechanical robot model with mixed mode and inline integration

Figure 9.6. Computed speedup figures for different communication costs, c, using the FTD method on the robot example.

models but is less suitable for general large and complex models. However, some uncertainty still remains since larger models than the robot example have not been tried yet.

9.4 ModPar Experiments

As the ModPar tool is part of the OpenModelica environment its use depends on the compilation capabilities of the OpenModelica front-end. Currently this limits the number of models that can be translated, since the OpenModelica compiler can not completely translate all language features present in e.g. the Modelica standard library.

However, the applications presented in Chapter 8 can be translated and of these, the `FlexibleShaft` example has been run on a Linux cluster and on an SGI shared memory machine. The Linux cluster, called `monolith`, is 200 PC's

each having two Intel Xeon processors at 2.2 GHz and 2GB primary memory. It has about 4.5μs latency and about 260 MB/s in bandwidth.

The shared memory machine is a Linux-based SGI Altix 3700 Bx2 supercomputer called `mozart` with a 512 GB shared memory. It has Intel Itanium 2 processors running at 1.6 GHz. Its latency of communication through the shared memory is below 1μs and its bandwidth is 6.4 GB/s.

Figure 9.7 shows the speedup for the simulation of the FlexibleShaft model on a Linux cluster (named monolith) using a fast SCI network for communication between processors. For such examples a typical speedup of around 2 can be achieved. The figure shows two different problem sizes, 100 shaft elements and 150 shaft elements. For the latter a slightly more speedup is gained, and probably for even larger problem sizes more speedup can be achieved.

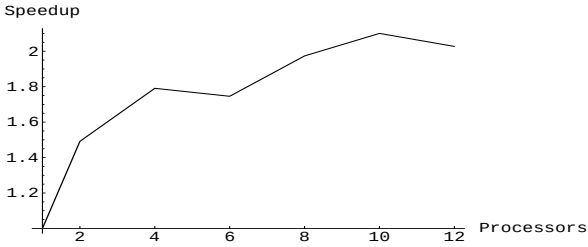
Figure 9.8 shows the speedup for the simulation of the same FlexibleShaft model on an 64 processors SGI machine (named mozart) with shared memory. This parallel machine has a bandwidth of 6.4GB/s and a latency of less than 1μs, which is substantially better compared to the monolith machine. The increased performance of the communication network is reflected in the increasing speedup, resulting in the highest speedup of about 4.8 for 16 processors, see Figure 9.8(b).

9.5 Summary

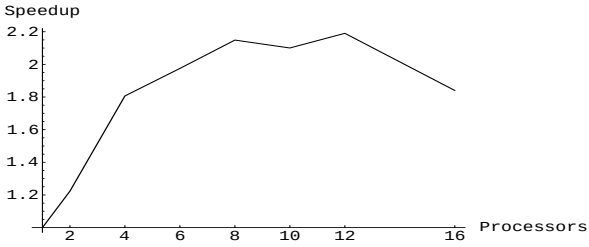
The results from this thesis work are mainly in three parts. The first part is results from the task merging method. It shows that task merging using graph rewrite rules can successfully increase the granularity of fine grained task graphs such that they can be more suitable for scheduling using many existing scheduling algorithms.

The second part of the research results is on the DSBPart parallelization tool which translates simulation code from the Dymola tool to parallel C-code. It shows speedup figures from some models and that other models does not produce speedup due to limited amount of parallelism.

The final part shows results on the ModPar parallelization tool, which is part of the OpenModelica compiler framework. This tool uses the task merging algorithm and shows that speedup is also possible using this approach. For the examples presented here, similar results as for the DSBPart tool are achieved.

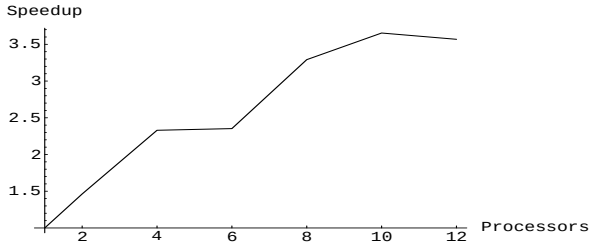


(a) Flexible shaft with 100 elements on monolith

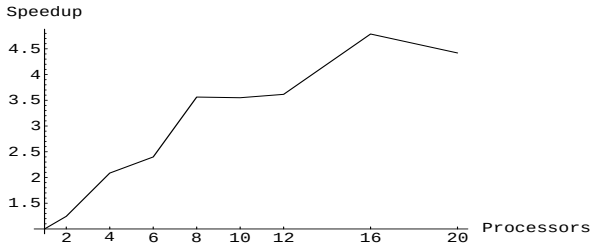


(b) Flexible shaft with 150 elements on monolith

Figure 9.7. Measured speedup when executing the simulation of the Flexibleshaft model on the monolith PC-cluster with SCI network interface.



(a) Flexible shaft with 100 elements on mozart



(b) Flexible shaft with 150 elements on mozart

Figure 9.8. Measured speedup when executing the simulation of the Flexiblesshaft model on the mozart 64 processors shared memory SGI machine.

Chapter 10

Related Work

This chapter relates the work presented in this thesis to the work of other people, both regarding scheduling and clustering of task graphs as well as regarding automatic parallelization of simulation code. Related work on clustering and scheduling algorithms have been discussed earlier in Chapter 2. However, in this chapter we discuss from how these algorithms relate to the work in this thesis.

10.1 Parallel Simulation

Related work in parallel simulation is divided into the three parallelization approaches presented in Chapter 1: parallelism over the method, parallelism over time, and parallelism over the system. Parallelism over the method covers parallel solving techniques for differential equations, while parallelism over time covers the work on parallel discrete event simulation techniques. Parallelism over the system is the approach taken in this thesis.

10.1.1 Parallel Solvers

Parallel solving techniques tries to extract parallelism from the solver approximation scheme, typically using multi-step solvers. These solvers calculates several instances of the solved equations, the function f, see Equation 3.5 on page 60, in parallel. One such solver is the parallel Runge-Kutta solver [68]. This technique could be combined with the work presented in this thesis to reveal even more parallelism. The parallelization would in that case be done on two different levels. First, a coarse grained parallelization of different computations of f and also a more fine grained parallelization of each computation

of f itself.

Using parallel solvers like Runge-Kutta can be beneficial for some examples. It is however not a generic solution to the automatic parallelization problem of simulation code since many simulation models require more complex solvers with better error estimates and better stability. Most *stiff* problems, i.e. problems with both fast and slow transients, requires advanced solvers like for instance DASSL. These kind of problems are note possible to simulate using simple techniques like the Runge-Kutta methods.

A special kind of parallel solving technique is to use data parallelism over the equations and variables. This is typically used for PDE problems and ODE problems on explicit form. For instance, the famous CVODE solver for solving ODE problems also exist in a parallel version which uses data parallelism as parallelization scheme [11].

Another parallel solver is the Concurrent DASSL [38]. It is a parallel implementation of the DASSL solver.

10.1.2 Discrete Event Simulations

When the simulation model is limited to using only discrete events without any continuous variables, a discrete event simulation solver can be used. Such simulations do not require the solution technique to sequentially increase the time variable a step at a time. Instead they adopt different parallelization schemes to jump forward (and backward) in time to better exploit parallelism.

The parallelization approaches for discrete event system simulations are also not generic enough for systems simulation where continuous variables are needed. There are however application areas like traffic simulations, where a completely discrete event formulation of the problem is sufficient. For such cases, discrete event simulation and the parallelization schemes for these can be successfully used.

10.1.3 Parallelism Over Equations in the System

The approach taken in this thesis to perform parallelism over the system means that the calculation of the equations are parallelized. This approach has been adopted earlier in for instance real time simulations using the Transputer parallel computer [84]. The Transputer parallel computer is quite old and has a relatively low communication to computation time ratio compared to todays parallel computers like Linux cluster computers. Therefore the parallelization work performed on the Transputer had a much more coarse grained task graph as starting point for the scheduling and clustering algorithms, thus giving good speedup results.

There are also hand written simulation codes, e.g. these presented in next section, that use parallelism over the system. For instance, the BEAST simulation tool calculates the equations, mostly regarding contact calculations between bearings and roller elements in roller bearing simulations in parallel.

10.1.4 Parallel Simulation Applications

The perhaps most common parallel simulations are hand written simulators for specific application areas. The reason for this being the most popular way of exploiting parallel computing in simulations are several.

One reason is that general simulation tools are too generic for special purpose simulations with large computation costs and such tools can not efficiently parallelize such simulation code. The simulation tools might not even have the strength to be able to specify the complex equations and dedicated solver techniques for such applications.

Another reason for handwritten simulator is that the parallelization scheme itself can involve approximations, e.g. to reveal more parallelism in comparison to the sequential execution code of such simulation. For instance, an implementation of a parallel particle simulation can divide the particle space into regions, with one region per processor, and neglect collisions and other interference between particles of different regions. This approximation reveals more parallelism and substantially improves the performance of the parallel simulation of particles. Such application specific knowledge can not be included in a generic automatic parallelization tool.

Examples of simulation code that is handwritten (typically in Fortran) are:

- Weather forecast simulations at the Swedish Meteorological Institute (SMHI). These models build on advanced partial differential equations written in Fortran.
- Photon particle simulations in supernovas.
- Biological and chemical large scale simulations.
- High performance roller bearing simulations in the BEAST tool [52]. These are written in C++, using object oriented techniques and C++ to model roller bearings.

10.2 Scheduling

There are scheduling algorithms that takes a task graph and schedules it for a fixed number of processors directly, without the need of clustering (or task

merging). The Modified Critical Path (MCP) scheduling algorithm is an example of such a scheduling algorithm. It takes a task graph with execution costs and communication costs and the number of processors for the schedule as input. Hence, it uses the delay model for task graph scheduling. The computational complexity of the MCP algorithm is $O(n^2(log(n) + P))$ for n tasks and P processors.

Another such algorithm is the ERT algorithm presented in detail in Chapter 2. It has similar performance as the MCP algorithm [35].

One problem with scheduling algorithms solving the entire scheduling problem, including mapping to a fixed number of processors, is that the computational complexity of the problem is too high. These algorithms tend to have a high complexity but still using e.g. a simple cost model and not using task duplication. Therefore they also have problems with scheduling fine grained task graph where much duplication is a necessity. On the other hand, by solving the complete problem in a single step, it is more probable to come up with optimal schedules. But since the multiprocessor scheduling problem is proved to be NP-complete, optimal algorithms are not practical.

10.3 Clustering and Merging

Much work has been done in the area of clustering of task graphs, and some work has also been done in the approach of merging tasks in a task graph to increase the granularity. The following sections present work related to this thesis and explain the differences and similarities.

10.3.1 Task Clustering

The work of forming clusters of tasks in a task graph has been studied in depth. Early approaches include the internalization algorithm [81] and the DSC algorithm [88], where the later can be seen as an improvement of the former. The DSC algorithm is presented in detail in Chapter 2. These two algorithms both use the parallel time, see Equation 2.16, as a metric for guiding the choice of inclusion into a cluster or starting building a new cluster. The same approach has been taken in this work, by trying to reduce the top level (see Equation 2.3 on page 26) of tasks in the task graph which will result in a lower parallel time of the task graph. These two clustering techniques do not use task replication to reduce communication, as is done in this work. Also, since they are pure clustering algorithms they do not agglomerate (or merge) communication messages between tasks. The reason being that the simple delay model does not give a benefit of such agglomeration.

There has also been efforts of clustering algorithms that use task replication to improve scheduling performance. Studies has shown that using task replication is beneficial [41, 58], hence many recent clustering algorithms use it.

One task duplication based clustering algorithm is the DFRN algorithm [59]. It uses task duplication at a lower complexity compared to earlier algorithms having task duplication. But still, the DFRN algorithm has the complexity $O(n^3)$, which is quite high for large and fine grained task graphs.

All these algorithms use the delay model, presented in Chapter 1, whereas this work use parts of the LogP model which is a more accurate cost model for task graphs. There are however also clustering and scheduling approaches emerging for the LogP cost model, like for instance [42, 9].

Others use simpler cost models which still are more accurate than the delay model [78, 37]. These cost models are very similar to the approach taken in this work, having a latency and bandwidth parameter in the cost model.

To conclude on the related work on both scheduling algorithms and clustering algorithms it should be noted that the task merging approach can be considered as a complement to these methods. There are no obstacles to using both task merging and task clustering followed by task scheduling to solve the multiprocessor scheduling of task graphs. However, the task merging algorithm takes over parts of the clustering work so a clustering phase is not as crucial if task merging is performed.

10.3.2 Task Merging

To merge tasks with the intent of increasing the granularity of the task graph has been used in several approaches. One approach is called grain packing [31], which is essentially the same as merging tasks.

However, as far as we know there are no task merging approaches besides our that uses task replication.

10.4 Summary

Work related to this thesis can be found in parallel simulation techniques including automatic parallelization, parallel numerical solving techniques and to some extent parallel simulation applications. Another area of related work deals with task clustering and scheduling techniques in general. Clustering algorithms are similar to our work of task merging in many ways, but many of them are using the simpler delay model as a cost model for task graphs. There are also other task merging techniques such as grain packing that are closely related to our work but does not use task replication.

To summarize our work distinguishes itself from the related work in that we are using a task merging technique where tasks and communication messages are merged together to increase granularity and also using task replication to obtain better results. This combination has not been investigated before. Our work is also unique in that we use the graph rewrite system formulation of our problem, which to our knowledge has not been done previously in task clustering or task merging.

Chapter 11

Future Work

In this chapter we present some future directions of further work on the research problem in this thesis.

11.1 The Parallelization Tool

Currently, a large a subset of the Modelica language is supported in the OpenModelica framework. There is still future work on the implementation of Modelica language constructs and of further equation optimizations in the compiler. Moreover, the ModPar module needs support for these missing features.

One important part of the Modelica language that has not been considered at all in this thesis work is the handling of discrete events, i.e. the hybrid parts of the Modelica language. When a discrete event happens, e.g. when a state variable climbs above a certain threshold, the exact point in time for this discrete event must be calculated. This means that the numerical solution process temporarily stops at that point. The event calculation is performed (the body of the `when` statement) and the numerical solution process is restarted. This method of handling hybrid systems complicates the parallelization work. When an event occurs all processors must be interrupted (or just a few of them, depending on using a distributed or centralized solver) and the solver stopped. The event must be handled by one (or several) processors, and thereafter must the previous work scheme be taken up again. The parallelization of hybrid systems has been put out of the scope of this thesis because it involves too much additional work.

However, purely discrete models in Modelica could be parallelized using existing parallel discrete event simulation method. The easiest way of performing such parallelizations is probably to try to automatically export such Modelica

models into existing discrete-event simulation tools. Thus, future work here includes finding a mapping (translation scheme) for such export functionality.

11.2 Task Merging

The task merging approach (ATMM) using Graph Rewrite Systems can be further improved in several directions. For instance the cost model of the communication can be fully extended to the LogP model, taking also the gap parameter g into consideration. It is also possible to consider the P parameter in the task merging, i.e. the number of available processors. But in this case, the task merging algorithm is also starting to take scheduling decisions, i.e. making it less flexible and more complex. As the task merging algorithm is presented in this thesis, it can be used with the same parameters for schedules on an arbitrary number of processors. For instance, the code generator could use the same merged task graph for scheduling the graph on 2, 4 8 and 16 processors at the same time, depending on available resources. But by considering the P parameter the rewrite rules could for instance be limited in the number of predecessors of tasks in the task graph, to limit the amount of parallelism to a suitable level for a fixed number of processors.

Another approach of improving the graph rewrite system is to further allow malleable tasks of the task graph. We have presented one approach of how malleable tasks can be introduced into the rewrite system, but by allowing more scheduling decisions in the algorithm, the introduction of malleable tasks can be made even more efficient. For instance, by deciding that a malleable task should execute on a certain number of processors, the execution time of the malleable task can be determined more accurately. This might potentially allow for more reductions of the task graph on other places, resulting in an increased granularity and therefore a better schedule in the end.

11.3 The Modelica Language

The Modelica language itself is evolving according to user feedback and user requirements in the Modelica Design Group. However, several extensions to Modelica in different directions are investigated at the Programming Environments Laboratory at Linköping University. These extensions include extending Modelica with support for Partial Differential Equations. Such extension would require new parallelization schemes including data parallelism. For instance, a Modelica model of a large system might then include a PDE problem as a component. Such models might use a specific (parallel) PDE solver for that component and other solvers for other components of the model. Future

work includes how to parallelize such models and also how to choose which solvers should be used.

Another extension to the Modelica language could be the introduction of TLM (Transmission Line Modeling) in Modelica []. This is a method of identifying parts of a model that have slow moving transients and can thereby be modeled as a transmission line having a certain delay. In that way, the equation system is broken apart at this transmission line and the two parts at each end of the line can be solved using their own numerical solver. Thus, the partitioning of the model onto the parallel machine is partly performed by the user by adding TLM components to the model. In this way, a large and complex simulation model can be partitioned manually and it is thereby easier to achieve good speedups.

11.4 Summary

The future work of the research in this thesis lies in three different areas. The automatic parallelization tool is only on the prototype stage and there is much implementation left to handle the complete Modelica language. The task merging algorithm can be further improved by a more accurate cost model and by improving the transformation rules. Finally the Modelica language and the research activities surrounding it, like partial differential equations and transmission line modeling, can pose new requirements on automatic parallelization, e.g. to investigate automatic solver selection techniques and how to fully exploit TLM modeling in the parallelization.

Chapter 12

Contributions and Conclusions

This chapter presents contributions and conclusions. Conclusions regarding the hypothesis of the thesis work are presented and of the task merging technique using graph rewrite systems.

12.1 Contributions

The contributions of this thesis are:

- ATMM, a new approach of merging tasks using a graph rewrite system formulation with a small set of transformation rules designed to merge tasks together to increase granularity without decreasing the amount of parallelism of the task graph. To our knowledge, the approach is the first known work on task merging using Graph Rewrite Systems and also adopting task replication.
- An automatic parallelization tool, ModPar, for the equation-based modeling language Modelica, that is shown with examples to give speedups when executing the simulations of these examples on parallel computers. The ModPar tool is based on the Open Source implementation OpenModelica, including task merging, task scheduling and code generation techniques adopted for automatic parallelization of simulation code. This tool was also partly based on the earlier DSBPart prototype tool.
- A task graph implementation in Mathematica providing an interactive environment with emphasis on experimentation and evaluation of schedul-

ing and clustering algorithms. The implementation includes a number of implemented scheduling and clustering algorithms and the task merging approaches presented in this thesis.

- Insight and conclusions of working with a research compiler for natural semantics by using the RML language. The author has spent much of the time implementing the Modelica compiler written in RML, an experience discussed in the conclusions below.
- The author has also taken an active role in the Design of the Modelica language, attending design meetings on many occasions, contributing to the improvement of the Modelica modeling language.

12.2 Conclusions

The conclusions of this thesis, automatic parallelization and implementation, are presented in the two following sections.

12.2.1 Automatic parallelization

We have presented an automatic parallelization framework for Modelica based on the OpenModelica compiler. A prototype implementation has been made that can successfully parallelize Modelica simulation models. The speedup of these simulations vary depending on the size and complexity of the simulations.

In this thesis we have proposed a new method (ATMM) of merging tasks using a generic method for transforming graphs called graph rewrite systems. This approach is based on the LogP cost model of parallel computing for the communication costs in a task graph. It also uses task replication to decrease the communication cost and improve performance.

The task merging method is based on a small set of transformation rules which are quite simple and easy to understand, but still have been demonstrated to be very efficient at increasing the granularity of fine grained task graphs. This has been one of the obstacles in the automatic parallelization of simulation code, since in this case the task graphs have been built on the finest granularity level possible. The reason being to detect all possible parallelism of the simulation code. Hence, the combination of building the task graph at this fine grained level, combined with the task merging method has resulted in a successful parallelization technique suitable for parallelization of many simulation models.

The termination of the graph rewrite system is guaranteed since the task merging system will never add tasks to the task graph. It will only merge tasks, reducing the total number of tasks in the graph. Thus, the task merging

system will always terminate and the computational complexity of the system can therefore be (theoretically) determined.

The computational complexity of the task merging algorithm is $O(n^2)$ for a fine grained task graph in the case of a total reduction/merging of the tasks of the task graph. Since this is seldom the case, the computational complexity is in practice lower than this boundary. In practice it very much depends on the granularity of the task graph and on the bandwidth and latency parameters. Many reductions will be made for high values of latency and fine grained task graphs, resulting in higher computation time, whereas for coarse grained task graphs, no or very few reductions will be performed. For such cases, the computational complexity is $O(n)$, i.e., linear in the number of tasks of the task graph.

The task merging system has been developed in two different versions, where the first attempt did not sufficiently succeed in reducing the task graphs. Therefore, a second more elaborated set of transformation rules was developed. Unfortunately, the enhancements also led to confluence problems regarding the rewrite system. Thus, the enhanced system needed a priority mechanism for the transformation rules such that the system was made confluent.

The task merging approach has been experimentally tested both on application examples from simulation code and by using the standard task graph set. The task merging approach succeeded in increasing granularity in most of the cases.

12.3 Implementation

Most of the implementation work in this research problem has been done using the RML language, presented in Chapter 6.3.2. Since RML is a research language developed at PELAB, Linköping university, it does not have a full-fledged development environment as can be found for e.g. Java, C++ or other wide-spread programming languages. It does not have good enough support for debugging, profiling, etc, and it does not have a large standard library as have most languages today (e.g. Java or C++). It does not for instance have support for the most common used data types such as associative arrays (maps or hash tables), reusable vectors, and many other data types commonly used in compilers. Some of these missing features, e.g. library support for hash tables, dictionaries, binary trees, etc., has during this work been implemented by this author and a colleague. However, some work still remains to convert this into a general reusable library.

However, among the advantages of RML are, quick to learn, powerful pattern matching, good support for abstract syntax trees, automatic memory management, and compilation to efficient C-code. Since RML uses Natural

Semantics it is also efficient at describing typical semantic rules in a compiler. However, for other parts of the compiler it is sometimes less suitable.

Today, the compiler is about 72000 lines of RML code, which can take quite some time to fully grasp. However, PELAB has initiated an effort to automatically translate this RML code into a MetaModelica language, which is an extension of the Modelica language. By performing this transformation, we will get a Modelica compiler written in MetaModelica, a superset of Modelica, i.e., the compiler is in principle written in itself. Through this effort, the OpenModelica compiler will have removed most of the drawbacks of RML but still retain the advantages. Also, the increasing number of Modelica users around the world will probably increase the standard library of both Modelica functions, Modelica data types such as associative arrays, etc., and Modelica models.

A final observation is that implementing compilers is a hard task which requires both experienced programmers *and* good tool support. The RML compiler did not fulfill these requirements fully, and if the OpenModelica compiler were to be written today it would probably not be done using RML, given the lack of tool support in compiler and development environment. However, with the automatic translation to MetaModelica along with better tool support in an integrated environment (eclipse [20]) with both debugging, browsing, etc., the OpenModelica compiler will be a success.

12.4 Summary

The main contribution of this thesis is the ATMM task merging method, and an automatic parallelization tool that enables engineers with little or no knowledge of parallel programming to speed up their simulations using parallel computing. The contribution consists of algorithms and techniques for automatic parallelization prototype tools and experimental frameworks for conducting further research in this area.

Bibliography

[1] Anderson, E. and Bai, Z. and Bischof, C. and Demmel, J. and Dongarra, J. and Du Croz, J. and Greenbaum, A. and Hammarling, S. and McKenney, A. and Ostrouchov, S. and Sorensen, D. *LAPACK Users' Guide, Release 1.0*. SIAM, Philadelphia, 1992.

[2] N. Andersson. Licentiate thesis: Compilation of Mathematical Models to Parallel Code. Department of Computer and Information Science, Linköpings Universitet, Sweden, 1996.

[3] P. Aronsson. Licentiate thesis: *Automatic Parallelization of Simulation Code from Equation Based Simulation Languages*. Department of Computer and Information Science, Linköpings universitet, Sweden, 2002.

[4] P. Aronsson and P. Fritzson. Multiprocessor Scheduling of Simulation Code from Modelica Models. In *Proceedings of the 2nd International Modelica Conference*. DLR, Oberpfaffenhofen, Germany, March 2002.

[5] Peter Aronsson and Peter Fritzson. Task merging and replication using graph rewriting. In *Tenth International Workshop on Compilers for Parallel Computers*. Amsterdam,the Netherlands, Jan 8-10 2003.

[6] Peter Aronsson, Peter Fritzson, Levon Saldamli, Peter Bunus, and Kaj Nyström. Meta programming and Function Overloading in OpenModelica. In *Proceedings of the 3rd International Modelica Conference*. Linköping, Sweden, November 3-4, 2003.

[7] M. Ayed, J-L Gaudiot. An efficient heuristic for code partitioning. *Parallel Computing*, 26:399–426, 2000.

[8] BOOST graph library, http://www.boost.org accessed 2006-01-05.

[9] C. Boeres and E.F. Rebello. Cluster-based static scheduling: Theory and practice. In *Proceedings of the 14th Symposium on Computer Architecture and High Performance Computing*, 2002.

[10] P. Bunus. *Debugging Techniques for Equation-Based Languages*. PhD thesis, Linköping Studies in Science and Technology. Dissertation Dept. of Computer and Information Science, 2004.

[11] George D. Byrne and Alan C. Hindmarsh. Pvode, an ode solver for parallel computers. *International Journal of High Performance Computing Applications*, 13(4):354–365, 1999.

[12] Thomas L. Casavant and Jon G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *Software Engineering*, 14(2):141–154, 1988.

[13] E. Coffman Jr. and R. Graham. Optimal scheduling for two processor systems. *Acta Informatica*, vol. 1(no. 3):200–213, 1972.

[14] A Catalogue of research compilers, http://compiler-tools.org, accessed 2006-01-05.

[15] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.

[16] S. Pande S. Darbha. A Robust Compile Time Method for Scheduling Task Parallelism on Distributed Memory Machines. In *Proceedings of PACT'96*, pages 156–162, 1996.

[17] S. Darbha, D. P. Agrawal. Optimal Scheduling Algorithm for Distributed-Memory Machines. *IEEE Transactions on Parallel and Distributed Systems*, vol. 9(no. 1):87–94, January 1998.

[18] C. Donnelly and R. Stallman. *Bison: the YACC-compatible Parser Generator, Bison Version 1.28*. Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA, Tel: (617) 876-3296, USA, 1999.

[19] *Dymola, http://www.dynasim.se*.

[20] The eclipse project, http://www.eclipse.org.

[21] H. Elmqvist, M. Otter, and F. Cellier. Inline integration: A new mixed symbolic /numeric approach for solving differential– algebraic equation systems, 1995.

[22] H. Elmqvist, M. Otter. Methods for Tearing Systems of Equations in Object-Oriented Modeling. In *Proceedings ESM'94 European Simulation Multi-conference, Barcelona, Spain*, June 1994.

[23] Vadim Engelson. *Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing.* PhD thesis, Dept. of Computer and Information Science, Linköping University, 2000.

[24] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[25] I. Foster. *Designing and Building Parallel Programs.* Addison-Wesley, 1995.

[26] D. Fritzson and P. Nordling. Adaptive scheduling strategy optimizer for parallel roller bearing simulation. *Future Generation Computer Systems*, 16:563–570, 2000.

[27] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1.* IEEE Press, 2004.

[28] C. Hanen, A. Munier. An approximation algorithm for scheduling dependent tasks on m processors with small communication delays. Technical report, Laboratoire Informatique Theorique Et Programmation, Institut Blaise Pascal, Universite P.et M. Curie, 1999.

[29] I.S Duff, A.M Erisman and J.K Reid". *Direct Methods for Sparse Matrices.* Oxford Science Publications, 1989.

[30] Gilles Kahn. Natural semantics. In Springer-Verlag, editor, *In Proc. of the Symposium on Theoretical Aspects on Computer Science (STACS'87)*, pages 22–39, 1987.

[31] B. Kruatrachue and T. Lewis. Grain Size Determination for Parallel Processing. *IEEE Software*, vol. 5(no. 1):23–32, 1988.

[32] B. Kruatrachue. *Static Task Scheduling and Grain Packing in Parallel Processor Systems.* PhD thesis, Dept. of Electrical and Computer Engineering, Oregon State University, 1987.

[33] Kurt Gieck and Reiner Gieck. *Engineering Formulas.* McGraw-Hill, 7th edition edition, 1997.

[34] Y-K. Kwok, I. Ahmad. Exploiting Duplication to Minimize the Execution Times of Parallel Programs on Message-Passing Systems. *Transactions on Parallel and Distributed Systems*, vol. 9(no. 9), 1998.

[35] Y.-K. Kwok and I. Ahmad. Benchmarking the task graph scheduling algorithms. In *Proceedings IPPS/SPDP*, pages p. 531–537, 1998.

[36] C.Y. Lee, J.J. Hwang, Y.C. Chow, F.D Anger. Multiprocessor Scheduling with Interprocessor Communication Delays. *Operations Research Letters*, vol.7(no. 3), 1988.

[37] "H. Lee, J. Kim, S.-J. Hong, and S. Lee". Task scheduling using a block dependency DAG for block-oriented sparse cholesky factorization. In *SAC (2)*, pages 641–648, 2000.

[38] A. Leung, A. Skjellum, and G. Fox. Concurrent dassl: a second-generation dae solver library. In *Proceedings of the Scalable Parallel Libraries Conference*, 1993.

[39] J.C. Liou, M. Palis. CASS: An Efficient Task Management System for Distributed Memory Architectures. In *Parallel Architectures, Algorithms and Networks, Proceedings*. IEEE Computer, 1997. December 18-20, Taipei, Taiwan.

[40] Zhen Liu. Worst-case analysis of scheduling heuristics of parallel systems. *Parallel Computing*, 24:863–891, 1998.

[41] E. Luque, A. Ripoll, P. Hernandez, T. Margalef. Impact of Task Duplication on Static-Scheduling Performance in Multiprocessor Systems with Variable Execution-Time Tasks. In *International Conference on Super-Computing*. ACM Press, 1990. Amsterdam, Netherlands.

[42] W. Löwe and W. Zimmermann. Scheduling balanced task-graphs to logp-machines. *Parallel Computing*, 26:1083–1108, 2000.

[43] *MathModelica, http://www.mathcore.com.*

[44] Matlab simulink. http://www.mathworks.com.

[45] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[46] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[47] A Migdalas, P. M. Pardalos, and S Storoy. *Parallel Computing in Optimization*. Kluwer Academic Press, 1997.

[48] *The Modelica Language, http://www.modelica.org.*

[49] Modelica Association. *The Modelica Language Specification Version 2.0*, Mars 2002. http://www.modelica.org.

[50] The *monolith* computer cluster at nsc (national computer center, sweden). http:/www.nsc.liu.se/systems/monolith.

[51] Myrinet, http://www.myrinet.com.

[52] P. Nordling and P. Fritzson. Parallelization of the CVODE Ordinary Differential Equation Solver with Applications to Rolling Bearing Simulation. In Bob Hertzberg and Giuseppe Serazzi, editors, *High Performance Computing and Networking LNCS 919, Springer-Verlag*, 1995.

[53] National Supercomputer Centre, Sweden. http://www.nsc.liu.se.

[54] OpenMP Forumn Home Page. http://www.openmp.org/.

[55] P. Aronsson and P. Fritzson and P. Bunus and L. Saldamli. Incremental Declaration Handling in Open Source Modelica. In *In Proceedings, SIMS - 43rd Conference on Simulation and Modeling*, September 2003.

[56] M. A. Palis, J-C- Liou and D. Wei. Task Clustering and Scheduling for Distributed Memory Parallel Architectures. *Transactions on Parallel and Distributed Systems*, vol. 7(no. 1), 1996.

[57] C. C. Pantilides. The consistent initialization of differential-algebraic systems. *Siam Journal of Scientific Computing*, 9(2), 1988.

[58] G.L Park, B. Shirazi, J. Marquis. Comparative Study of Static Scheduling with Task Duplication for Distributed Systems. *Solving Irregularly Structured Problems in Parallel Computing*, 1997.

[59] G.L. Park, B. Shirazi, J. Marquis. DFRN: A New Approach for Duplication Based Scheduling for Distributed Memory Multiprocessor Systems. In *Proceedings of Parallel Processing Symposium*, 1997.

[60] C. S. Park, S. B. Choi. Multiprocessor Scheduling Algorithm Utilizing Linear Clustering of Directed Acyclic Graphs. In *Parallel and Distributed Systems, Proceedings*, 1997.

[61] V. Paxson. GNU Flex Manual, Version 2.5.3, Free Software Foundation, 1996.

[62] Sriram Pemmaraju and Steven Skiena. *Computational Discrete Mathematics : Combinatorics and Graph Theory with Mathematica*. Cambridge University Press, 2003.

[63] Mikael Pettersson. *Compiling Natural Semantics*. PhD thesis, Linköping Studies in Science and Technology, Dissertation
Dept. of Computer and Information Science, 1995.

[64] Pvm - parallel virtual machine. http://www.csm.ornl.gov/pvm/pvm_home.html.

[65] Andrei Radulescu, A. J.C. van Gemund. FLB:Fast Load Balancing for Distributed-Memory Machines. Technical report, Faculty of Information Technology and Systems, Delft University of Technology, 1999.

[66] A. Radulescu and A. van Gemund. On the complexity of list scheduling algorithms for distributed-memory systems. In *ACM International Conference on Supercomputing*, 1999.

[67] Rauber, T. and Runger, G. Iterated Runge-Kutta Methods on Distributed Memory Multiprocessors. In *Parallel and Distributed Processing, Proceedings*, pages 12–19. First Aizu International Symposium on, 1995.

[68] T. Rauber and G. Runger. Parallel iterated runge– kutta methods and applications. *International Journal of Supercomputer Applications*, 10(1):62–90, 1996.

[69] Levon Saldamli. *PDEModelica - A High-Level Language for Modeling with Partial Differential Equations*. PhD thesis, Department of Computer and Information Science, Linköping University,, May 2006.

[70] Scali - Scalable Linux Systems, http://www.scali.com.

[71] Sca mpi. http://www.scali.com/.

[72] A. Schiela, H. Olsson. Mixed-mode Integration for Real-time Simulation. In P. Fritzson, editor, *Proceedings of Modelica (2000) Workshop, http://www.modelica.org*, pages 69–75, 2000.

[73] Sci network from dolphin. http://www.dolphinics.com/.

[74] G. Sih and E. Lee. A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures. *IEEE Transactions on Parallel and Distributed Systems*, vol. 4(no. 2), 1993.

[75] Standard ml, http://www.standardml.org.

[76] SO/IEC Standard 9945-1:1996. Information Technology-Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) [C Language]. Technical report, The Institute of Electrical and Electronics Engineers, 1996.

[77] Standard Task Graph Set, http://www.kasara.elec.waseda.ac.jp/schedule accessed 2006-01-04.

[78] H. Topcuoglu and M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13, 2002.

[79] H. Tummescheit. *Design and Implementation of Object-Oriented Model Libraries using Modelica*. PhD thesis, Dept. of Automatic Control. Dissertation. Lund University, 2002.

[80] J. D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, 1975.

[81] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, 1989.

[82] L. G- Valliant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[83] L. Viklund, J. Herber, and P. Fritzson. The Implementation of ObjectMath - a High-Level Programming Environment for Scientific Computing. In *Computational Complexity*, pages 312–318, 1992.

[84] B. E. Wells. A Hard Real-Time Static Task Allocation Methodology for Highly-Constrained Message-Passing Environments. *The International Journal of Computers and Their Applications*, II(3), December 1995.

[85] G. William and E. Lusk. User's guide for mpich, a portable implementation of mpi.

[86] Stephen Wolfram. *The Mathematica Book, Fifth Edition*. Wolfram Media, 2003.

[87] Wu, M. Y. and Gajski, D. D. Hypertool: A Programming Aid for Message-Passing Systems. *Transactions on Parallel and Distributed Systems*, vol. 1(no. 3), 1990.

[88] T. Yang, A. Gerasoulis. DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. *Transactions on Parallel and Distributed Systems*, vol. 5(no. 9), 1994.

Appendix A

Task Merging Experiments

Table A.1 below shows the task merging algorithm on a subset of the Standard Graph Set (STG) with graphs of size 100. The first column is the filename for the task graph. The second column gives the parallel time before the merge. The third column shows the parallel time after the task mering has been run. The fourth column is the number of tasks in the graph and the fifth column is the number of tasks after the task merging. All these figures are the same when having different priority order of the task merging rules.

File	*PT before*	*PT after*	*No. tasks before*	*No. tasks after*
rand0161.stg	520	351	100	100
rand0167.stg	252	233	100	100
rand0106.stg	293	230	100	100
rand0122.stg	577	429	100	99
rand0053.stg	165	126	100	100
rand0163.stg	809	490	100	99
rand0166.stg	93	49	89	78
rand0163.stg	809	490	100	99
rand0152.stg	159	110	99	93
rand0157.stg	843	576	100	100
rand0076.stg	353	329	100	97
rand0071.stg	698	401	100	100
rand0114.stg	125	86	100	95
rand0171.stg	265	247	100	100
rand0114.stg	125	86	100	95

continued on next page

continued from previous page				
File	*PT before*	*PT after*	*No. tasks before*	*No. tasks after*
rand0168.stg	242	212	100	100
rand0119.stg	242	228	100	100
rand0102.stg	191	147	100	99
rand0080.stg	185	86	97	90
rand0068.stg	818	605	100	100
rand0061.stg	519	282	100	98
rand0100.stg	151	97	100	97
rand0134.stg	376	282	100	100
rand0157.stg	843	576	100	100
rand0078.stg	802	660	100	100
rand0111.stg	84	36	88	76
rand0027.stg	423	295	100	100
rand0035.stg	154	112	100	98
rand0025.stg	227	142	99	97
rand0126.stg	573	367	100	96
rand0040.stg	193	173	100	96
rand0051.stg	159	88	100	98
rand0052.stg	143	86	100	98
rand0148.stg	179	136	100	99
rand0142.stg	156	137	98	94
rand0158.stg	459	296	100	98
rand0116.stg	235	220	100	100
rand0155.stg	310	231	100	93
rand0028.stg	362	278	100	99
rand0168.stg	242	212	100	100
rand0175.stg	212	153	100	98
rand0053.stg	165	126	100	100
rand0126.stg	573	367	100	96
rand0100.stg	151	97	100	97
rand0114.stg	125	86	100	95
rand0131.stg	227	152	100	98
rand0170.stg	255	232	100	100
rand0121.stg	257	178	100	97
rand0035.stg	154	112	100	98
rand0020.stg	370	175	100	99

Table A.1. Task Merging on STG using $B = 1$ and $L = 10$

Table A.2 below presents results for the same STG subset as above but with $B = 1$ and $L = 100$.

File	*PT before*	*PT after*	*No. tasks before*	*No. tasks after*
rand0161.stg	2860	1115	100	94
rand0167.stg	1062	907	100	100
rand0106.stg	1100	727	100	97
rand0122.stg	3187	1362	100	93
rand0053.stg	975	649	100	98
rand0163.stg	4679	1922	100	97
rand0166.stg	543	111	89	41
rand0163.stg	4679	1922	100	97
rand0152.stg	699	348	99	69
rand0157.stg	4173	1482	100	98
rand0076.stg	1603	625	100	92
rand0071.stg	4208	1662	100	93
rand0114.stg	755	322	100	87
rand0171.stg	1075	835	100	98
rand0114.stg	755	322	100	87
rand0168.stg	1052	703	100	97
rand0119.stg	1052	923	100	100
rand0102.stg	1001	652	100	97
rand0080.stg	1175	321	97	69
rand0068.stg	4224	1625	100	97
rand0061.stg	3219	1312	100	97
rand0100.stg	851	347	100	91
rand0134.stg	1906	782	100	92
rand0157.stg	4173	1482	100	98
rand0078.stg	3907	1763	100	92
rand0111.stg	534	106	88	35
rand0027.stg	2223	1007	100	89
rand0035.stg	872	358	100	92
rand0025.stg	1127	453	99	86
rand0126.stg	2733	876	100	90
rand0040.stg	993	579	100	96
rand0051.stg	969	431	100	97
rand0052.stg	863	428	100	87

continued on next page

continued from previous page				
File	*PT before*	*PT after*	*No. tasks before*	*No. tasks after*
rand0148.stg	899	447	100	94
rand0142.stg	786	364	98	75
rand0158.stg	2709	807	100	89
rand0116.stg	1045	694	100	99
rand0155.stg	1840	707	100	86
rand0028.stg	1802	683	100	92
rand0168.stg	1052	703	100	97
rand0175.stg	1022	570	100	95
rand0053.stg	975	649	100	98
rand0126.stg	2733	876	100	90
rand0100.stg	851	347	100	91
rand0114.stg	755	322	100	87
rand0131.stg	1217	439	100	87
rand0170.stg	1065	919	100	100
rand0121.stg	1427	565	100	82
rand0035.stg	872	358	100	92
rand0020.stg	2350	965	100	94

Table A.2. Task Merging on STG using $B = 1$ and $L = 100$

Table A.3 has $L = 1000$, giving even more reduction of the number of tasks of the task graphs.

File	*PT before*	*PT after*	*No. tasks before*	*No. tasks after*
rand0161.stg	26260	8095	100	89
rand0167.stg	9162	6072	100	97
rand0106.stg	9200	4101	100	98
rand0122.stg	29287	11133	100	83
rand0053.stg	9075	4034	100	86
rand0163.stg	43379	14187	100	85
rand0166.stg	5043	0	89	34
rand0163.stg	43379	14187	100	85
rand0152.stg	6099	1020	99	42
rand0157.stg	37473	11183	100	86
rand0076.stg	14203	4096	100	82
rand0071.stg	39308	12148	100	87
rand0114.stg	7055	2014	100	70
rand0171.stg	9175	5113	100	94
rand0114.stg	7055	2014	100	70
rand0168.stg	9152	4070	100	91
rand0119.stg	9152	6096	100	98
rand0102.stg	9101	5045	100	98
rand0080.stg	11075	0	97	17
rand0068.stg	38424	11182	100	90
rand0061.stg	30219	9086	100	84
rand0100.stg	8051	2031	100	81
rand0134.stg	17206	6081	100	76
rand0157.stg	37473	11183	100	86
rand0078.stg	35407	12244	100	82
rand0111.stg	5034	0	88	33
rand0027.stg	20223	7092	100	85
rand0035.stg	8072	3047	100	71
rand0025.stg	10127	2034	99	74
rand0126.stg	24333	9121	100	91
rand0040.stg	9093	4046	100	90
rand0051.stg	9069	2016	100	88
rand0052.stg	8063	2014	100	66

continued on next page

continued from previous page				
File	*PT before*	*PT after*	*No. tasks before*	*No. tasks after*
rand0148.stg	8099	3033	100	77
rand0142.stg	7086	1034	98	28
rand0158.stg	25209	6083	100	75
rand0116.stg	9145	5078	100	99
rand0155.stg	17140	4052	100	70
rand0028.stg	16202	3051	100	72
rand0168.stg	9152	4070	100	91
rand0175.stg	9122	4054	100	93
rand0053.stg	9075	4034	100	86
rand0126.stg	24333	9121	100	91
rand0100.stg	8051	2031	100	81
rand0114.stg	7055	2014	100	70
rand0131.stg	11117	3039	100	72
rand0170.stg	9165	7103	100	99
rand0121.stg	13127	4048	100	74
rand0035.stg	8072	3047	100	71
rand0020.stg	22150	6043	100	71

Table A.3. Task Merging on STG using $B = 1$ and $L = 1000$

Department of Computer and Information Science
Linköpings universitet

Dissertations

Linköping Studies in Science and Technology

No 14 **Anders Haraldsson:** A Program Manipulation System Based on Partial Evaluation, 1977, ISBN 91-7372-144-1.

No 17 **Bengt Magnhagen:** Probability Based Verification of Time Margins in Digital Designs, 1977, ISBN 91-7372-157-3.

No 18 **Mats Cedwall:** Semantisk analys av process-beskrivningar i naturligt språk, 1977, ISBN 91-7372-168-9.

No 22 **Jaak Urmi:** A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978, ISBN 91-7372-188-3.

No 33 **Tore Risch:** Compilation of Multiple File Queries in a Meta-Database System 1978, ISBN 91-7372-232-4.

No 51 **Erland Jungert:** Synthesizing Database Structures from a User Oriented Data Model, 1980, ISBN 91-7372-387-8.

No 54 **Sture Hägglund:** Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980, ISBN 91-7372-404-1.

No 55 **Pär Emanuelson:** Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation, 1980, ISBN 91-7372-403-3.

No 58 **Bengt Johnsson, Bertil Andersson:** The Human-Computer Interface in Commercial Systems, 1981, ISBN 91-7372-414-9.

No 69 **H. Jan Komorowski:** A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981, ISBN 91-7372-479-3.

No 71 **René Reboh:** Knowledge Engineering Techniques and Tools for Expert Systems, 1981, ISBN 91-7372-489-0.

No 77 **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982, ISBN 91-7372-527-7.

No 94 **Hans Lunell:** Code Generator Writing Systems, 1983, ISBN 91-7372-652-4.

No 97 **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983, ISBN 91-7372-660-5.

No 109 **Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation,1984, ISBN 91-7372-801-2.

No 111 **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984, ISBN 91-7372-805-5.

No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons, 1987, ISBN 91-7870-133-3.

No 165 **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning, 1987, ISBN 91-7870-183-X.

No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems, 1987, ISBN 91-7870-225-9.

No 174 **Johan Fagerström:** A Paradigm and System for Design of Distributed Systems, 1988, ISBN 91-7870-301-8.

No 192 **Dimiter Driankov***:* Towards a Many Valued Logic of Quantified Belief, 1988, ISBN 91-7870-374-3.

No 213 **Lin Padgham:** Non-Monotonic Inheritance for an Object Oriented Knowledge Base, 1989, ISBN 91-7870-485-5.

No 214 **Tony Larsson:** A Formal Hardware Description and Verification Method, 1989, ISBN 91-7870-517-7.

No 221 **Michael Reinfrank:** Fundamentals and Logical Foundations of Truth Maintenance, 1989, ISBN 91-7870-546-0.

No 239 **Jonas Löwgren:** Knowledge-Based Design Support and Discourse Management in User Interface Management Systems, 1991, ISBN 91-7870-720-X.

No 244 **Henrik Eriksson:** Meta-Tool Support for Knowledge Acquisition, 1991, ISBN 91-7870-746-3.

No 252 **Peter Eklund:** An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies,1991, ISBN 91-7870-784-6.

No 258 **Patrick Doherty:** NML3 - A Non-Monotonic Formalism with Explicit Defaults, 1991, ISBN 91-7870-816-8.

No 260 **Nahid Shahmehri:** Generalized Algorithmic Debugging, 1991, ISBN 91-7870-828-1.

No 264 **Nils Dahlbäck:** Representation of Discourse-Cognitive and Computational Aspects, 1992, ISBN 91-7870-850-8.

No 265 **Ulf Nilsson:** Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs, 1992, ISBN 91-7870-858-3.

No 270 **Ralph Rönnquist:** Theory and Practice of Tense-bound Object References, 1992, ISBN 91-7870-873-7.

No 273 **Björn Fjellborg:** Pipeline Extraction for VLSI Data Path Synthesis, 1992, ISBN 91-7870-880-X.

No 276 **Staffan Bonnier:** A Formal Basis for Horn Clause Logic with External Polymorphic Functions, 1992, ISBN 91-7870-896-6.

No 277 **Kristian Sandahl:** Developing Knowledge Management Systems with an Active Expert Methodology, 1992, ISBN 91-7870-897-4.

No 281 **Christer Bäckström:** Computational Complexity

of Reasoning about Plans, 1992, ISBN 91-7870-979-2.

No 292 **Mats Wirén:** Studies in Incremental Natural Language Analysis, 1992, ISBN 91-7871-027-8.

No 297 **Mariam Kamkar:** Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.

No 302 **Tingting Zhang:** A Study in Diagnosis Using Classification and Defaults, 1993, ISBN 91-7871-078-2.

No 312 **Arne Jönsson:** Dialogue Management for Natural Language Interfaces - An Empirical Approach, 1993, ISBN 91-7871-110-X.

No 338 **Simin Nadjm-Tehrani**: Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification, 1994, ISBN 91-7871-237-8.

No 371 **Bengt Savén:** Business Models for Decision Support and Learning. A Study of Discrete-Event Manufacturing Simulation at Asea/ABB 1968-1993, 1995, ISBN 91-7871-494-X.

No 375 **Ulf Söderman:** Conceptual Modelling of Mode Switching Physical Systems, 1995, ISBN 91-7871-516-4.

No 383 **Andreas Kågedal:** Exploiting Groundness in Logic Programs, 1995, ISBN 91-7871-538-5.

No 396 **George Fodor:** Ontological Control, Description, Identification and Recovery from Problematic Control Situations, 1995, ISBN 91-7871-603-9.

No 413 **Mikael Pettersson:** Compiling Natural Semantics, 1995, ISBN 91-7871-641-1.

No 414 **Xinli Gu:** RT Level Testability Improvement by Testability Analysis and Transformations, 1996, ISBN 91-7871-654-3.

No 416 **Hua Shu:** Distributed Default Reasoning, 1996, ISBN 91-7871-665-9.

No 429 **Jaime Villegas:** Simulation Supported Industrial Training from an Organisational Learning Perspective - Development and Evaluation of the SSIT Method, 1996, ISBN 91-7871-700-0.

No 431 **Peter Jonsson:** Studies in Action Planning: Algorithms and Complexity, 1996, ISBN 91-7871-704-3.

No 437 **Johan Boye:** Directional Types in Logic Programming, 1996, ISBN 91-7871-725-6.

No 439 **Cecilia Sjöberg:** Activities, Voices and Arenas: Participatory Design in Practice, 1996, ISBN 91-7871-728-0.

No 448 **Patrick Lambrix:** Part-Whole Reasoning in Description Logics, 1996, ISBN 91-7871-820-1.

No 452 **Kjell Orsborn:** On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications, 1996, ISBN 91-7871-827-9.

No 459 **Olof Johansson:** Development Environments for Complex Product Models, 1996, ISBN 91-7871-855-4.

No 461 **Lena Strömbäck:** User-Defined Constructions in Unification-Based Formalisms,1997, ISBN 91-7871-857-0.

No 462 **Lars Degerstedt:** Tabulation-based Logic Programming: A Multi-Level View of Query Answering, 1996, ISBN 91-7871-858-9.

No 475 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av hur ekonomiska styrsystem utformas och används efter företagsförvärv, 1997, ISBN 91-7871-914-3.

No 480 **Mikael Lindvall:** An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution, 1997, ISBN 91-7871-927-5.

No 485 **Göran Forslund**: Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997, ISBN 91-7871-938-0.

No 494 **Martin Sköld**: Active Database Management Systems for Monitoring and Control, 1997, ISBN 91-7219-002-7.

No 495 **Hans Olsén**: Automatic Verification of Petri Nets in a CLP framework, 1997, ISBN 91-7219-011-6.

No 498 **Thomas Drakengren:** Algorithms and Complexity for Temporal and Spatial Formalisms, 1997, ISBN 91-7219-019-1.

No 502 **Jakob Axelsson:** Analysis and Synthesis of Heterogeneous Real-Time Systems, 1997, ISBN 91-7219-035-3.

No 503 **Johan Ringström:** Compiler Generation for Data-Parallel Programming Langugaes from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.

No 512 **Anna Moberg:** Närhet och distans - Studier av kommunikationsmmönster i satellitkontor och flexibla kontor, 1997, ISBN 91-7219-119-8.

No 520 **Mikael Ronström:** Design and Modelling of a Parallel Data Server for Telecom Applications, 1998, ISBN 91-7219-169-4.

No 522 **Niclas Ohlsson:** Towards Effective Fault Prevention - An Empirical Study in Software Engineering, 1998, ISBN 91-7219-176-7.

No 526 **Joachim Karlsson:** A Systematic Approach for Prioritizing Software Requirements, 1998, ISBN 91-7219-184-8.

No 530 **Henrik Nilsson:** Declarative Debugging for Lazy Functional Languages, 1998, ISBN 91-7219-197-x.

No 555 **Jonas Hallberg:** Timing Issues in High-Level Synthesis,1998, ISBN 91-7219-369-7.

No 561 **Ling Lin:** Management of 1-D Sequence Data - From Discrete to Continuous, 1999, ISBN 91-7219-402-2.

No 563 **Eva L Ragnemalm:** Student Modelling based on Collaborative Dialogue with a Learning Companion, 1999, ISBN 91-7219-412-X.

No 567 **Jörgen Lindström:** Does Distance matter? On geographical dispersion in organisations, 1999, ISBN 91-7219-439-1.

No 582 **Vanja Josifovski:** Design, Implementation and

Evaluation of a Distributed Mediator System for Data Integration, 1999, ISBN 91-7219-482-0.

No 589 **Rita Kovordányi**: Modeling and Simulating Inhibitory Mechanisms in Mental Image Reinterpretation - Towards Cooperative Human-Computer Creativity, 1999, ISBN 91-7219-506-1.

No 592 **Mikael Ericsson:** Supporting the Use of Design Knowledge - An Assessment of Commenting Agents, 1999, ISBN 91-7219-532-0.

No 593 **Lars Karlsson:** Actions, Interactions and Narratives, 1999, ISBN 91-7219-534-7.

No 594 **C. G. Mikael Johansson:** Social and Organizational Aspects of Requirements Engineering Methods - A practice-oriented approach, 1999, ISBN 91-7219-541-X.

No 595 **Jörgen Hansson:** Value-Driven Multi-Class Overload Management in Real-Time Database Systems, 1999, ISBN 91-7219-542-8.

No 596 **Niklas Hallberg:** Incorporating User Values in the Design of Information Systems and Services in the Public Sector: A Methods Approach, 1999, ISBN 91-7219-543-6.

No 597 **Vivian Vimarlund:** An Economic Perspective on the Analysis of Impacts of Information Technology: From Case Studies in Health-Care towards General Models and Theories, 1999, ISBN 91-7219-544-4.

No 598 **Johan Jenvald:** Methods and Tools in Computer-Supported Taskforce Training, 1999, ISBN 91-7219-547-9.

No 607 **Magnus Merkel:** Understanding and enhancing translation by parallel text processing, 1999, ISBN 91-7219-614-9.

No 611 **Silvia Coradeschi:** Anchoring symbols to sensory data, 1999, ISBN 91-7219-623-8.

No 613 **Man Lin:** Analysis and Synthesis of Reactive Systems: A Generic Layered Architecture Perspective, 1999, ISBN 91-7219-630-0.

No 618 **Jimmy Tjäder:** Systemimplementering i praktiken - En studie av logiker i fyra projekt, 1999, ISBN 91-7219-657-2.

No 627 **Vadim Engelson:** Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing, 2000, ISBN 91-7219-709-9.

No 637 **Esa Falkenroth:** Database Technology for Control and Simulation, 2000, ISBN 91-7219-766-8.

No 639 **Per-Arne Persson:** Bringing Power and Knowledge Together: Information Systems Design for Autonomy and Control in Command Work, 2000, ISBN 91-7219-796-X.

No 660 **Erik Larsson:** An Integrated System-Level Design for Testability Methodology, 2000, ISBN 91-7219-890-7.

No 688 **Marcus Bjäreland:** Model-based Execution Monitoring, 2001, ISBN 91-7373-016-5.

No 689 **Joakim Gustafsson:** Extending Temporal Action Logic, 2001, ISBN 91-7373-017-3.

No 720 **Carl-Johan Petri:** Organizational Information Provision - Managing Mandatory and Discretionary Use of Information Technology, 2001, ISBN-91-7373-126-9.

No 724 **Paul Scerri:** Designing Agents for Systems with Adjustable Autonomy, 2001, ISBN 91 7373 207 9.

No 725 **Tim Heyer**: Semantic Inspection of Software Artifacts: From Theory to Practice, 2001, ISBN 91 7373 208 7.

No 726 **Pär Carlshamre:** A Usability Perspective on Requirements Engineering - From Methodology to Product Development, 2001, ISBN 91 7373 212 5.

No 732 **Juha Takkinen:** From Information Management to Task Management in Electronic Mail, 2002, ISBN 91 7373 258 3.

No 745 **Johan Åberg:** Live Help Systems: An Approach to Intelligent Help for Web Information Systems, 2002, ISBN 91-7373-311-3.

No 746 **Rego Granlund:** Monitoring Distributed Teamwork Training, 2002, ISBN 91-7373-312-1.

No 757 **Henrik André-Jönsson:** Indexing Strategies for Time Series Data, 2002, ISBN 917373-346-6.

No 747 **Anneli Hagdahl:** Development of IT-suppor-ted Inter-organisational Collaboration - A Case Study in the Swedish Public Sector, 2002, ISBN 91-7373-314-8.

No 749 **Sofie Pilemalm:** Information Technology for Non-Profit Organisations - Extended Participatory Design of an Information System for Trade Union Shop Stewards, 2002, ISBN 91-7373-318-0.

No 765 **Stefan Holmlid:** Adapting users: Towards a theory of use quality, 2002, ISBN 91-7373-397-0.

No 771 **Magnus Morin:** Multimedia Representations of Distributed Tactical Operations, 2002, ISBN 91-7373-421-7.

No 772 **Pawel Pietrzak:** A Type-Based Framework for Locating Errors in Constraint Logic Programs, 2002, ISBN 91-7373-422-5.

No 758 **Erik Berglund:** Library Communication Among Programmers Worldwide, 2002, ISBN 91-7373-349-0.

No 774 **Choong-ho Yi:** Modelling Object-Oriented Dynamic Systems Using a Logic-Based Framework, 2002, ISBN 91-7373-424-1.

No 779 **Mathias Broxvall:** A Study in the Computational Complexity of Temporal Reasoning, 2002, ISBN 91-7373-440-3.

No 793 **Asmus Pandikow:** A Generic Principle for Enabling Interoperability of Structured and Object-Oriented Analysis and Design Tools, 2002, ISBN 91-7373-479-9.

No 785 **Lars Hult:** Publika Informationstjänster. En studie av den Internetbaserade encyklopedins bruksegenskaper, 2003, ISBN 91-7373-461-6.

No 800 **Lars Taxén:** A Framework for the Coordination of Complex Systems´ Development, 2003, ISBN 91-7373-604-X

No 808 **Klas Gäre:** Tre perspektiv på förväntningar och förändringar i samband med införande av informa-

tionsystem, 2003, ISBN 91-7373-618-X.

No 821 **Mikael Kindborg:** Concurrent Comics - programming of social agents by children, 2003, ISBN 91-7373-651-1.

No 823 **Christina Ölvingson:** On Development of Information Systems with GIS Functionality in Public Health Informatics: A Requirements Engineering Approach, 2003, ISBN 91-7373-656-2.

No 828 **Tobias Ritzau:** Memory Efficient Hard Real-Time Garbage Collection, 2003, ISBN 91-7373-666-X.

No 833 **Paul Pop:** Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems, 2003, ISBN 91-7373-683-X.

No 852 **Johan Moe:** Observing the Dynamic Behaviour of Large Distributed Systems to Improve Development and Testing - An Emperical Study in Software Engineering, 2003, ISBN 91-7373-779-8.

No 867 **Erik Herzog:** An Approach to Systems Engineering Tool Data Representation and Exchange, 2004, ISBN 91-7373-929-4.

No 872 **Aseel Berglund:** Augmenting the Remote Control: Studies in Complex Information Navigation for Digital TV, 2004, ISBN 91-7373-940-5.

No 869 **Jo Skåmedal:** Telecommuting's Implications on Travel and Travel Patterns, 2004, ISBN 91-7373-935-9.

No 870 **Linda Askenäs:** The Roles of IT - Studies of Organising when Implementing and Using Enterprise Systems, 2004, ISBN 91-7373-936-7.

No 874 **Annika Flycht-Eriksson:** Design and Use of Ontologies in Information-Providing Dialogue Systems, 2004, ISBN 91-7373-947-2.

No 873 **Peter Bunus:** Debugging Techniques for Equation-Based Languages, 2004, ISBN 91-7373-941-3.

No 876 **Jonas Mellin:** Resource-Predictable and Efficient Monitoring of Events, 2004, ISBN 91-7373-956-1.

No 883 **Magnus Bång:** Computing at the Speed of Paper: Ubiquitous Computing Environments for Healthcare Professionals, 2004, ISBN 91-7373-971-5

No 882 **Robert Eklund:** Disfluency in Swedish human-human and human-machine travel booking dialogues, 2004. ISBN 91-7373-966-9.

No 887 **Anders Lindström:** English and other Foreign Linquistic Elements in Spoken Swedish. Studies of Productive Processes and their Modelling using Finite-State Tools, 2004, ISBN 91-7373-981-2.

No 889 **Zhiping Wang:** Capacity-Constrained Production-inventory systems - Modellling and Analysis in both a traditional and an e-business context, 2004, ISBN 91-85295-08-6.

No 893 **Pernilla Qvarfordt:** Eyes on Multimodal Interaction, 2004, ISBN 91-85295-30-2.

No 910 **Magnus Kald:** In the Borderland between Strategy and Management Control - Theoretical Framework and Empirical Evidence, 2004, ISBN 91-85295-82-5.

No 918 **Jonas Lundberg:** Shaping Electronic News: Genre Perspectives on Interaction Design, 2004, ISBN 91-85297-14-3.

No 900 **Mattias Arvola:** Shades of use: The dynamics of interaction design for sociable use, 2004, ISBN 91-85295-42-6.

No 920 **Luis Alejandro Cortés:** Verification and Scheduling Techniques for Real-Time Embedded Systems, 2004, ISBN 91-85297-21-6.

No 929 **Diana Szentivanyi:** Performance Studies of Fault-Tolerant Middleware, 2005, ISBN 91-85297-58-5.

No 933 **Mikael Cäker:** Management Accounting as Constructing and Opposing Customer Focus: Three Case Studies on Management Accounting and Customer Relations, 2005, ISBN 91-85297-64-X.

No 937 **Jonas Kvarnström:** TALplanner and Other Extensions to Temporal Action Logic, 2005, ISBN 91-85297-75-5.

No 938 **Bourhane Kadmiry:** Fuzzy Gain-Scheduled Visual Servoing for Unmanned Helicopter, 2005, ISBN 91-85297-76-3.

No 945 **Gert Jervan:** Hybrid Built-In Self-Test and Test Generation Techniques for Digital Systems, 2005, ISBN: 91-85297-97-6.

No 946 **Anders Arpteg:** Intelligent Semi-Structured Information Extraction, 2005, ISBN 91-85297-98-4.

No 947 **Ola Angelsmark:** Constructing Algorithms for Constraint Satisfaction and Related Problems - Methods and Applications, 2005, ISBN 91-85297-99-2.

No 963 **Calin Curescu:** Utility-based Optimisation of Resource Allocation for Wireless Networks, 2005. ISBN 91-85457-07-8.

No 972 **Björn Johansson:** Joint Control in Dynamic Situations, 2005, ISBN 91-85457-31-0.

No 974 **Dan Lawesson:** An Approach to Diagnosability Analysis for Interacting Finite State Systems, 2005, ISBN 91-85457-39-6.

No 979 **Claudiu Duma:** Security and Trust Mechanisms for Groups in Distributed Services, 2005, ISBN 91-85457-54-X.

No 983 **Sorin Manolache:** Analysis and Optimisation of Real-Time Systems with Stochastic Behaviour, 2005, ISBN 91-85457-60-4.

No 986 **Yuxiao Zhao:** Standards-Based Application Integration for Business-to-Business Communications, 2005, ISBN 91-85457-66-3.

No 1004 **Patrik Haslum:** Admissible Heuristics for Automated Planning, 2006, ISBN 91-85497-28-2.

No 1005 **Aleksandra Tesanovic:** Developing Reusable and Reconfigurable Real-Time Software using Aspects and Components, 2006, ISBN 91-85497-29-0.

No 1008 **David Dinka:** Role, Identity and Work: Extending the design and development agenda, 2006, ISBN 91-85497-42-8.

No 1009 **Iakov Nakhimovski:** Contributions to the Modeling and Simulation of Mechanical Systems with Detailed Contact Analysis, 2006, ISBN 91-85497-43-X.

No 1013 **Wilhelm Dahllöf:** Exact Algorithms for Exact Satisfiability Problems, 2006, ISBN 91-85523-97-6.

No 1016 **Levon Saldamli:** PDEModelica - A High-Level Language for Modeling with Partial Differential Equations, 2006, ISBN 91-85523-84-4.

No 1017 **Daniel Karlsson:** Verification of Component-based Embedded System Designs, 2006, ISBN 91-85523-

79-8.

No 1018 **Ioan Chisalita:** Communication and Networking Techniques for Traffic Safety Systems, 2006, ISBN 91-85523-77-1.

No 1019 **Tarja Susi:** The Puzzle of Social Activity - The Significance of Tools in Cognition and Cooperation, 2006, ISBN 91-85523-71-2.

No 1021 **Andrzej Bednarski:** Integrated Optimal Code Generation for Digital Signal Processors, 2006, ISBN 91-85523-69-0.

No 1022 **Peter Aronsson:** Automatic Parallelization of Equation-Based Simulation Programs, 2006, ISBN 91-85523-68-2.

Linköping Studies in Information Science

No 1 **Karin Axelsson:** Metodisk systemstrukturering- att skapa samstämmighet mellan informa-tionssystemarkitektur och verksamhet, 1998. ISBN-9172-19-296-8.

No 2 **Stefan Cronholm:** Metodverktyg och användbarhet - en studie av datorstödd metodbaserad systemutveckling, 1998. ISBN-9172-19-299-2.

No 3 **Anders Avdic:** Användare och utvecklare - om anveckling med kalkylprogram, 1999. ISBN-91-7219-606-8.

No 4 **Owen Eriksson:** Kommunikationskvalitet hos informationssystem och affärsprocesser, 2000. ISBN 91-7219-811-7.

No 5 **Mikael Lind:** Från system till process - kriterier för processbestämning vid verksamhetsanalys, 2001, ISBN 91-7373-067-X

No 6 **Ulf Melin:** Koordination och informationssystem i företag och nätverk, 2002, ISBN 91-7373-278-8.

No 7 **Pär J. Ågerfalk:** Information Systems Actability - Understanding Information Technology as a Tool for Business Action and Communication, 2003, ISBN 91-7373-628-7.

No 8 **Ulf Seigerroth:** Att förstå och förändra systemutvecklingsverksamheter - en taxonomi för metautveckling, 2003, ISBN91-7373-736-4.

No 9 **Karin Hedström:** Spår av datoriseringens värden - Effekter av IT i äldreomsorg, 2004, ISBN 91-7373-963-4.

No 10 **Ewa Braf:** Knowledge Demanded for Action - Studies on Knowledge Mediation in Organisations, 2004, ISBN 91-85295-47-7.

No 11 **Fredrik Karlsson:** Method Configuration - method and computerized tool support, 2005, ISBN 91-85297-48-8.

No 12 **Malin Nordström:** Styrbar systemförvaltning - Att organisera systemförvaltningsverksamhet med hjälp av effektiva förvaltningsobjekt, 2005, ISBN 91-85297-60-7.

No 13 **Stefan Holgersson:** Yrke: POLIS - Yrkeskunskap, motivation, IT-system och andra förutsättningar för polisarbete, 2005, ISBN 91-85299-43-X.