Linköping Studies in Science and Technology

Thesis No. 933

Automatic Parallelization of Simulation Code from Equation Based Simulation Languages

by

Peter Aronsson

INSTITUTE OF TECHNOLOGY
LINKÖPINGS UNIVERSITET

Submitted to the School of Engineering at Linköping University in partial fulfillment of the requirements for degree of Licentiate of Engineering

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

Linköping 2002

Automatic Parallelization of Simulation Code from Equation Based Simulation Languages

by

Peter Aronsson

April 2002
ISBN 91-7373-280-X
Linköping Studies in Science and Technology
Thesis No. 933
ISSN 0280-7971
LiU-Tek-Lic-2002:06

ABSTRACT

Modern state-of-the-art equation based object oriented modeling languages such as Modelica have enabled easy modeling of large and complex physical systems. When such complex models are to be simulated, simulation tools typically perform a number of optimizations on the underlying set of equations in the modeled system, with the goal of gaining better simulation performance by decreasing the equation system size and complexity. The tools then typically generate efficient code to obtain fast execution of the simulations. However, with increasing complexity of modeled systems the number of equations and variables are increasing. Therefore, to be able to simulate these large complex systems in an efficient way parallel computing can be exploited.

This thesis presents the work of building an automatic parallelization tool that produces an efficient parallel version of the simulation code by building a data dependency graph (task graph) from the simulation code and applying efficient scheduling and clustering algorithms on the task graph. Various scheduling and clustering algorithms, adapted for the requirements from this type of simulation code, have been implemented and evaluated. The scheduling and clustering algorithms presented and evaluated can also be used for functional dataflow languages in general, since the algorithms work on a task graph with dataflow edges between nodes.

Results are given in form of speedup measurements and task graph statistics produced by the tool. The conclusion drawn is that some of the algorithms investigated and adapted in this work give reasonable measured speedup results for some specific Modelica models, e.g. a model of a thermofluid pipe gave a speedup of about 2.5 on 8 processors in a PC-cluster. However, future work lies in finding a good algorithm that works well in general.

This work has been supported by NUTEK (Swedish National Board for Industrial and Technical Development), the EU-IST projects Joses and RealSim, and MathCore AB.

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

Acknowledgements

Several people have made this work possible. First, I would like to thank my supervisor, Peter Fritzson, for awakening my interests of research studies and introducing me to Modelica and parallel computing. He has also been very helpful in improving my writing. I would also like to thank two other persons who have helped me in many ways, Christoph Kessler and Johan Gunnarson.

Many thanks goes to all the employees at PELAB who have contributed to a nice working atmosphere with interesting discussions over the coffee breaks. The same gratitude also goes to all employees at MathCore, and I also thank MathCore for financing part of my research and letting me take part in the evolution of the company. I also would like to thank Dynasim for providing the Dymola software.

Finally, I would like to thank my family, who has supported me in many ways, especially my girl friend Marie, who has supported me tremendously and who I love very much.

Peter Aronsson
Linköping, February 2002

Contents

List of Figures

Chapter 1

Introduction

In this chapter we introduce the area of modeling and simulation, and explain why it is important to parallelize the simulation code generated from simulation tools for equation based modeling languages.

1.1 Modeling and Simulation

Modeling and simulation tools are becoming a powerful aid in the product development process. In order to shorten the time for developing and manufacturing a product, also known as time-to-market, modeling and simulation have an important role to play. By building a model of the product using advanced tools and languages, and simulate 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. This leads to shorter development time, since the earlier an error is detected, the cheaper it is to correct.

Quite recently in the history of modeling and simulation technology, models were built by hand. The equations and formulas describing the physical behavior of a system described by a model were written by hand and gradually 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 much of the work was made manually, it was expensive to maintain and change models in order to adapt them to new requirements. In fact, this manual development of models is still used today, but is gradually replaced by using automatic tools.

In this manual approach still used today, the knowledge of the models typically is divided between different places and persons. Some people are responsible for the physical behavior of the model with knowledge about the equations and variables of the model. Other people know how the equations are implemented in the programming language used for the simulation of the model. This also make maintenance expensive and reuse of models difficult.

To remedy these problems, object oriented modeling languages such as Modelica [34] were developed. By using an object oriented modeling language it is possible to describe the physical behavior of individual objects by using Modelica classes, corresponding to models of real physical objects. These 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 gained.

Also, 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") of the object is left unspecified. This makes it possible to use the component both as an input and as an output. 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 The Need for Parallelization

With increasing complexity and size of modeled systems, the need for parallelization of the simulation code for such systems is emerging. An object oriented equation based modeling language makes it feasible for a less experienced user to build large and complex models, with the help of advanced graphical modeling tools [14, 17, 31]. One such tool is MathModelica [31], where models can be built in the model editor in a drag and drop fashion as illustrated in Figure 1.1. Typically, a large model can contain over one hundred thousand equations, which can be both differential and algebraic equations. When simulating such a model it is necessary to solve the equations for the state variables and their derivatives for each time step of the numerical solver. For large models this can be very time consuming and the need of parallelization of these calculations are large.

In this thesis we develop and evaluate methods to speed up the execution of the simulations of large and complex models described in the Modelica modeling language. The speedup is achieved by an automatic parallelization tool that translates the sequential simulation code generated for a Modelica model into a parallel version of the code. This parallel code is executed on a parallel computer giving a speedup of executing the simulation.

The automatic parallelization tool translates the sequential simulation code into a internal representation in the form of a task graph, which is a directed acyclic graph. Each node in this graph is a small piece of computation, like an expression or a function call. The edges of the task graph represent data that is sent between tasks. Communication and execution costs are associated with the edges and nodes of the task graph respectively. This information is used by the clustering and scheduling algorithms that traverse the task graph, resulting in a partitioning of the graph for a specific parallel computer. By using the standard task graphs as input to the scheduling and clustering algorithms the tool can be generalized to work for any functional dataflow language. Finally, the automatic parallelization tool produces platform independent simulation code for execution on the specified parallel computer.

Such an automatic parallelization tool would typically be integrated into a modeling and simulation environment so that an engineer could easily use it in an intuitive way. This requirement emphasizes the need of making the tool fully automatic, to eliminate the complexity of parallelization for the user.

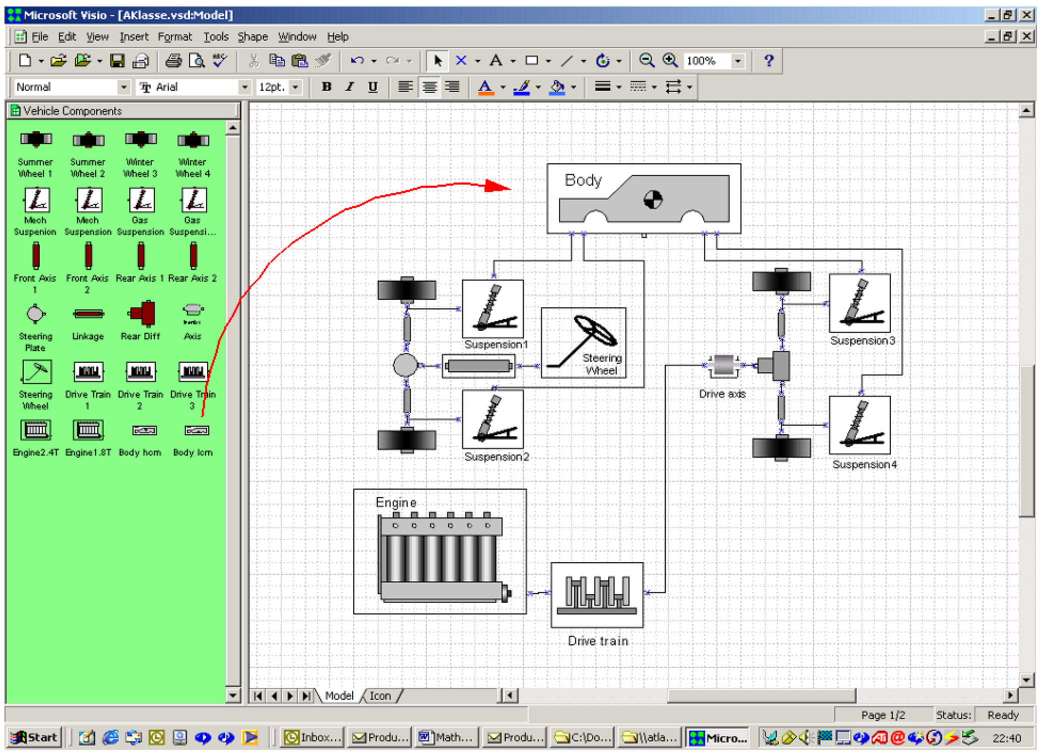


Figure 1.1. The model editor in the MathModelica modeling and simulation environment, showing a model of a car.

1.3 Thesis Overview

The rest of this thesis is organized as follows. Chapter two gives some background regarding the Modelica language as well as basic concepts in parallel computation and data dependency graphs used by scheduling and clustering algorithms. Chapter three presents related work in scheduling and clustering algorithms. Chapter four presents the research problem of this thesis. In Chapter five an automatic parallelization tool is presented. This tool is built to validate the hypothesis regarding feasible automatic parallelization stated in the section about the research problem. Chapter six presents some contributions in the form of scheduling and clustering algorithms, followed by performance results of these algorithms in chapter seven. A summary of the work achieved so far is given in chapter eight. Finally, conclusions and future work is presented in chapter nine.

Chapter 2

Background

This chapter presents the modeling language Modelica, which is an equation based object oriented modeling language. This language has been designed for increasing the reuse of model components and with the capability to build complex hierarchical models of physical systems in a natural way. The chapter also gives a background in parallel computation and automatic parallelization.

2.1 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 [35], a modern object oriented equation based modeling language well suited for modeling of large and complex physical systems.

A trivial example Modelica model with two variables and two ordinary differential equations (ODE) is given in Figure 2.1. The variables are of the builtin type `Real`. The `x` variable has an optional modification of the start attribute, setting its value to 5.2 compared to the default value of 0 for Real variables. The `start` value is used by the simulation tool to determine initial values for the simulated model. 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 `y` variable from simulating the `ODE` model for ten seconds is plotted in Figure 2.2.

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

Figure 2.1. A small ODE example in Modelica.

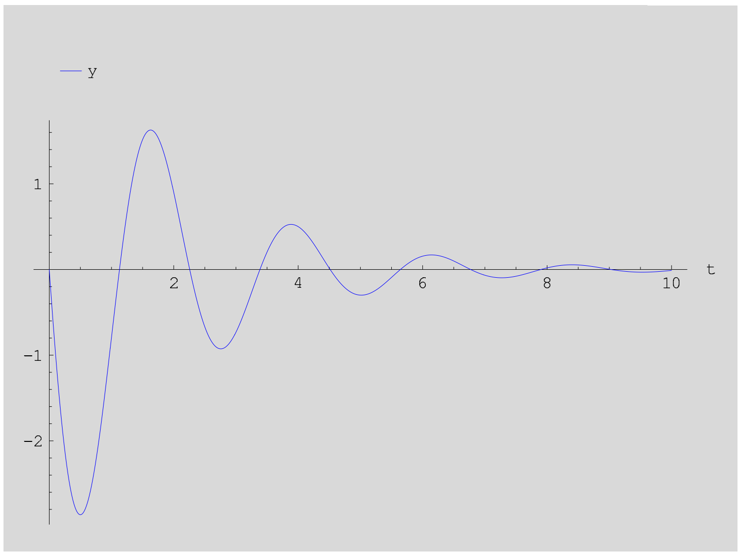


Figure 2.2. The plot of the solution variable y after simulating for 10 seconds.

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*. Classes can be instantiated inside other classes, enabling an hierarchical modeling methodology. Basically, the end user can build complex models by instantiating classes to create objects inside user defined model definitions and connecting these objects together.

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

- **Inheritance**
  A class can inherit equations and variables from a base class. Local classes are also inherited.

- **Redeclarations**
  The type of an instance can be replaced by a new type, which means that the old declaration using the old type is replaced by a new declaration using the replaced type. This is a special case of inheritance.

- **Modifications**
  A class or an instance of a class can be modified using a list of modifications, changing for instance parameter values, constants, variables or even local classes in that class. This is a simple, but quite common, variant of redeclaration where the type of a variable is not changed.

Figure 2.3 shows these three language constructs in a small Modelica example.

```
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;

                        // Redeclaration
MiniCircuit tempcircuit1(redeclare TempResistor R1);

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

Figure 2.3. Three language constructs that enables reuse.

Another important language construct in Modelica is the kind of restricted class called *connector*, along with the `connect` operator. Instances of connector classes constitute the interfaces of a component, i.e. how it connects to the outer world. A connector class contains instances of 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) [34] contain two connector classes for electrical components (one for the positive pin and one for the negative pin of an electrical component). The positive pin class definition is:

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

Since most electrical components have two pins, this information is collected into a class called `OnePort`[1], 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 stand-alone 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 only three equations and five variables, 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

[1]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.

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
   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 are already provided in Modelica Standard Library, any basic electrical circuit can easily be modeled. For instance, the simple electrical circuit in Figure 2.4 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;
```

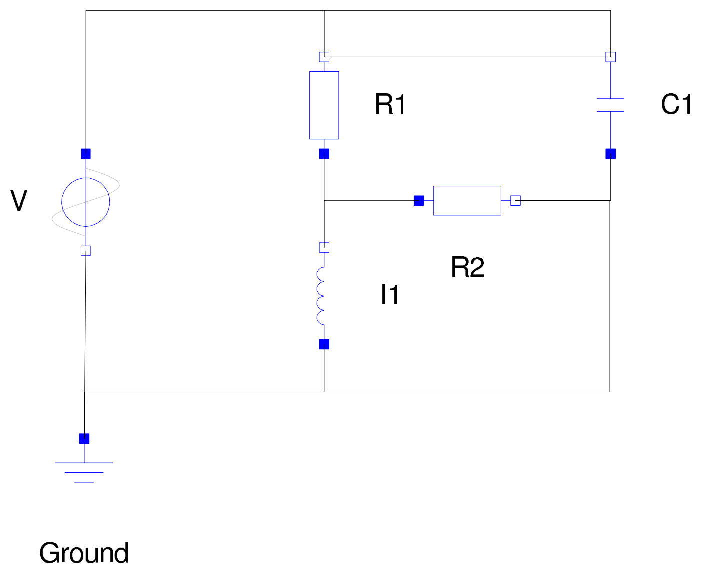


Figure 2.4. An electrical circuit resulting in a DAE problem.

2.2 Modelica Compilation

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 statements. 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.

An ODE system on explicit form can be expressed as:

$$\dot{X} = f(X, t) \tag{2.1}$$

where X is a vector of all state variables. A DAE system on implicit form is expressed as follows:

$$\begin{aligned} g(\dot{X}, X, Y, t) &= 0 \\ h(X, Y, Z) &= 0 \end{aligned} \tag{2.2}$$

where X is the vector of state variables and Y is the vector of algebraic variables. When the DAE system is on an implicit form the equation system has to be solved for the derivatives and the algebraic variables.

For example, the equations generated from flattening the `DAECircuit` model is shown in Figure 2.5. Equation 1-4 originates from the Resistor instance `R1`, 5-8 from `R2`, etc. Equation 23-25 are generated from the two connect statements 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 2.2.

After the flattening phase of a Modelica compilation, several optimizations on the 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. An efficient Modelica compiler such as Dymola [14], will perform several equation oriented optimizations. Among these optimizations are:

- **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. The equations used for simplifications can also have a more complicated structure.

- **BLT transformation**
  The BLT transformation transforms a system of equations into Block Lower Triangular form, which will identify blocks of equations that form subsystems of equations. This is done by building a bipartite graph of all equations and variables and identify strongly connected components in that graph. The vertices of the bipartite graph are of two types; equation vertice or variable vertice. 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.

  The BLT transformation gives a unique representation of subsystems of equations [22], where some of the subsystems can be solved analytically while others constitute a system of equations that needs to be solved by numerical solvers.

- **Index reduction**
  Index reduction is an optimization performed to reduce the index of a system of equations. The index of a system of equations corresponds to the number of times a variable in the equation system must be differentiated before the equation system can be solved. 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 solver can handle, the index needs to be reduced. For instance, models of mechanical systems typically are index 3 problems.

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	$V.src.outPort.signal\_1 = V.src.p\_offset\_1$ $+(if time < V.src.p\_startTime\_1 then 0 else$ $V.src.p\_amplitude\_1 * sin(2 * V.src.pi$ $*V.src.p\_freqHz\_1 * (time - V.src.p\_startTime\_1)$ $+V.src.p\_phase\_1))$	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 2.5. The complete set of equations generated from the `DAECircuit` model.

- **Tearing of equations**
  Tearing of equations is yet another way of optimizing equation sys-

tems [16]. Tearing is used to break algebraic loops in a system of equations. A system of equations containing an algebraic loop is a simultaneous system of equations, which is a strongly connected component or subsystem of equations after BLT transformation. These loops are found by detecting loops in the bipartite graph built in the BLT transformation.

- **Mixed Mode Integration**
  Mixed mode integration is a method for breaking apart an equation system and using several numeric solvers together instead of one [46]. 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 stepsize and better 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 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.

- **Inline Integration**
  Inline Integration is an optimization method that inserts an inline expansion of the numerical solver into the equation system parts [15]. By doing so, and again performing BLT transformations, etc., a substantial speedup in simulation time can be achieved. The inline integration method has successfully been combined with mixed mode integration to further reduce simulation time [46].

When all optimizations have been performed on the system of equations code is generated. The generated code corresponds to the calculation of the right hand side of Equation 2.1, i.e. the calculation of f. For the DAE case, i.e. Equation 2.2, the subsystems of equations that are on explicit form are generated as for the ODE case, with assignment statements of arithmetic expressions to variables. But for the subsystems of equations which are on implicit form code that solves the subsystems numerically is generated, and/or linked with a numerical solver.

2.3 Exploiting Parallel Computing

Efficient simulation is becoming more important as the modeled systems are increasing 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 reasonable 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. However, the number of equations and variables of such systems tend to grow over time since it is easier to build large simulation models when using Modelica. Therefore, to increase the size of problems that can be efficiently simulated it is necessary to exploit all possible ways of reducing the execution time of the simulation.

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 method. For instance, by using a Runge-Kutta method in the numerical solver some degree of parallelism can be detected within the numerical solver [43]. Since the Runge-Kutta methods involve calculations of several time steps simultaneously, parallelism is easy to extract by letting each time step calculation be performed in parallel.

- **Parallelism over time**
  A second alternative is to parallelize a simulation over time. This approach is however best suited for discrete event simulations and less suitable for simulation of continuous systems, since the solution 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 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 2.1 and 2.2).

The simulation code consist of two separate parts, a numerical solver and the code that computes new values of the state variable, 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 and 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 used for the calculation of f or g. Therefore, we focus on parallelizing the computation of these parts. This approach has for instance successfully been used in [3, 20].

When the simulation code has been parallelized, timing measurements on the execution time of the simulation code is performed. To measure parallel programs in a problem size independent way the execution time of the parallel program is not a suitable metric. Instead the term *relative* speedup is used, defined as [19]:

$$S_{relative} = \frac{T_1}{T_N} \tag{2.3}$$

where

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

This speedup is called relative because the same program is used for measurement of both the sequential time and the parallel time. However, there might exist a more efficient sequential implementation of the program. Therefore, there is also a definition for *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:

$$S_{absolute} = \frac{T_{seq}}{T_N} \tag{2.4}$$

where

- T_{seq} is the execution time of the most effective sequential implementation and
- 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, i.e. the code produced by the standard Modelica compiler, the speedup definition used through out the rest of this thesis is the absolute speedup.

2.4 Task Graphs

A task graph is a Directed Acyclic Graph (DAG), with costs associated with the edges and the nodes. It is described by the tuple

$$G = (V, E, c, \tau) \tag{2.5}$$

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.
- $c(e)$ 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 2.6 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.

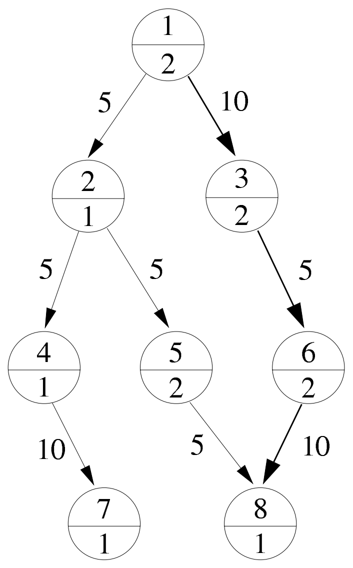


Figure 2.6. Task graph with communication and execution costs.

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 $pred(n)$. 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 $succ(n)$.

A *join* node is a node with more than one predecessor, illustrated in Figure 2.7(a). A *split* node is a node with more than one successor node, see Figure 2.7(b).

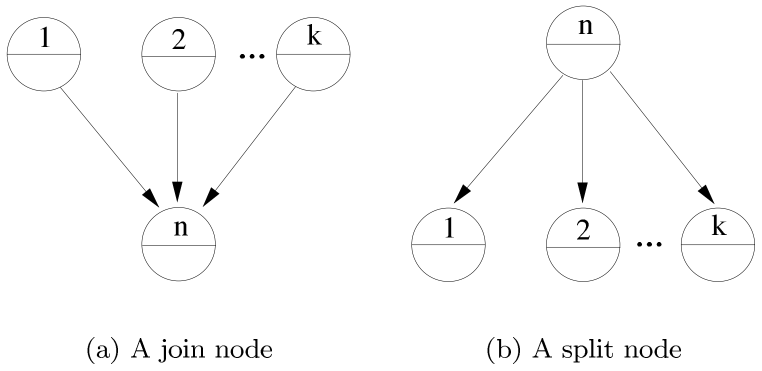


Figure 2.7. Graph definitions

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 before the node itself can start to execute.

Since the task graph representation is used in this work as an input for the scheduling and clustering algorithms the work 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.

2.4.1 Graph Attributes For Scheduling Algorithms

The scheduling algorithms use additional values, or attributes, mostly associated with the nodes of the task graph. Some attributes are common to several algorithms. Others are specific for one particular algorithm. This section defines a subset of such attributes.

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 2.6 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 [6, 49, 55], 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 [55].

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

$$level(n) = \begin{cases} 0 & ,pred(n) = \emptyset \\ \max_{k \in pred(n)}(level(k) + \tau(k) + c(k,n)) & ,pred(n) \neq \emptyset \end{cases} \tag{2.6}$$

The relation between the critical path and the level attribute is that for nodes on the critical path, the level of a node will always be the maximum value among all predecessors of the node.

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 [54]. We use the terms $est(n)$ and $last(n)$ and the definitions found in [12], which will later be used when explaining the TDS algorithm in Section 3.2.1.

$$est(n) = \begin{cases} 0 & ,pred(n) = \emptyset \\ \min_{k \in pred(n)} \max_{l \in pred(n), k \neq l}(ect(k), ect(k) + c_{k,n}) & ,pred(n) \neq \emptyset \end{cases} \tag{2.7}$$

$$ect(n) = est(n) + \tau(n) \tag{2.8}$$

$$fpred(n) = max_{k \in pred(n)}(ect(k) + c_{k,n}) \tag{2.9}$$

$$lact(n) = \begin{cases} ect(n) & ,succ(n) = \emptyset \\ \min\limits_{k \in succ(n), k \neq fpred(n)} (min\ (last(k) - c_{n,k}, \min\limits_{k \in succ(n), k = fpred(n)} (last(k)))) & ,succ(n) \neq \emptyset \end{cases} \tag{2.10}$$

$$last(n) = lact(n) - \tau(n) \tag{2.11}$$

- *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 2.7.

- *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 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 2.8 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 execute to fulfill an optimal schedule, as defined in Equation 2.11.

- *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 2.10.

- *fpred(n)* is the *favorite predecessor* of a node n, see Equation 2.9, 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 like 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.

2.5 Parallel Programming Models

Developing parallel programs, whether it is performed automatically by a tool or manually by a developer or a team of developers, is an advanced and error prone process. It can be difficult for a developer to estimate the structural or computational complexity of the program he/she has written. By having a programming model to follow, the programmer can be guided both in the design and the implementation of his or 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 varying sized problem instances.

When choosing a parallel programming model, there are several factors to consider. 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 with 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 is simple enough to minimize the effort needed for developers to implement and understand their parallel programs.

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

2.5.1 The PRAM Model

The Parallel Random Access Machine (PRAM) model [18] is a simple programming model and also the most popular one. Its popularity is due to its simplicity, all global data is instantly available on all processors. This is a powerful simplification which can also 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, its major drawback is also the simplicity. The communication cost is neglected, which often leads to large differences between the model and the reality.

2.5.2 The Logp Model

The Logp model [10] 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 if 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.

2.5.3 The BSP Model

In the Bulk-Synchronous Parallel (BSP) model [50] 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 *superstep.* 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 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.

2.5.4 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 2.4. Some authors instead use the term *macro data flow model*. 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. The delay model is used in the work presented in this thesis. However, a discussion about choosing other models is given in chapter 9.

Chapter 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 parallel programs for execution on multiprocessor architectures. Early research in the area focused on simple 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. This chapter introduces some of the many scheduling and clustering algorithms found in literature and explains common techniques used in these scheduling and clustering algorithms.

3.1 Task Scheduling

A task scheduling algorithm traverses a directed acyclic graph (DAG), as defined in Section 2.4. 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 proven to be NP-complete [48], thus it is common to use heuristics in scheduling algorithms.

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. There are even algorithms for a specific number of processors, e.g. for two processors there exist an optimal scheduling algorithm [9].

3.1.1 Classification

There exists a general classification scheme for task scheduling algorithms [8], which gives a hierarchical categorization of the algorithms, see Figure 3.1. At the top level, algorithms belong to one of two categories, *local* or *global* scheduling. Local scheduling involves scheduling of tasks locally on a 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 subcategorized into *optimal* and *suboptimal* scheduling.

The *suboptimal* category contains two subcategories, *heuristic* and *approximate* scheduling. 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.

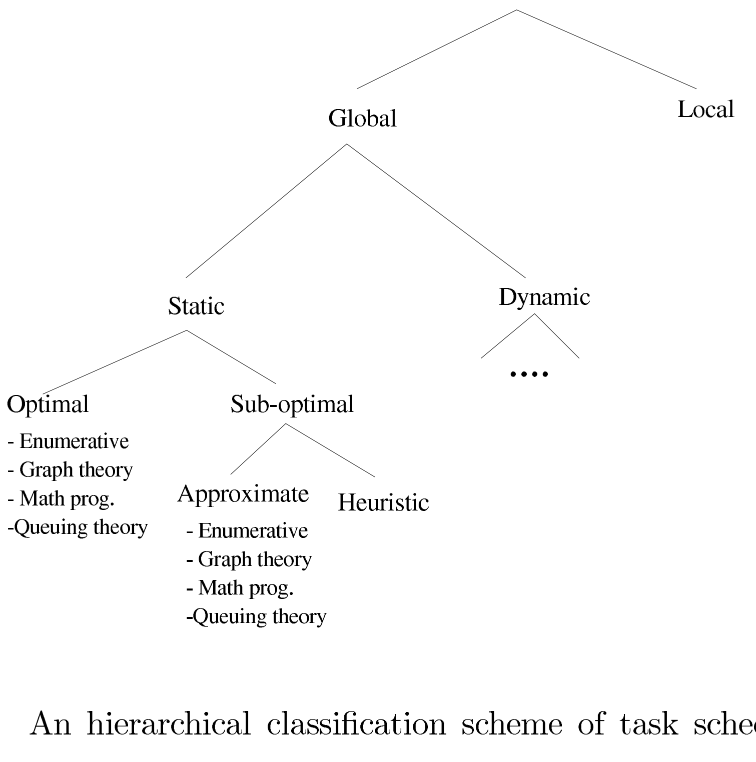


Figure 3.1. An hierarchical classification scheme of task scheduling algorithms.

3.1.2 List Scheduling Algorithms

The list scheduling technique has been extensively studied in the literature [21, 26, 41, 42, 47]. The list scheduling algorithms belong to the *heuristics* category in the classification scheme given in section 3.1.1. 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 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 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 *level* of the node, see Equation 2.6. If the level of a node has a high value, there are many computations to be performed after the node has finished its execution. Therefore, that node should be given a higher priority when choosing nodes from the ready list, compared to another node with a much lower level value.

When a node has been scheduled it is removed from the ready list and potentially successor nodes to the scheduled node are added. The algorithm terminates when the ready list is empty, i.e. all nodes have been 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.

The ERT Algorithm

One list scheduling algorithm is the Earliest Ready Task (ERT) algorithm [26]. Initially all tasks without any predecessors are put 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 added to the communication time among all parents of the task. This calculation is shown in Equation 3.1 as defined in [26]. In the ERT algorithm, the communication cost has been divided into two parameters: the first parameter is the size of the message sent between tasks, $d(k, n)$ in Equation 3.1. The second parameter, $t_{comm}(P_i, P_j)$, is the time required to send one message from processor P_i to processor P_j. Finally, the *Alloc* function in Equation 3.1 performs the allocation of a task, returning a processor number.

$$X(n, P_i) = max_{k \in pred(n)}(F(k) + d(k, n) * t_{comm}(Alloc(k), P_i)) \tag{3.1}$$

However, the calculation of the ready time above can not always be met as the real ready time in a schedule, since no check if the processor is available at that time is processed. Therefore, once the ready time has been calculated for *all* processors, the real ready time is calculated by considering if each processor is available or not, as performed in Equation 3.2.

$$R(n, P_i) = max(Avail(P_i), X(n, P_i)) \tag{3.2}$$

Then, the processor giving the earliest starting time is calculated for each task, see Equation 3.3 (m is the number of processors).

$$R(n) = min_{j \in \{1,..,m\}} R(n, P_j) \tag{3.3}$$

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 [26] 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.

3.1.3 Graph Theory Oriented Algorithms

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 [39] and several clustering approaches [11, 27].

The critical path scheduling technique identifies the critical path of the task graph, see Section 2.4.1. Then it schedules all 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.

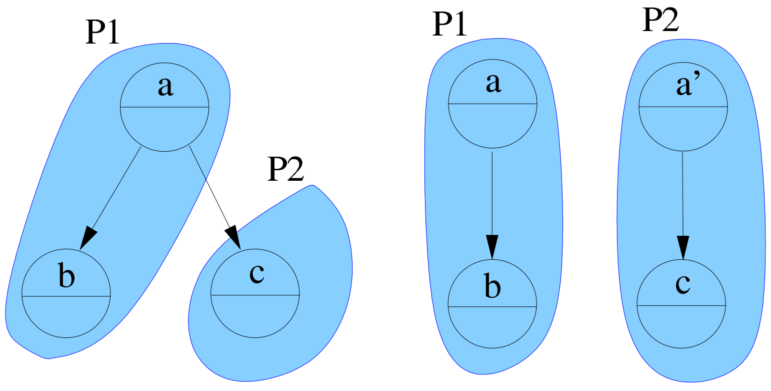
3.1.4 Orthogonal Considerations

The classification scheme presented in section 3.1.1 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 duplication

One approach of improving the efficiency of a scheduling algorithm that has increased in popularity over the past decade is to employ task duplication as a means of reducing the communication cost [12, 25, 29, 38]. The use of task duplication to reduce the total execution time of a parallel task graph is illustrated in Figure 3.2.

For certain applications where the cost of communication is far from negligible, duplicating a task to several processors can decrease the execution time significantly [25]. A drawback of using task duplication in a scheduling algorithm is that it increases the time complexity of the algorithm. The increase can be substantial. For instance, the CPFD algorithm[25] has a time complexity of $O(n^4)$.



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

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

Figure 3.2. Using task duplication to reduce total execution time.

Granularity

An important metric for task graphs with communication costs as defined in 2.5 is the granularity of the graph. The literature also contain variants on the definition of granularity [25, 28, 37]. 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 of granularity [25]. The granularity g is defined by Equation 3.4.

$$g = \frac{\max_{e \in E} c(e)}{\min_{n \in V} \tau(n)} \tag{3.4}$$

The granularity factor is an important metric for task graph scheduling. A fine grained task graph, i.e. when the granularity value is high 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 duplication 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 decreasing the granularity of the task graph by merging tasks [23].

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.

3.2 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.

3.2.1 TDS Algorithm

The TDS (Task Duplication based Scheduling) algorithm is a *linear clustering* algorithm, with task duplication. 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 2.9.

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 3.5 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 ($last(x)$).

$$last(x) - lact(y) < c_{x,y} \tag{3.5}$$

The TDS algorithm will only duplicate tasks that are critical, keeping the number of duplicated 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.

3.2.2 The Internalization Algorithm

A task clustering algorithm called internalization is presented in [49]. 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.

3.2.3 The Dominant Sequence Clustering Algorithm

Another task clustering algorithm specially designed for a low time complexity is the *Dominant Sequence Clustering* (DSC) algorithm [55]. 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 3.2.2 above.

The simplified version of the algorithm is given in Figure 3.3. 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 $tlevel(n_f)$
        decrease 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 3.3. The simplified DSC algorithm, DSCI.

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 3.1.2.

The priority which is used for selecting a task in the first step of the loop is defined in Equation 3.6.

$$PRIO(n_f) = tlevel(n_f) + blevel(n_f) \tag{3.6}$$

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:

$$PT = max_{v \in V} PRIO(n) \tag{3.7}$$

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 cluster 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 becomes zero. The merge operation is also responsible for adding pseudo edges such that each cluster has a determined schedule. Figure 3.4 clarifies the addition of a pseudo edge.

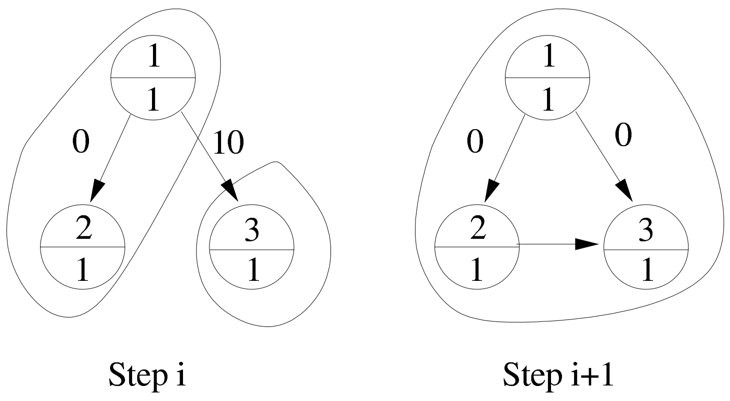


Figure 3.4. The addition of pseudo edges when performing a DSC-merge.

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

In [55] the initial version of the DSC algorithm is presented (also shown in Figure 3.3) 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 be merged such that the critical path is not reduced to a maximum degree, see [55] 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 3.5, also found in [55] 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.

3.3 Task Merging

Task merging is *stronger* in the way tasks are joined compared to *task clustering*. When tasks are clustered, they are only assumed to be put on the same processor, which means that the communication cost between tasks belonging to the same cluster are zero. Unfortunately 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

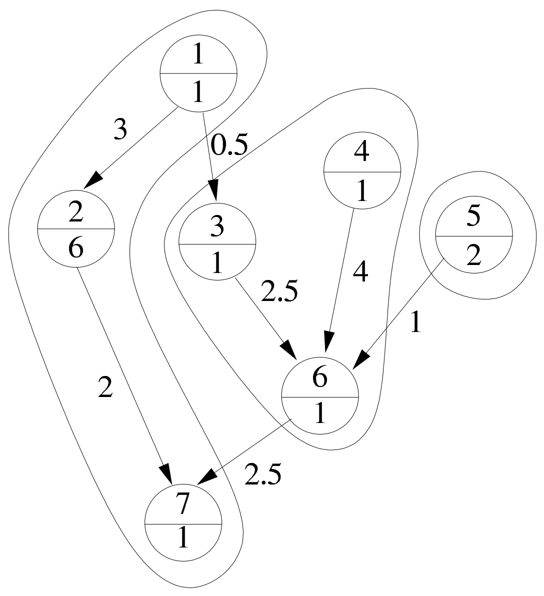


Figure 3.5. A task graph clustered by the DSC algorithm

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 3.6 shows how a merge is performed. The cluster of tasks are merged into a new task graph that still is a DAG.

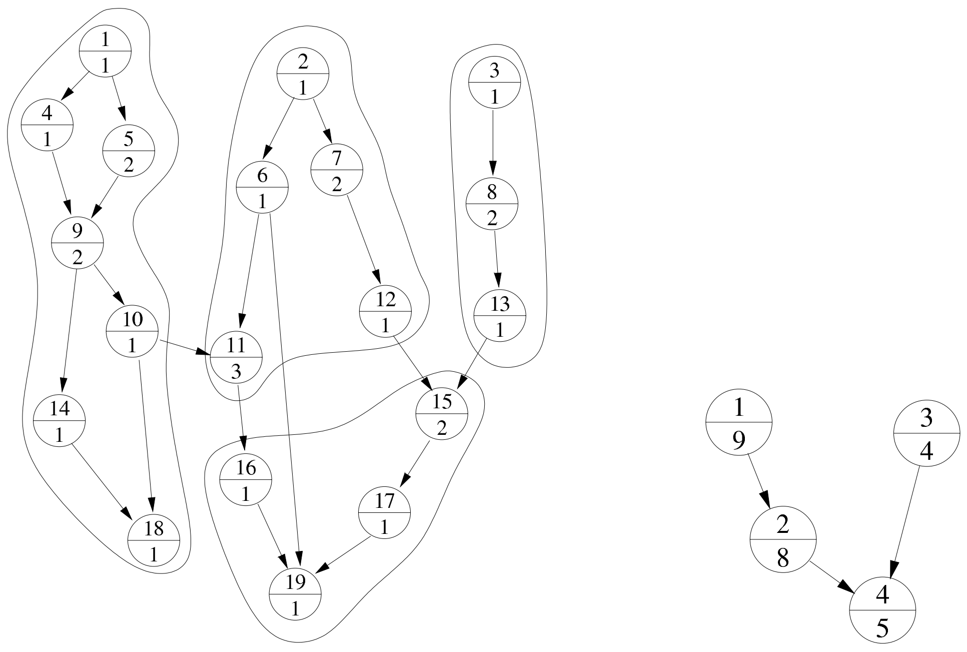
3.3.1 The Grain Packing Algorithm

A combined scheduling and task merging technique called *grain packing* is presented in [23, 24]. The grain packing algorithm is designed to handle fine grained task graphs, i.e. task graphs with high granularity. The grain packing algorithm is a complete scheduling algorithm, i.e. it schedules fine grained task graph onto a fixed number of processors.

The algorithm is divided into four steps [24]:

- **Building a task graph**
  The first stage is to build a fine grained task graph. This approach 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



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

(b) The resulting task graph after merge.

Figure 3.6. An example of task merging.

fixed number of processors. In [24] 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 duplication, i.e. duplicating grains from other processors to further reduce execution time.

- **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.

3.3.2 A Task Merging Algorithm

In [6] is a task merging algorithm 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 [6], 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 task causing a cycle can not be performed.

3.4 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 high granularity values (according to the definition of granularity given in Section 3.1.4), 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 as targeted in this work. One such coarse grained approach, also targeting simulation code, is described in [53].

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.

Chapter 4

The Research Problem

This chapter states the major research problem of this work. First, a hypothesis is stated which captures the essential research problem in a single sentence. Then follows a division of the research problem stated in the hypothesis into three parts. Finally we discuss the relevance of the work as well as the scientific method.

4.1 Forming a Hypothesis

The research problem can be summarized by the following hypothesis:

Hypothesis 1
It is possible to build an automatic parallelization tool that efficiently 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 computer than on a sequential one.

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 such that producing the parallel code is possible within reasonable time limits. The parallel code should also be efficient. This means that the parallelization should be worthwhile, 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.

4.2 Subproblems

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

4.2.1 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 [3] for an equation based modeling language called ObjectMath [52, 33].

The highest level where parallelism can be extracted is at the component level. Each component of a large complex system usually contain many equations. The computational work work for each component can potentially be put on one processor, with communication of the connector variables in between processors. However, earlier work showed that in the general case there is 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 produced better parallelism compared to extracting parallelism at the component level, but the degree of parallelism was in general not sufficient.

The third level is to go down to the subequation level, where we consider parallelism between parts of equations like for instance arithmetic operations. At this level, the largest degree of parallelism was found among the three levels [3].

However, the Modelica compiler we are using (Dymola [14]) has far more optimizations performed on the model equations prior to code generation compared to the modeling framework used in earlier work (ObjectMath [52]) 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 has become even more important for obtaining speedups, since the processor speed has increased much faster that the communication speed.

4.2.2 Scheduling Algorithms

The results, i.e the speedups achieved in [3] were not good enough, due to bad clustering 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 work, and in other work as well. Efficient algorithms with a low complexity should be used in order to fulfill 1. Also, task duplication has to be used to better exploit the sometimes low amount of parallelism that can be extracted from the simulation code at the subequation level, i.e. looking at expressions and statements in the generated C-code.

It is also 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.

4.2.3 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 5.3. The costs of some tasks can be determined with high accuracy, like for instance the cost of an arithmetic operation, or a function call to any standard math function. Other more advanced tasks, like for instance the task of solving a non-linear system of equations, can be harder to estimate accurately. The problem is to estimate such tasks in a convenient and general way, such 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 model of the parallel computer (i.e parallel computational model of communication and computation time) should be used, see Section 2.5. If the model is too simple and unrealistic the difference between estimated speedup and measured speedup will be too great. However, if the parallel model is too complicated the scheduling algorithm might increase in computational complexity since it has more parameters to consider.

4.3 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 it earlier was not possible to model and/or simulate a given problem. But with modern modeling techniques, such as object oriented modeling languages and advanced combined graphical and textual modeling tools, it is possible to model larger and more complex models than previously. 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 have longer execution time on their simulation runs.

Also, 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 their modeled systems. These users will probably have even less knowledge of parallel computing. This makes an *automatic* parallelization tool highly relevant if the tool is to be 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 is needed for increasing the performance of parallel programs. This is also further discussed in Chapter 9.

4.4 Scientific Method

The scientific method that is 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 is 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 are also 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 is collected from these executions. When comparing the parallel execution time with the execution time 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, 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, the hypothesis can be validated.

Chapter 5

An Automatic Parallelization Tool

This chapter presents a prototype implementation of an automatic parallelization tool. Preliminary versions of this tool have previously been described in [4, 5]. First an overview of the tool and its different parts is presented, followed by a description of the format of the simulation code passed as input to the tool. We also present the use of the parallel programming Message Passing Interface (MPI) [32] and other issues in the code generation and execution phases of the tool.

5.1 Overview

Figure 5.1 gives an overview of how the parallelization tool is used and 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 format of this code is presented in detail in section 5.2. 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.

The following sections presents the different parts of the tool in detail, except the scheduling algorithms which are presented in the next chapter.

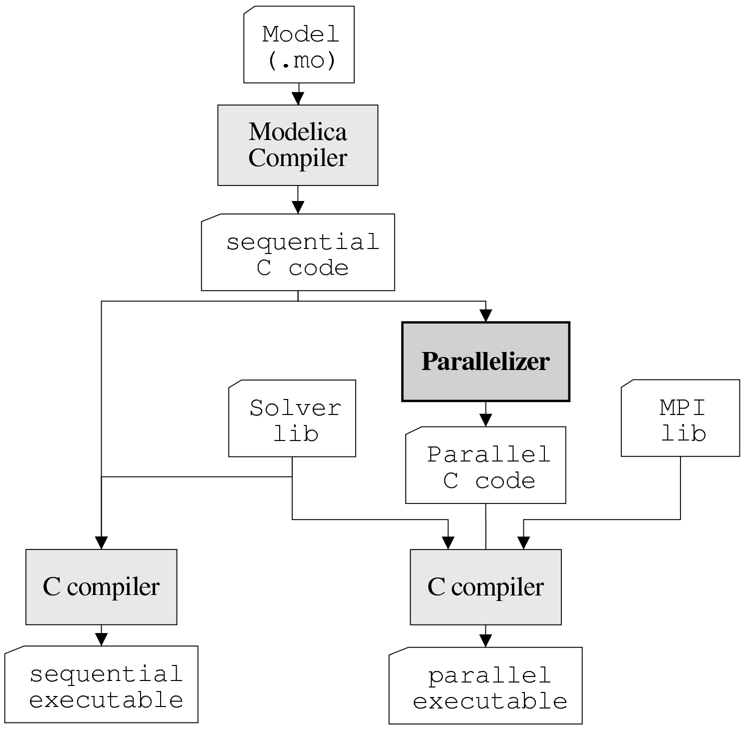


Figure 5.1. An overview of the automatic parallelization tool and its environment.

5.2 Input Format

The input to the tool is the sequential simulation code generated from a Modelica compiler, in our case Dymola [14]. The simulation code corresponds to the calculation of the right hand sides, i.e. the function f in Equation 2.1 or solving $\dot{X}$ from the equation system in Equation 2.2. 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 can also contain function calls, e.g. the standard math functions like `sin`, `cos` and `exp` are frequently used.

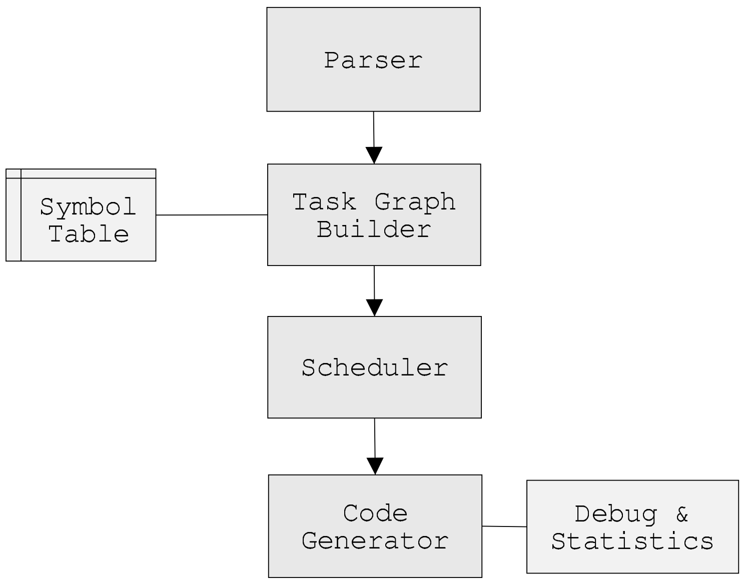


Figure 5.2. The internal architecture of the tool.

- **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.

For example, the generated C-code for the circuit example shown in Figure 2.4 is given in Appendix A.

The parallelization tool uses Bison [13] for generating a parser to parse the input. The lexical analysis is generated from a specification using the Flex tool [40]. The parser calls a set of functions for building the task graph, described in the next section.

5.3 Building Task Graphs

While parsing the input file, a fine grained task graph is built. For each arithmetic expression, function call, macro statement, etc. a task is created. A data dependency 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 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 simulation code for the `SmallODE` examples 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. 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 writes to variable `der_x`) is parsed, tasks 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 dependency 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.

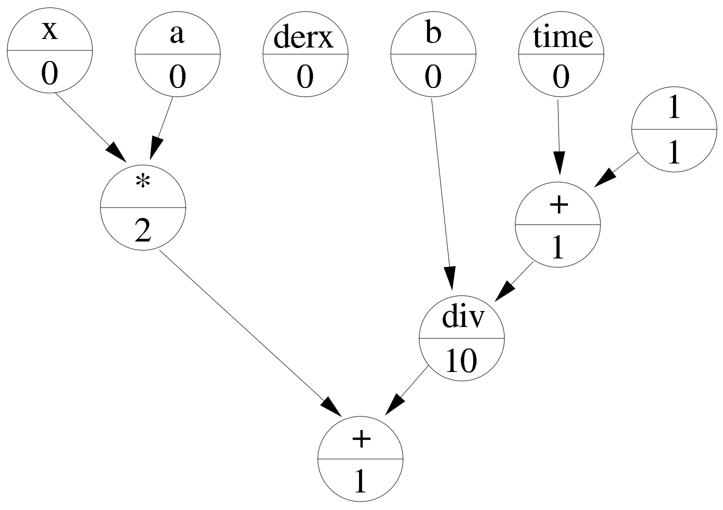


Figure 5.3. The task graph produced from the simulation code for the `SmallODE` example, on page 44.

5.3.1 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 statement. 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 and one continuous-time variable. A discrete-time variable only changes at certain points in time, at events, whereas continuous-time variables may change at any point in time. The `when` statement is a discrete event handling language construct in Modelica. 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. The first one sets the discrete variable `a` to the value 2.3, and the second one reinitializes the state variable `x` with a new value. The generated simulation code for the when equation has the following structure[1]:

```
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

[1]The when equation is split internally by Dymola and the reinit equation is treated elsewhere in the code.

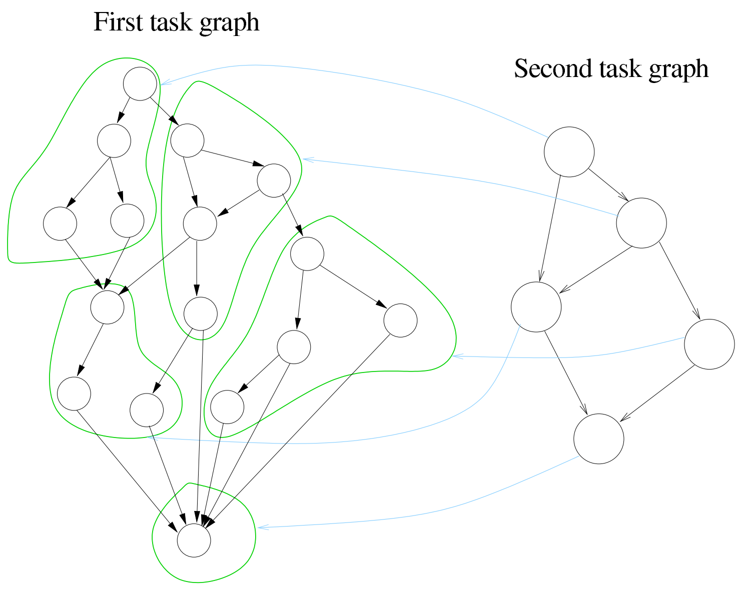


Figure 5.4. The two task graphs used in the tool.

is also 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 3.4, must have a low value. 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.

5.3.2 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.

```
...
{ /* 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`.

5.4 Cost Estimation

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.

5.4.1 Communication Cost

The communication costs can be measured by measuring the time it takes to send different 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 2.5.2. 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 5.6 gives the latency and bandwidth measured (by each specific vendor) for different multiprocessor architectures [2] [45, 36]. The values in Figure 5.6 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 5.6. Bandwidth and latency figures for a few different multiprocessor architectures.

5.4.2 Execution Cost

The execution cost of the tasks in the task graph can be either estimated given the architecture specifications of the targeted platform, or measured 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 sufficient for measuring high resolution time. The code in Figure 5.7 illustrates how it 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 5.7). This can be solved by turning off the optimizations responsible

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

for moving the code. However, this 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 5.7. Pentium assembler code for high resolution timing

There are 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 estimated 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 $f(10)$. If the standard LaPack [2] function for solving a linear system of equations is used (`xGESV`), the function described in Equation 5.1 can be used, where C_1 and C_2 are constants that can be determined by for instance profiling as above.

$$f(n) = C_1 \cdot n^3 + C_2 \tag{5.1}$$

5.5 Code Generation

When the scheduling algorithm has executed the code generation phase is started. The parallel code is generated in a master/slave fashion, where the master node runs the numerical solver, and each of the slave nodes executes a part of the calculation of the right hand side, as illustrated by Figure 5.8. The master processor first executes the numerical solver to calculate the next

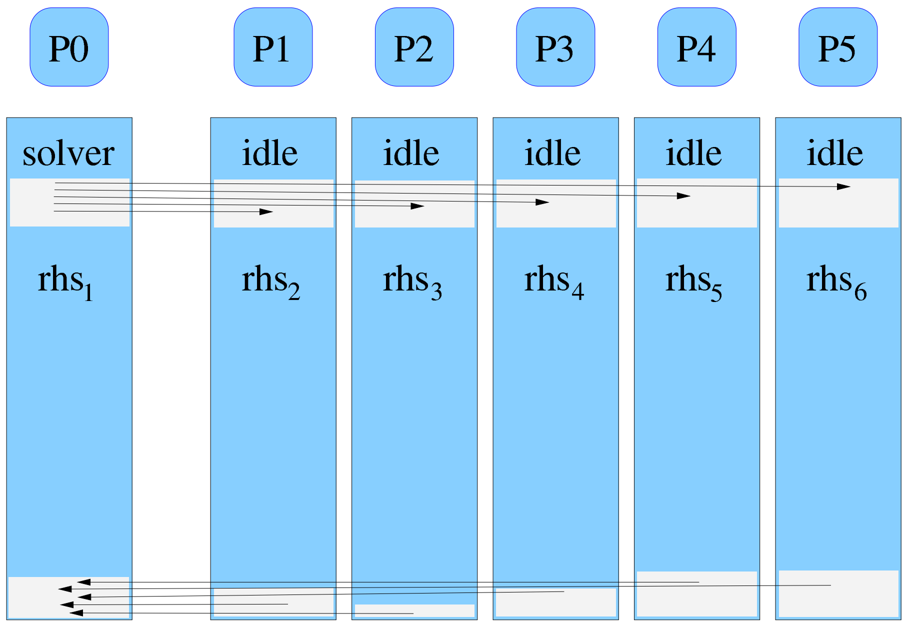


Figure 5.8. The parallel code executes in a Master/Slave fashion, with the numerical solver executing on processor 0.

values of the state variables. During this calculation the slave processors are idle, since they are waiting for the master processor to send the state variables and other variables, calculated by the numerical solver. When the numerical solver has finished its integration step, the state variables are sent to the slave processors. The slaves start their executions for calculating the right hand sides, followed by another message sending phase when data is sent back to the master processor. When the master processor has received all the data from the slaves the process restarts with another call to the numerical solver, for calculating the states at the next time step of the simulation.

The details of the code generation is described in the next section, followed by a section describing several of the optimizations that can be performed in the code generation phase.

5.5.1 Traversal of Task Graphs

From the second task graph, see Section 5.3.1, code is scheduled onto M processors by building a list of tasks, one list for each processor. By having a list of tasks for the processor assignment of nodes it is possible to duplicate a node onto several processors. The lists are first sorted in descending level order, such that the precedence constraints imposed by the data dependencies of the task graph are fulfilled (if not already performed by the scheduling algorithm). Then, code for each task is emitted, including code to write the result of the task to a temporary variable if needed. A typical case when temporaries are needed is when generating code for the `SmallODE` example, which task graph is shown in Figure 5.3. The generated code, when all tasks are assigned to the same processor, is:

```
tempr[1] = 1;
tempr[2] = tempr[1] + Time;
tempr[3] = divmacro(b,"b",tempr[2],"tempr[2]");
tempr[4] = a * x;
der_x = tempr[4] + tempr[3];
```

The reason for having temporary variables for expressions not assigned to a value is that the scheduling algorithm might choose to put sub-expressions within the same statement on different processors (i.e. assigning an expression to a variable). Then the intermediary result from one expression to the next needs to be stored in a variable. The `divmacro` macro call in the code above is an ordinary division, but including error handling (hence the extra string arguments) for preventing division by zero.

In our example above, the temporary variables are stored in an array. However it might be necessary to store the result in local variables instead. By storing the temporary variables in an array, the C compiler will with a large likelihood not be able to optimize the code as well as compared to the case where the temporary variables are stored in local variables. A C compiler normally only considers local scalar variables to be stored in registers, thus increasing performance by reducing the number of memory accesses. Our parallelization tool can generate code for both cases. However, the difference between the two approaches was in practice neglectable.

5.5.2 Optimizations

There are several optimization opportunities to consider in the code generation of parallel code using message passing.

- **Broadcast vs. individual message sending**
  For smaller examples data sizes of all variables that need to be sent to the slaves are relatively small. Then it might be cheaper to broadcast the complete data structure to all processors instead of dividing the data into parts and send only the data required by each individual processor.

The reason for this is that for small data sizes the network latency is the dominant factor of the communication cost. Therefore it does not matter if an additional amount of data is sent. Also, by not dividing the data into parts for each processor the cost of performing the partitioning of the data itself is removed.

By using a simple model of the network and an additional cost for the splitting of data arrays, a payoff heuristic can be implemented that can choose between the two different approaches.

- **Collecting messages**
  If the scheduling algorithm itself does not perform a proper clustering of tasks it can be worthwhile implementing an optimization to collect the send primitives of n outgoing messages from one processor P_i to another processor P_j into a single send primitive. By collecting n send primitives the communication cost is reduced from $n \cdot (L + x)$ to $L + n \cdot x$, where L is the latency and x is the cost proportional to the data size of each send primitive. The same optimization can also be performed on the receive primitives in the code generated for processor P_j.

  This approach is somewhat ad-hoc since it tries to optimize a poor schedule/clustering of a parallel program. The correct way to solve this problem should instead be to cluster (or merge) the nodes prior to scheduling. Therefore, this optimization is not yet implemented in our parallelization tool. Instead, effort has been put on developing clustering and merging algorithms that can solve this problem.

- **Remove resends of data**
  For some schedules a particular data item could already have been sent from a processor P_i to another processor P_j. Thus it does not need to be sent a second time. Figure 5.9 illustrates this typical scenario: the `a` node allocated to processor P_i sends data to both `d` and `e`, both allocated to another processor P_j. But since `d` and `e` are both assigned to the same processor, the data sent from `a` to both of the nodes are the same, the second message sent (between `a` and `e`) is not needed.

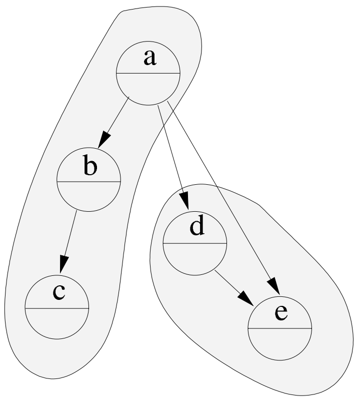


Figure 5.9. A task graph with processor assignment showing multiple sends of the same data to different nodes on the same processor.

Chapter 6

Scheduling and Clustering Algorithms

This chapter presents our contributions in the area of scheduling and clustering algorithms. The contributions consist of improvements of earlier algorithms and insights in how well suited some scheduling and clustering algorithms found in literature are in the area of scheduling automatically generated simulation code from optimized model equations.

First, we emphasize the requirements put on scheduling and clustering algorithms for our application area. This is followed by a presentation of the algorithms, or improvements on algorithms found in literature, contributed in this work. Finally, an experimental task graph scheduling environment developed in the *Mathematica* programming environment is presented, along with some standard scheduling algorithms that also have been implemented in that framework.

6.1 Requirements

The major requirement imposed by the task graphs from our parallelization tool is the granularity of the graph. Since the parallelization tool builds the task graph at the lowest level, building small tasks from each sub-expression, the task graph granularity (communication to computation ratio as defined in Equation 3.4), g, becomes high. Typical values for g lies between 100 and 1000, depending on the communication network of the parallel computer and the processor architecture in general.

The choice of building task graphs at the expression level also affects the number of tasks produced from the simulation code. Hence, a second important requirement is that the computational complexity of the task scheduling and clustering algorithms must be low. For instance, consider a scheduling algorithm of complexity $O(n^3)$, where each iteration takes about 1000 cycles. For a task graph with 10000 tasks the algorithm would take $\frac{(10^4)^3 \cdot 1000}{10^9} 10^6$ seconds $\approx$ 11 days[1]. This example shows that it is very important to keep the algorithm complexity low with respect to the task graph size.

The simulation code generated from the Modelica compiler that we use in this work (Dymola [14]) is highly optimized, thus giving many dependencies (i.e. edges) in the task graph. The large number of data dependencies, combined with the high granularity value of the task graph, requires the use of task duplication techniques in order to fully take advantage of the amount of parallelism in the task graph. This requirement is in conflict with the requirement on low computational complexity. By allowing task duplication to decrease the parallel execution time, the complexity of the algorithm increases. Hence, it is important to find a balance between the complexity added by introducing task duplication and the total complexity of the scheduling algorithm.

6.2 The TDS algorithm

The TDS algorithm is a linear clustering algorithm with task duplication [12], which produces an optimal schedule given some constraints on the task graph. It has a low complexity ($O(n^2)$) in comparison with many other scheduling algorithms, which makes the algorithm attractive for our purpose. However, there are several problems to consider when using the TDS algorithm.

Since the TDS algorithm is a linear clustering algorithm, it can not guarantee a fixed number of processors, i.e. the processor requirement is unlimited. Therefore, for having a fixed number of processors a second algorithm has to be run to reduce the number of processors to the desired value.

A second problem is the task graph constraint that needs to be fulfilled for the TDS algorithm to produce the optimal schedule. For fine grained task graphs, i.e. task graphs with a high granularity value, the optimality constraint is not fulfilled. Thus, the TDS algorithm does not produce the optimal schedule for our application area. However, the task duplication feature of TDS combined with the low complexity still makes it interesting for further investigation.

A third problem is the linear clustering approach of the TDS algorithm. Since the TDS algorithm uses linear clustering it will never put two siblings on the same processor. However, for our fine-grained task graphs, such a restriction can give poor performance. For instance, consider the small task graph in

[1]The processor speed is assumed to be 1GHz

Figure 6.1. Since the communication cost is much greater than the execution cost, the approach of only extracting linear clusters and assign them to processors is far from optimal. Instead, a non-linear clustering approach is far more effective. The parallel execution time for the non-linear clustering with task duplication, shown in Figure 6.2(a) is 6, whereas the parallel execution time in the linear clustering with task duplication is 24, as shown in Figure 6.2(b).

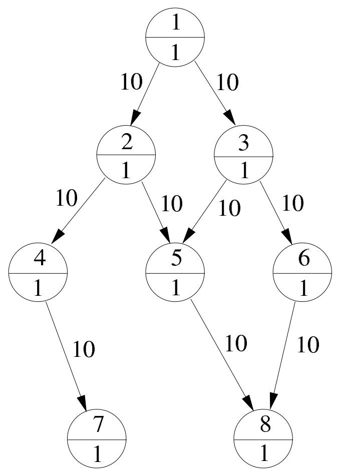


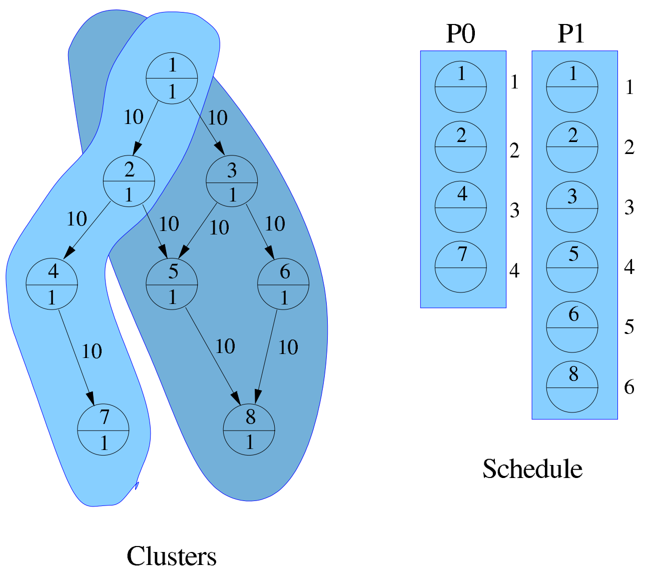
Figure 6.1. A small DAG (Directed Acyclic Graph) with high granularity value.

We have implemented the TDS algorithm in our parallelization tool [30][2], as well as the second phase described above to limit the number of processors. The second phase uses a combination of a load balancing strategy and a minimization of communication sending strategy when choosing clusters to merge tasks for limiting the number of processors.

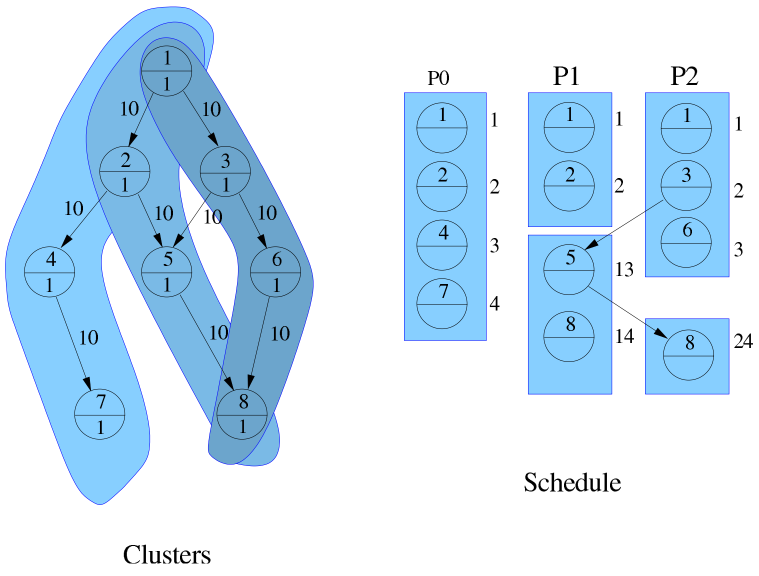
6.3 A Pre-Clustering approach

The high granularity (defined in Equation 3.4) values of the generated task graph is a problem when using an algorithm such as the TDS algorithm. Therefore, a pre-clustering approach was also considered. The pre-clustering approach merges tasks together, thus increasing the granularity of the task graphs. Another positive side effect of merging tasks together is that the message sizes increase.

[2]The TDS algorithm was initially implemented by Magnus Gustavsson, PELAB, Linköping University, Sweden



(a) A non-linear clustering with task duplication, parallel execution time = 6.



(b) A linear clustering with task duplication, parallel execution time = 24.

Figure 6.2. The problem of using linear clustering on fine grained task graph.

Our pre-clustering algorithm builds on earlier work of merging tasks found in [44]. The basic idea is to repeatedly add tasks to a set of tasks that will be merged into a larger task by considering related tasks in a specific order. Addition of new tasks is terminated when the sum of the execution costs of the tasks in the set reaches a certain threshold. When adding a new task to the set a cycle can occur in the resulting task graph. This is illustrated in Figure 6.3 where adding a task b to the cluster consisting of a will cause a cycle in the final task graph. This means that the resulting task graph does not fulfill the properties of a DAG (Directed Acyclic Graph), thus many scheduling algorithms can not be applied to the task graph. We have added prevention of cycles into the algorithm, which is presented in Figure 6.4.

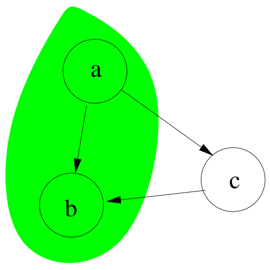


Figure 6.3. A cluster that forms a cycle in the resulting task graph when putting node a and node b into the same cluster.

The algorithm `buildCluster` presented in Figure 6.4 takes a starting node as a parameter and a list of nodes that are candidates for inclusion into the cluster. The first candidates that are considered for addition to the cluster are children of the starting node. The next candidates are children to already added nodes that only have one predecessor. Then siblings to the starting node, i.e. nodes with a common parent, are considered for inclusion into the cluster. Finally, any node in the list of candidates is considered.

6.4 The Full Task Duplication Method

For task graphs with high granularity numbers, in the range $100 - 1000$, a simple method is to perform Full Task Duplication(FTD). This means that we build clusters 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 6.5.

```
algorithm buildCluster(n:node,l:list of nodes)
    cluster:=addNode(n)
    t:=calcSize(cluster)
    while $t < size$ do
        if $c \in Child(n)$ and $c \notin cluster$ then
            selectedNode:=c
            break
        end if
        if $c \in cluster$ and $Length(Child(c)) = 1$ and $Child(c) \notin cluster$ then
            selectedNode:=Child(c)
            break
        end if
        if $c \in Sibling(n)$ and $c \notin cluster$ then
            selectedNode:=c
            break
        end if
        if $c \notin cluster$ and $c \in l$ then
            selectedNode:=c
        end if
        cluster:=removeCycles(cluster)
        cluster:=addNode(selectedNode)
        l:=removeNode(l,selectedNode)
        t:=calcSize(cluster)
    end while
```

Figure 6.4. Algorithm for merging tasks to increase granularity, used in the pre-clustering approach described in Section 6.3.

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 among all clusters are determined. This value is a measure on how much speedup can be achieved by parallelizing the code using the FTD approach.
- Secondly, tasks are merged in a load balancing manner by repeatedly 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, tasks are merged until the processor requirement is met by merging the two tasks with the largest number of common nodes. This approach is greedy, since it will always minimize the maximum cluster size.

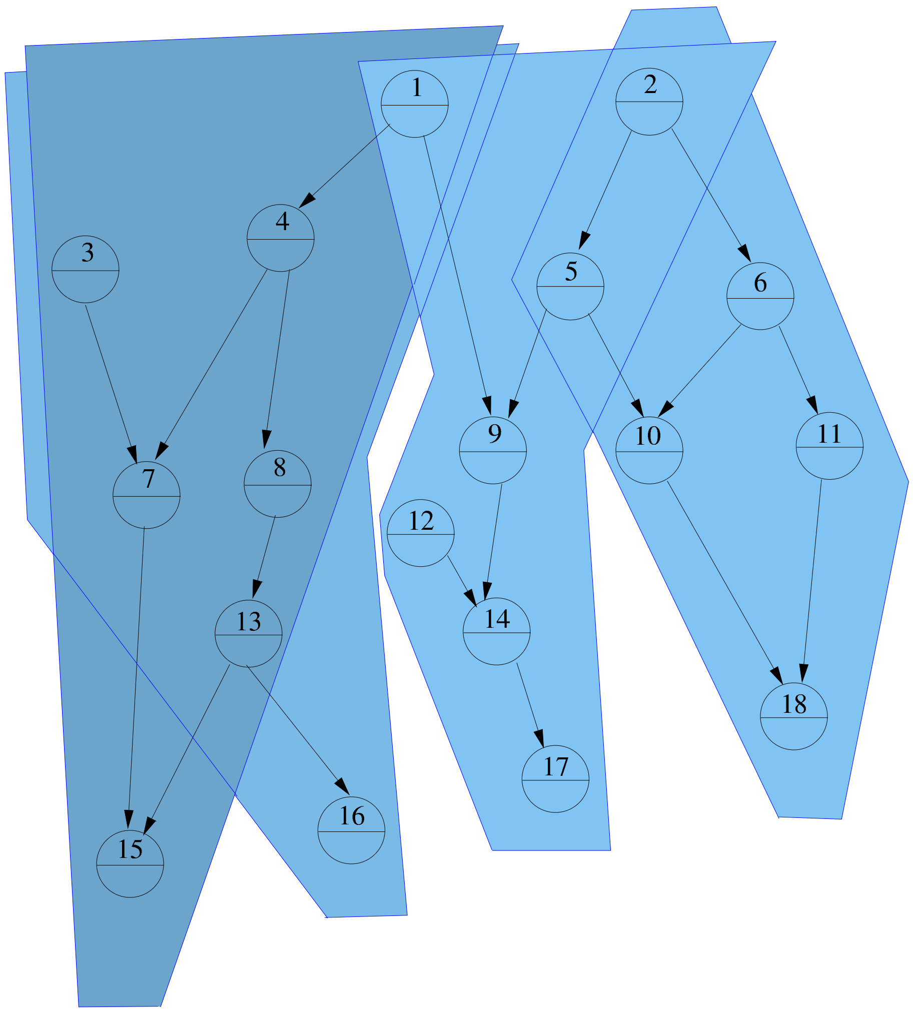


Figure 6.5. Applying full duplication to a task graph.

Figure 6.6 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 a list data structure where each element is a set of nodes. The input to the algorithm is a task graph as defined earlier in 2.5, and the number of processors. The algorithm returns a list of sets, where each element in a set contains the nodes to be executed on one processor.

Since the FTD algorithm duplicates all necessary tasks, 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

```
algorithm FTD(G: Graph $(V, E, c, \tau)$, N:Integer)
    $cl, cl2$ : $list\ of\ S \subseteq V$
    $maxSize$ : $Integer$

    cl:=emptyList()
    cl2:=emptyList()
    0. $\forall n \in V \mid pred(n) := \emptyset$   $do$
        cl:=addElt({n, predecessors(n)},cl)
    1. $maxSize := max_{\forall S \in cl}(\sum_{n \in S} \tau(n))$
    2. cl2 = addElt( $(cl(1) \cup \ldots \cup cl(i)), cl2) \mid \sum_{n \in cl(1) \cup \ldots \cup cl(i)} \tau(n) <$ maxSize
    while $length(cl2) > N$ do
        3.find $S_1, S_2 \in cl2 \mid \sum_{n \in S_1 \cup S_2} \tau(n) = min(\sum_{v \in S_i \cup S_j} \tau(v))$
                $\forall i, j \in \{1 \ldots, length(cl2)\}$
        $cl2 := delElt(S_1, cl2)$
        $cl2 := delElt(S_2, cl2)$
        $cl2 := delElt(S_1 \cup S_2, cl2)$
    end while
```

Figure 6.6. An algorithmic description of the FTD Method.

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, as discussed in section 5.5.2. 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 6.1. Here cl_{max} is the maximum execution 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.

$$C_p = cl_{max} + 2 * (L + n * B) \tag{6.1}$$

6.5 A Scheduling Framework in Mathematica

To be able to experiment with new scheduling and clustering algorithms in an easy, interactive and powerful way, a task scheduling framework for developing scheduling algorithms in the computer algebra system *Mathematica* has been developed. *Mathematica* is also a powerful functional programming language, which enables us to do fast implementations of prototype algorithms and still keep the algorithms readable and easy to understand.

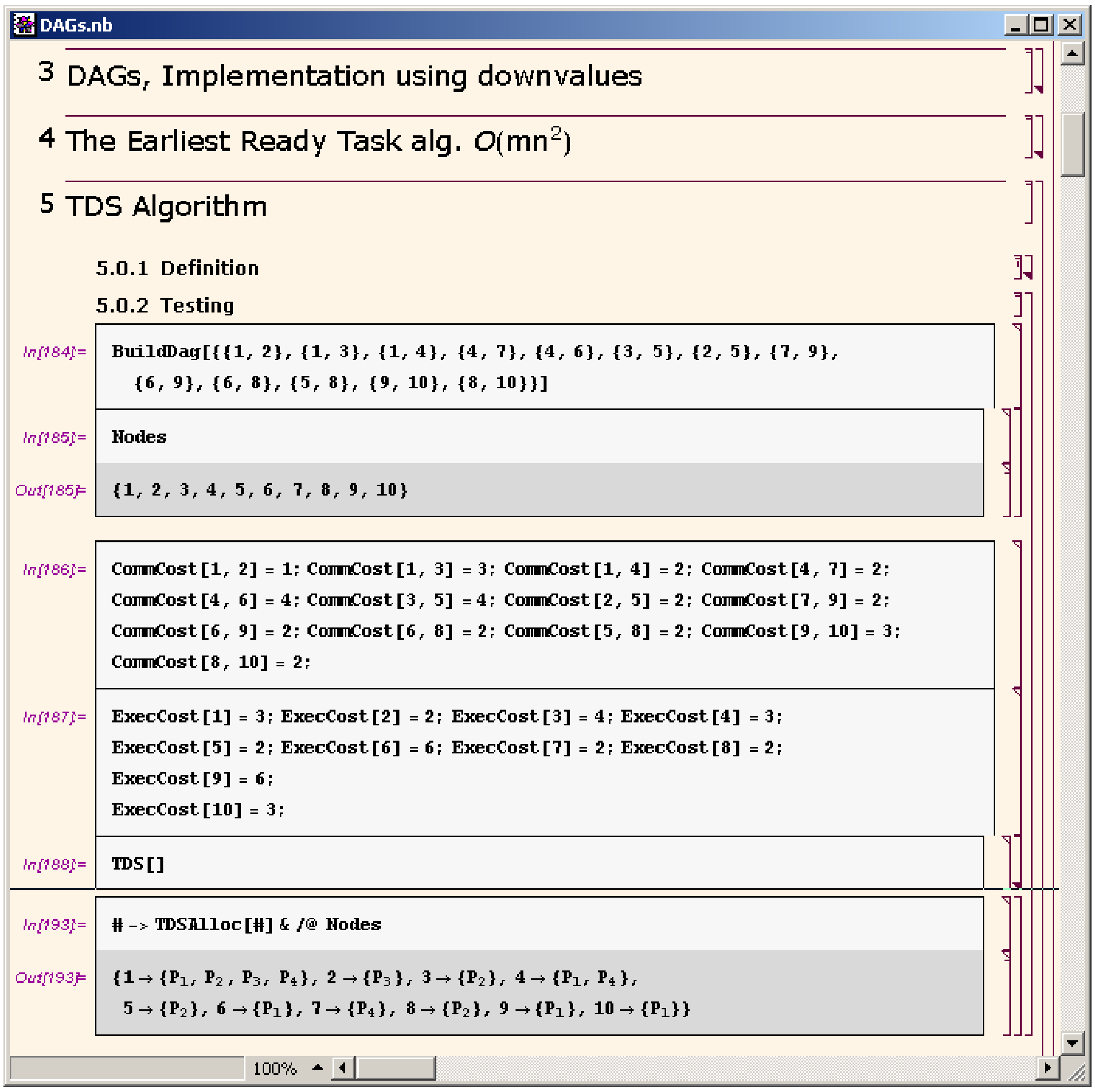
The interactive programming environment that the *Mathematica* tool provides also increases productivity and quality of the algorithms developed, since it enables the user to work in an interactive way, experimenting with different ideas, different task graphs, several parameter settings, and so on.

The scheduling framework consists of a set of functions gathered in a Mathematica *package*. There are functions for building task graphs, setting execution costs and communication costs. Each task in a task graph is uniquely represented by an integer value. The task graph is built from a list of edges, where an edge is a list of two integer values identifying the edge between the two tasks represented by those values. The direction of the edge is indicated by the position of the task values in the list; the first position is the source task and the second position is the target task of the edge.

There is also a function for generating graphical information from the task graph, making it possible to view the task graph in a graph tool, Vcg [51] or Aisee [1]. This ability is important for getting a good understanding on the structure of the task graphs and on how clustering and scheduling algorithms behave. The graph viewing tools also have some support for graph statistics and other helpful means for better understanding the task graph structure and how to attack the scheduling and clustering problem.

Figure 6.7 shows a Mathematica notebook with an example on how a task graph is built and how it can be scheduled using the TDS algorithm.

The package also contains two additional implementations of known scheduling algorithms. The ERT algorithm [26] is a list scheduling technique, earlier described in Section 3.1.2, which is easily implemented in the framework. The DSC algorithm [55] is also implemented, however without considering using the most optimal techniques, like for instance priority queues. Instead, a more straightforward implementation has been done, with somewhat higher complexity in some parts of the algorithm.



3 DAGs, Implementation using downvalues

4 The Earliest Ready Task alg. $O(mn^2)$

5 TDS Algorithm

5.0.1 Definition

5.0.2 Testing

```
In[184]:= BuildDag[{{1, 2}, {1, 3}, {1, 4}, {4, 7}, {4, 6}, {3, 5}, {2, 5}, {7, 9},
             {6, 9}, {6, 8}, {5, 8}, {9, 10}, {8, 10}}]

In[185]:= Nodes

Out[185]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

In[186]:= CommCost[1, 2] = 1; CommCost[1, 3] = 3; CommCost[1, 4] = 2; CommCost[4, 7] = 2;
          CommCost[4, 6] = 4; CommCost[3, 5] = 4; CommCost[2, 5] = 2; CommCost[7, 9] = 2;
          CommCost[6, 9] = 2; CommCost[6, 8] = 2; CommCost[5, 8] = 2; CommCost[9, 10] = 3;
          CommCost[8, 10] = 2;

In[187]:= ExecCost[1] = 3; ExecCost[2] = 2; ExecCost[3] = 4; ExecCost[4] = 3;
          ExecCost[5] = 2; ExecCost[6] = 6; ExecCost[7] = 2; ExecCost[8] = 2;
          ExecCost[9] = 6;
          ExecCost[10] = 3;

In[188]:= TDS[]

In[193]:= # -> TDSAlloc[#] & /@ Nodes

Out[193]= {1 → {P1, P2, P3, P4}, 2 → {P3}, 3 → {P2}, 4 → {P1, P4},
           5 → {P2}, 6 → {P1}, 7 → {P4}, 8 → {P2}, 9 → {P1}, 10 → {P1}}
```

Figure 6.7. A Mathematica notebook containing example pieces of the scheduling framework.

Chapter 7

Results

This chapter presents the main results achieved so far in this work. The results consist of computed speedups based on the delay model and the task graphs generated from the simulation code. Additionally, measured speedup from executing the simulation code on a parallel computer is given. The results also include an evaluation of several scheduling and partitioning algorithms for scheduling of automatically generated scheduling code derived from the optimized system of equations.

The different scheduling and clustering approaches are illustrated by a small, but still realistic task graph built from the simulation code of a small Modelica model. The model is the `PreLoad` mechanical model from the Modelica Standard Library, see Figure 7.2 below. Appendix A contains the Modelica source code for the `PreLoad` example and Appendix A and C contains the sequential and parallel C code for the same model. The generated task graph is depicted in Figure 7.2. The start node and the end node have been removed to obtain a nicer graphical layout of the task graph.

Moreover, larger examples from the `ModelicaAdditions` library and models from the `Thermofluid` package are used. Table 7.1 contains a list of the used models.

7.1 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 task graph generated in our tool. As an example, we will use the small task graph shown in Figure 7.2. For that task graph, the TDS algorithm produces the values found in Figure 7.3. The table shows the

Model	Size (# equations)
`ModelicaAdditions.MultiBody.Examples.Robots.r3`	6071
`ThermoFluid.Examples.PressureWaveDemo(n=50)`	4725
`ThermoFluid.Examples.PressureWaveDemo(n=100)`	9175
`ThermoFluid.Examples.PressureWaveDemo(n=150)`	13625
`Modelica.Mechanics.Translational.Examples.PreLoad`	84

Figure 7.1. The Modelica models used in this thesis as application examples for automatic parallelization.

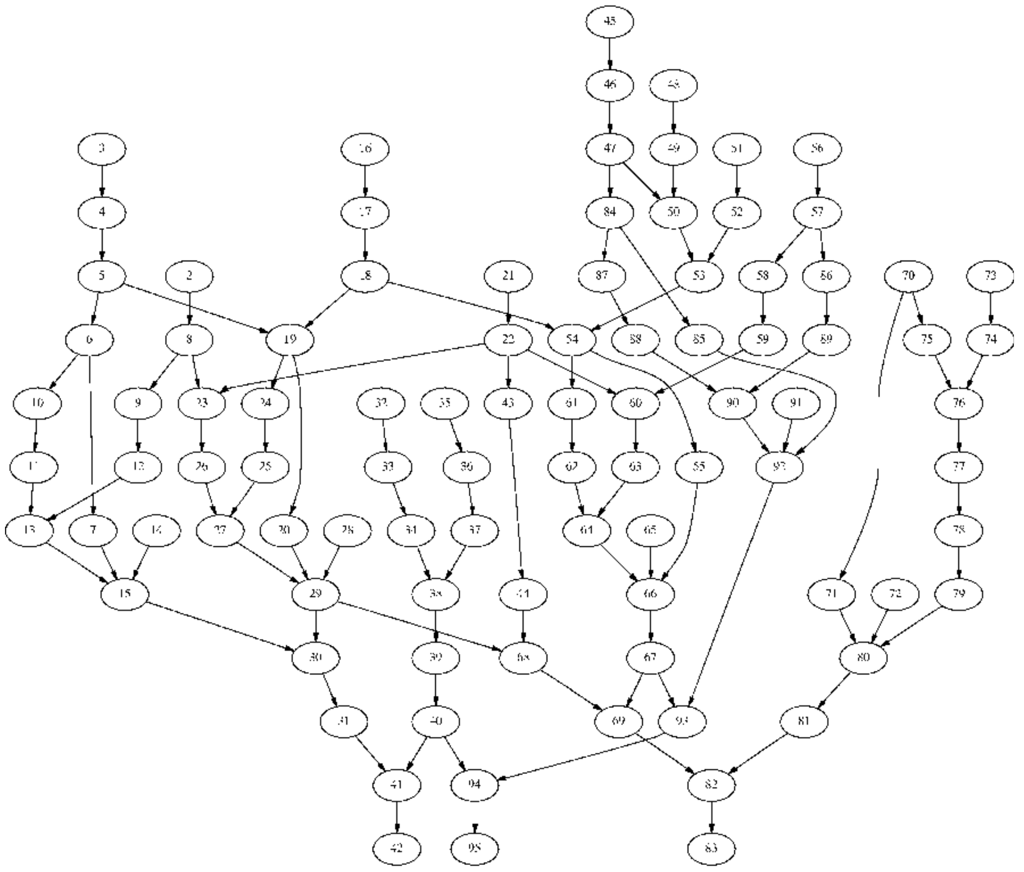


Figure 7.2. The task graph built from the code produced from the `PreLoad` example in the mechanics part of Modelica Standard Library.

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 required processors are 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.

Total sequential execution time	Number of nodes
100	221

(a) Graph size and total sequential execution cost.

c	1000	500	100	10	1
ect	5008	2508	508	58	16

(b) Parallel computation time, i.e. the ect value of the exit node of the DAG, for different values of node-to-node communication cost c.

Figure 7.3. Results for the TDS algorithm on the `PreLoad` example with varying communication cost.

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 2.2. The result for that example is shown in Figure 7.4, 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.

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, i.e. the ect value of the exit node of the DAG, for different values of node-to-node communication cost c.

Figure 7.4. Results of the TDS algorithm on the robot example, using mixed mode and inline integration with varying communication cost.

7.2 Results From the Pre-Clustering Approach

The pre-clustering approach, described in Section 6.3, did not produce a satisfactory result. Figure 7.5 shows the resulting task graph after the pre-clustering phase has been run. Due to the many dependencies between tasks in the original task graph, an efficient pre-clustering could not be achieved by using the algorithm presented in Figure 6.4. Any attempt of using a scheduling algorithm on the task graph in Figure 7.5 will result in a sequential schedule.

One reason for the poor performance of the pre-clustering algorithm is that the original task graph has such high granularity that many nodes need to be clustered together before a merge can be performed. This phenomena, combined with the non-duplication scheme in the pre-clustering algorithm introduce many dependencies in the resulting task graph. An alternative viewpoint is that the pre-clustering algorithm does not succeed in limiting the amount of parallelism in such a manner that a sufficient amount of parallelism is left in the resulting task graph.

Another reason for the poor performance are cyclic dependencies in the resulting task graph. When a cycle is detected as a consequence of adding a node, the whole cycle path is included (or excluded) in the cluster. This substantially reduces the amount of parallelism in the task graph if the number of nodes belonging to the cycle path is large.

A conclusion that can be drawn from the above problem is that task duplication schemes have to be employed already at the pre-clustering phase, *if* a multi phased scheduling approach is to be used. However, this is in conflict with the low complexity requirement discussed in Section 6.1, since allowing task duplication in the pre-clustering phase will increase the complexity of the scheduling problem.

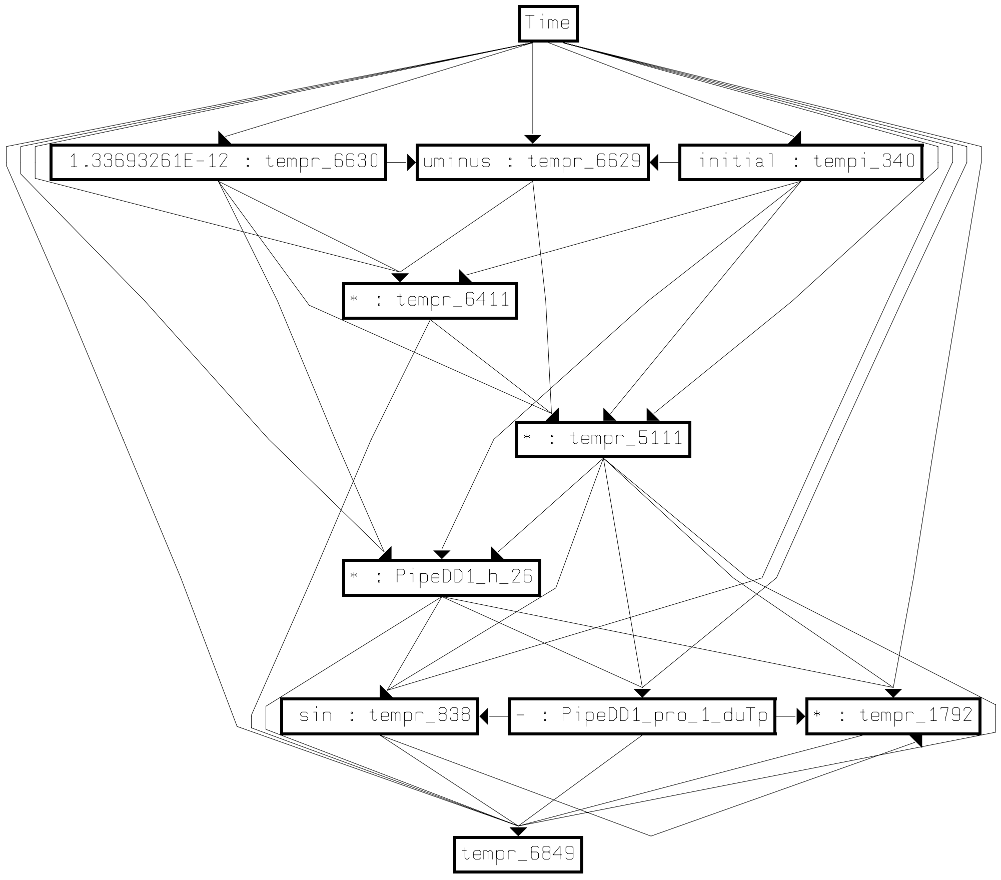


Figure 7.5. The resulting task for simulation code from a thermofluid pipe when pre-clustering is performed.

7.3 Results From the FTD Method

Figure 7.6 gives some computed theoretical speedup figures using the Full Task Duplication Method for a discretized thermofluid pipe. Figure 7.8 contains the 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 6.1. 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.

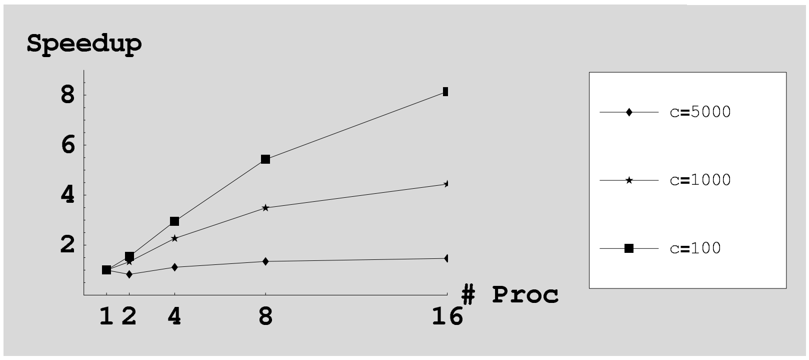
The parallel simulation code from the discretized thermofluid pipe has also been executed on a PC-cluster with a SCI network as the communication device. Figure 7.7 gives the measured speedup when executing on the PC-cluster. The measurements on execution time differ from the computed theoretical speedup figures given in Figure 7.6 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.

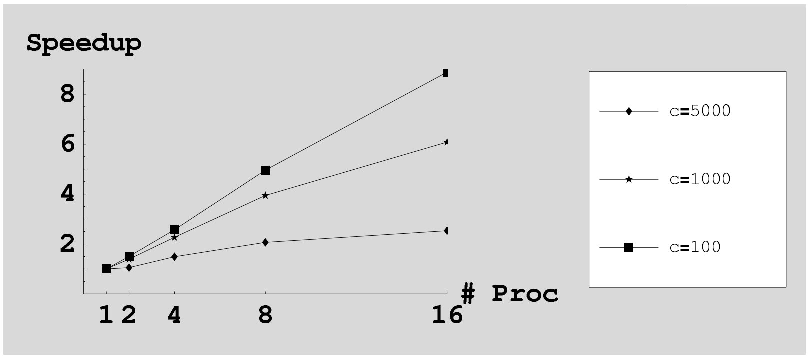
Second, all curves have a tendency of degraded speedup as the number of processors increase. The figures shows a degradation after about 8 processors. This effect is due to the parallel communication and computation model used in this work, the delay model described in Section 2.5.4. The delay model does not cover all costs of communication, e.g. the gap cost (see Section 2.5.2) 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 7.8. When using 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 almost none.

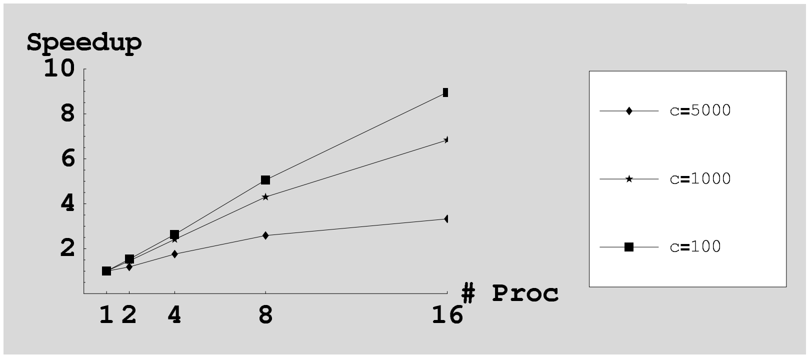
Since the robot example is the most realistic example among the examples studied in this thesis, it substantially influences the interpretation of the results. Therefore, a preliminary conclusion that can be drawn is that the FTD method works well for some nice structured examples such as discretized flow models but is less suited for general large and complex models. However, some uncertainty still remains since larger models than the robot example have not been tried yet.



(a) Thermofluid pipe with 50 discretization points.

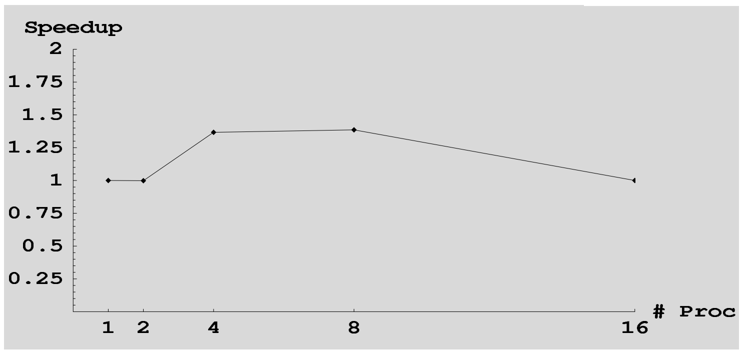


(b) Thermofluid pipe with 100 discretization points.

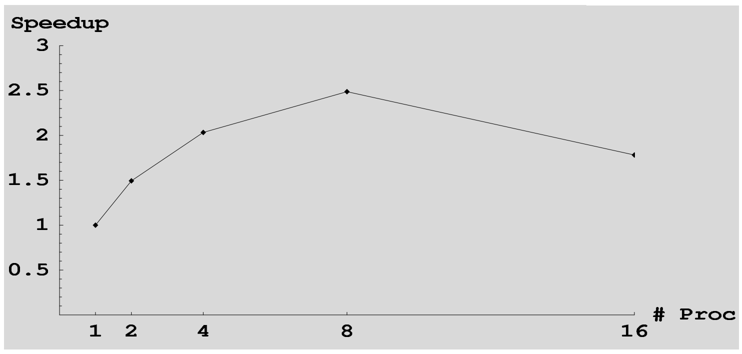


(c) Thermofluid pipe with 150 discretization points.

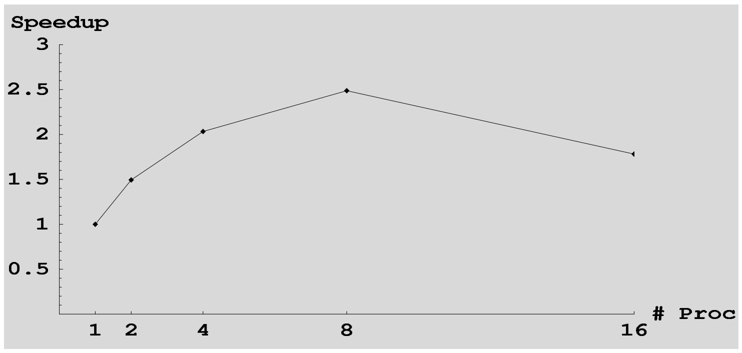
Figure 7.6. Computed speedup figures for different communication costs c using the FTD method on the Thermofluid pipe model.



(a) Thermofluid pipe with 50 discretization points.

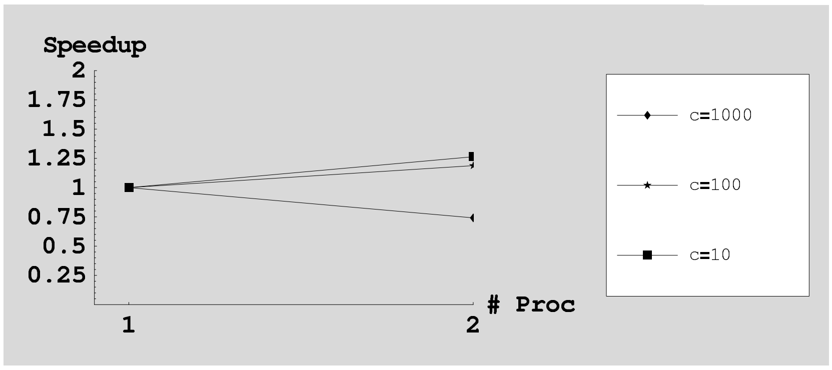


(b) Thermofluid pipe with 100 discretization points.

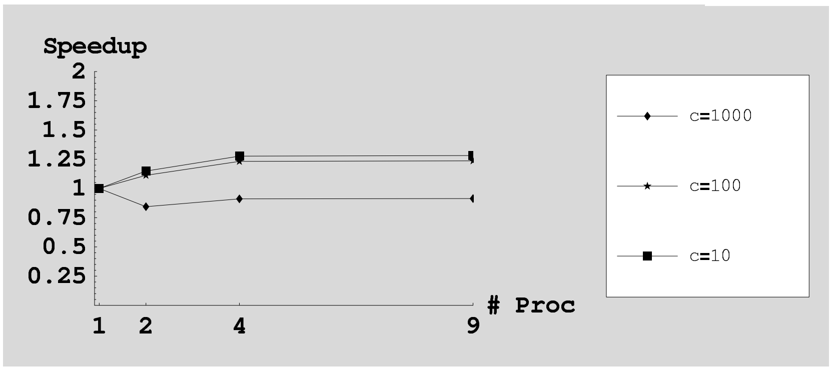


(c) Thermofluid pipe with 150 discretization points.

Figure 7.7. Measured speedup figures when executing on a PC-cluster with SCI network interface using the FTD method on the Thermofluid pipe model.



(a) Mechanical robot model with a standard solver



(b) Mechanical robot model with mixed mode and inline integration

Figure 7.8. Computed speedup figures for different communication costs, c, using the FTD method on the robot example.

7.4 The Scheduling Framework in Mathematica

Several scheduling algorithms have also been implemented in the prototype scheduling framework written in Mathematica, see Section 6.5. Our parallelization tool also have a rudimentary export functionality for transferring task graphs from real simulation code files into the Mathematica framework.

The DSC clustering algorithm has been implemented in this framework and Figure 7.9 gives some results for the parallel time produced by the DSC clustering. In comparison with the TDS algorithm, the DSC algorithm performs better. This is because the DSC algorithm is a non-linear clustering algorithm, which makes it better suited for fine grained task graphs.

c	1000	500	100	10	1
Preload (PT)	2013	213	33	16	14
Robot (PT)	30321	4221	1611	-	-

Figure 7.9. Computed results from the DSC clustering with different communication cost. The sequential costs (normalized) are 221 and 8369 for PreLoad and Robot models respectively.

Chapter 8

Conclusions

This chapter presents the conclusions that can be drawn so far in this work. The conclusions emphasize the scheduling and clustering algorithms used.

8.1 Scheduling Approach

The traditional scheduling approach of solving the complete scheduling problem in one step does not work well for the task graphs built from simulation code that can be produced by our tool. In fact, there exist no scheduling algorithm in the literature with support for task duplication that schedules fine grained task graphs well for a fixed number of processors in a single step algorithm. And even if one such algorithm should be invented, the time complexity of such an algorithm would probably be too large for practical usage.

By using the task clustering approach the scheduling problem is handled in two stages. The first stage clusters tasks together and the second stage schedules the clusters onto a fixed number of processors. The advantage of this approach is that each phase gets a lower complexity, thus reducing the overall complexity of the scheduling problem.

The conclusion regarding what scheduling approach to take is that a multistep approach to the scheduling problem with task duplication for a fixed number of processors is the most suitable for this problem.

8.2 Task Graph Granularity

The task graph granularity, i.e. communication to computation ratio, is a crucial factor when choosing which scheduling or clustering algorithm to use. In our case the task graph granularity value is high, which means that the execution costs are small in comparison to the communication costs. This implies that an efficient solution to the parallelization problem must involve task duplication and task clustering algorithms in order to reduce the communication overhead.

An alternative solution is to reduce the granularity of the task graph by using task merging algorithms, such that traditional scheduling algorithms, like for instance the TDS algorithm, work better. The results from our task merging algorithm resulted in a task graph with little or no parallelism, thus indicating that in order for this approach to be successful a task duplication based merging algorithm should be used. However, no such algorithm based on task duplication has been found in the literature.

Finally, results from the DSC algorithm shows that the clustering approach, where tasks are first clustered for an unlimited number of processors and thereafter scheduled on a fixed number of processors, did not give a satisfying result.

One conclusion regarding the task graph granularity is that a satisfying scheduling or clustering algorithm that handles task graphs with large granularity values has not yet been found. Either a new task merging algorithm must be developed that decreases the task graph granularity such that any standard scheduling algorithms can be used, or new scheduling algorithms with low time complexity and task duplication haves to be developed in order to produce successful results.

8.3 Cost Estimation

So far, we have only used simple estimations of the execution costs of the tasks. For instance, the solution of an equation system is always given an execution cost of 1000. This is a major simplification, and better cost estimations have to be implemented in the parallelization tool. However, for the early development of good scheduling and clustering algorithms the detailed execution cost is of less importance. Further on in this work, a better cost estimation heuristic has to be designed and implemented in order to tweak the algorithms for maximum performance.

Chapter 9

Future Work

This chapter introduces issues that have not been solved, or which have not been solved efficiently enough with good results. The future work is divided into subareas, each discussed in the sections that follow.

9.1 Scheduling and Clustering Algorithms

So far in this work, we have not found an efficient scheduling or clustering algorithm that works well enough on the fine grained task graphs produced from simulation code generated for equation based simulation languages. The attempts tried in this work give varying results, where the FTD approach works reasonably well for some examples. However, we still have not tried very large examples that would be impossible to execute on a single processor machine. For such examples, it might be necessary to adapt the FTD algorithm even further, by limiting the complete duplication by for instance some heuristic. This implies that a scheduling phase needs to be used after the FTD clustering.

Another unexplored possibility is to build the task graphs from a different level of granularity, for instance at the equation level, or even higher at the block level. This could be necessary for parallelizing really large models, with for instance more than one million equations and variables. Such systems would generate too many tasks if the task graph were built at the expression level. The drawback for building the task graph at a higher level is loss of possible parallelism.

There are large research opportunities for developing efficient task duplication based clustering algorithms or task merging algorithms with duplication. Such algorithms are needed to fully take advantage of all possible parallelism in the fine grained task graphs produced by our tool.

When efficient clustering and scheduling algorithms have been developed and implemented, there are other ways of improving the performance. The simple delay model used in most scheduling algorithms is often not precise enough, yielding too large errors. Thus, a more realistic parallel computational model can be introduced. This will affect the scheduling and clustering algorithms, giving additional variables to consider, which will increase the complexity of the algorithms. However, with a more realistic model the performance could increase, resulting in even more speedup.

9.2 Exploiting Parallelism

An orthogonal research area is to investigate how to further increase the amount of parallelism in general for differential and algebraic equation systems. For instance, there are several optimizations performed on the equation system prior to sequential code generation. One possibility could be to adapt those optimizations to produce more parallelism. One such example is the use of mixed mode integration combined with inline integration, which reveals more parallelism in the simulation code, see Section 2.2.

Another area for future research is to adapt parallel solvers for solving the different subsystems of equations that can be found in the simulation code. For instance, solving a linear system of equations can be parallelized by using the functionality provided by the Scalapack package [7]. In the examples studied in this thesis the size of the subsystems of equations, linear or non-linear, are typically in the range of one to ten equations. Hence, a parallelization does not produce much speedup, or even no speedup at all. However for particular applications, like for instance large discretized models, larger subsystems of equations might be found. For those examples it might be fruitful to parallelize the solution of such systems of equations. This will also affect the scheduling algorithm since the parallelization of the equation systems will need to be integrated into the schedule. This will further complicate the scheduling problem. The easiest approach is to allocate a set of slaves dedicated only to solving a part of the subsystem. But to fully use all processors available, research has to be made on how to integrate such data-parallel parts into the task scheduling algorithm.

9.3 Data Parallelism in Modelica

The current version of the Modelica language has no support for data parallelism, apart from arrays and loops being present in the language. In order to fully support scientific computing using Modelica, and thereby increase the use of the Modelica language as a simulation language with the capabilities of performing advanced scientific computations, data-parallel language constructs might be needed.

A future research problem is to design data-parallel language constructs that can in a consistent way be added to the Modelica language, or as a separate extension of the Modelica language. These language constructs should be easy to understand, with well defined semantics, enabling the programmer to perform large and complex simulations and calculations by writing both algorithmic data-parallel code and declarative model systems using equation based models of physical systems.

9.4 Heterogeneous Multi-Processor Systems

For some types of simulations, like for instance Hardware-in-the-loop simulations, it might be necessary to use a heterogeneous multi-processor architecture. A hardware-in-the-loop simulation is a simulation of a physical system where some parts of the systems are already implemented in hardware. For instance, a control system for an industrial mechanical robot might be simulated before the robot is actually built by coupling the control system together with a computer simulating the physical behavior of the robot. For such simulations, the execution time of the simulation is of uttermost importance. Since some parts are already built, the simulation must be performed in real time.

To handle the simulation of large and complex systems in real time it is sometimes necessary to parallelize the problem. Also, for specific simulations the most economical solution might be to mix the computer hardware and software resources. For instance, the cheapest solution to simulate a mechanical robot might be to connect two conventional processors with two digital signal processors.

To fully take advantage of such a heterogeneous multiprocessor system, the scheduling algorithm must consider different processor speeds, computational skill, network layout, and so on. This area is becoming increasingly important in the future and contains many unsolved research problems, like the development of scheduling algorithms and parallel programming models for heterogeneous systems. Research is also needed regarding cost effective exploitation of computational power for specific computational needs of different kinds, like signal processing and scientific computations.

Appendix A

Modelica source code for the PreLoad example

This appendix contain the Modelica source code for the `PreLoad` example. The source code is taken from the Modelica Standard Library, but the graphical annotations in the source code have been removed to improve the readability of the code.

```
model PreLoad "Preload of a spool using ElastoGap models."
  extends Modelica.Icons.Example;
  Translational.ElastoGap SpringLe(
    s_rel0=1e-3,
    c=1000e3,
    d=250);
  Translational.ElastoGap SpringRi(
    s_rel0=1e-3,
    c=1000e3,
    d=250);
  Translational.SlidingMass Spool(
    L=0.19,
    m=0.150,
    s(start=8.5e-5));
  Translational.Fixed FixedLe(s0=-95.5e-3);
  Translational.SlidingMass PotLe(
    L=2e-3,
    m=10e-3,
    s(start=-93e-3));
  Translational.SlidingMass PotRi(
    L=2e-3,
    m=10e-3,
    s(start=-69.25e-3));
  Translational.Spring Spring(c=20e3, s_rel0=25e-3);
  Translational.ElastoGap PlateLe(
    s_rel0=1.5e-3,
    c=1000e3,
    d=250);
  Translational.ElastoGap PlateRi(
    c=1000e3,
    d=250,
    s_rel0=1.5e-3);
  Translational.Rod Rod(L=29.5e-3);
  Translational.Damper Friction(d=2500);
  Translational.Force Force1;
  Modelica.Blocks.Sources.Sine Sine1(amplitude={150}, freqHz={0.01});
  Translational.Rod Housing(L=30.5e-3);
equation
  connect(SpringLe.flange_b, Spool.flange_a);
  connect(PlateLe.flange_b, PotLe.flange_a);
  connect(PotLe.flange_b, Spring.flange_a);
  connect(Spring.flange_b, PotRi.flange_a);
  connect(PotRi.flange_b, PlateRi.flange_a);
  connect(SpringRi.flange_a, Spool.flange_a);
  connect(PotLe.flange_a, SpringRi.flange_b);
  connect(Rod.flange_b, PotRi.flange_b);
  connect(Rod.flange_a, SpringLe.flange_a);
  connect(FixedLe.flange_b, PlateLe.flange_a);
  connect(Friction.flange_a, FixedLe.flange_b);
  connect(Force1.flange_b, Spool.flange_b);
  connect(Sine1.outPort, Force1.inPort);
  connect(FixedLe.flange_b, Housing.flange_a);
  connect(PlateRi.flange_b, Housing.flange_b);
  connect(Friction.flange_b, Spool.flange_b);
end PreLoad;
```

Appendix B

Sequential code for the PreLoad example

This appendix shows the sequential C code produced by the Modelica compiler Dymola from the `preload` example. The actual code that is parallelized is found after the macro `DynamicsSection` and ends before the macro `AcceptedSection1`.

```
/* DSblock model generated by Dymola from Modelica model Modelica.Mechanics.Translational.Examples.PreLoad
 */

#include <matrixop.h>
/* Prototypes for functions used in model */
/* Codes used in model */
/* DSblock C-code: */

#include <moutil.c>
#include <dsblock1.c>

/* Define variable names. */

#define Sections_
#define SpringLe_flangex_0a_s  Variable(0)
#define SpringLe_flangex_0a_der_s  Variable(1)
#define SpringLe_sx_0rel  Variable(2)
#define SpringLe_der_sx_0rel  Variable(3)
#define SpringLe_sx_0rel0  Parameter(0)
#define SpringLe_c  Parameter(1)
#define SpringLe_d  Parameter(2)
#define SpringLe_Contact  Variable(4)
#define SpringRi_flangex_0b_f  Variable(5)
#define SpringRi_sx_0rel  Variable(6)
#define SpringRi_der_sx_0rel  Variable(7)
#define SpringRi_sx_0rel0  Parameter(3)
#define SpringRi_c  Parameter(4)
#define SpringRi_d  Parameter(5)
#define SpringRi_Contact  Variable(8)
#define Spool_s  State(0)
#define Spool_der_s  Derivative(0)
#define Spool_L  Parameter(6)
#define Spool_flangex_0a_s  Variable(9)
#define Spool_flangex_0a_der_s  Variable(10)
#define Spool_flangex_0a_f  Variable(11)
#define Spool_flangex_0b_s  Variable(12)
#define Spool_m  Parameter(7)
#define Spool_v  State(1)
#define Spool_der_v  Derivative(1)
#define FixedLe_s0  Parameter(8)
```

```
#define FixedLe_flangex_0b_s  Variable(13)
#define FixedLe_flangex_0b_f  Variable(14)
#define PotLe_s  State(2)
#define PotLe_der_s  Derivative(2)
#define PotLe_L  Parameter(9)
#define PotLe_flangex_0a_s  Variable(15)
#define PotLe_flangex_0a_der_s  Variable(16)
#define PotLe_flangex_0a_f  Variable(17)
#define PotLe_m  Parameter(10)
#define PotLe_v  State(3)
#define PotLe_der_v  Derivative(3)
#define PotRi_s  State(4)
#define PotRi_der_s  Derivative(4)
#define PotRi_L  Parameter(11)
#define PotRi_flangex_0b_s  Variable(18)
#define PotRi_flangex_0b_der_s  Variable(19)
#define PotRi_flangex_0b_f  Variable(20)
#define PotRi_m  Parameter(12)
#define PotRi_v  State(5)
#define PotRi_der_v  Derivative(5)
#define Spring_flangex_0a_s  Variable(21)
#define Spring_flangex_0b_s  Variable(22)
#define Spring_sx_0rel  Variable(23)
#define Spring_f  Variable(24)
#define Spring_sx_0rel0  Parameter(13)
#define Spring_c  Parameter(14)
#define PlateLe_flangex_0a_s  Variable(25)
#define PlateLe_flangex_0b_f  Variable(26)
#define PlateLe_sx_0rel  Variable(27)
#define PlateLe_der_sx_0rel  Variable(28)
#define PlateLe_sx_0rel0  Parameter(15)
#define PlateLe_c  Parameter(16)
#define PlateLe_d  Parameter(17)
#define PlateLe_Contact  Variable(29)
#define PlateRi_flangex_0b_s  Variable(30)
#define PlateRi_sx_0rel  Variable(31)
#define PlateRi_der_sx_0rel  Variable(32)
#define PlateRi_sx_0rel0  Parameter(18)
#define PlateRi_c  Parameter(19)
#define PlateRi_d  Parameter(20)
#define PlateRi_Contact  Variable(33)
#define Rod_s  Variable(34)
#define Rod_der_s  Variable(35)
#define Rod_L  Parameter(21)
#define Rod_flangex_0b_f  Variable(36)
#define Friction_flangex_0a_s  Variable(37)
#define Friction_flangex_0b_f  Variable(38)
#define Friction_sx_0rel  Variable(39)
#define Friction_der_sx_0rel  Variable(40)
#define Friction_d  Parameter(22)
#define Force1_inPort_n  Variable(41)
#define Sine1_nout  Variable(42)
#define Sine1_outPort_n  Variable(43)
#define Sine1_outPort_signal_1  Variable(44)
#define Sine1_y_1  Variable(45)
#define Sine1_amplitude  &Parameter(23)
#define Sine1_amplitude_1  Parameter(23)
#define Sine1_freqHz  &Parameter(24)
#define Sine1_freqHz_1  Parameter(24)
#define Sine1_phase  &Parameter(25)
#define Sine1_phase_1  Parameter(25)
#define Sine1_offset  &Parameter(26)
#define Sine1_offset_1  Parameter(26)
#define Sine1_startTime  &Parameter(27)
#define Sine1_startTime_1  Parameter(27)
#define Sine1_pi  Variable(46)
#define Sine1_px_0amplitude  &Variable(47)
#define Sine1_px_0amplitude_1  Variable(47)
#define Sine1_px_0freqHz  &Variable(48)
#define Sine1_px_0freqHz_1  Variable(48)
#define Sine1_px_0phase  &Variable(49)
#define Sine1_px_0phase_1  Variable(49)
#define Sine1_px_0offset  &Variable(50)
#define Sine1_px_0offset_1  Variable(50)
#define Sine1_px_0startTime  &Variable(51)
#define Sine1_px_0startTime_1  Variable(51)
#define Housing_s  Variable(52)
#define Housing_L  Parameter(28)
#define Housing_flangex_0a_s  Variable(53)
#define Housing_flangex_0a_f  Variable(54)
#define Housing_flangex_0b_s  Variable(55)
```

```
TranslatedEquations

InitialSection
Sine1_nout = 1;
Sine1_outPort_n = 1;
Sine1_pi = 3.14159265358979;
Force1_inPort_n = 1;
BoundParameterSection
Sine1_px_Oamplitude_1 = Sine1_amplitude_1;
Sine1_px_OfreqHz_1 = Sine1_freqHz_1;
Sine1_px_Ophase_1 = Sine1_phase_1;
Sine1_px_Ooffset_1 = Sine1_offset_1;
Sine1_px_OstartTime_1 = Sine1_startTime_1;
Housing_s = FixedLe_s0+0.5*Housing_L;
PlateRi_flangex_Ob_s = Housing_s+0.5*Housing_L;
Friction_flangex_Oa_s = FixedLe_s0;
FixedLe_flangex_Ob_s = FixedLe_s0;
Housing_flangex_Oa_s = FixedLe_s0;
PlateLe_flangex_Oa_s = FixedLe_s0;
Housing_flangex_Ob_s = PlateRi_flangex_Ob_s;
InitialSection
DefaultSection
InitializeData(0)

OutputSection

DynamicsSection
PotLe_der_s = PotLe_v;
PotLe_flangex_Oa_s = PotLe_s-0.5*PotLe_L;
PlateLe_sx_Orel = PotLe_flangex_Oa_s-FixedLe_s0;
PlateLe_Contact = Less(PlateLe_sx_Orel,"PlateLe.s_rel", PlateLe_sx_Orel0,
"PlateLe.s_rel0", 0);
PotLe_flangex_Oa_der_s = PotLe_der_s;
PlateLe_der_sx_Orel = PotLe_flangex_Oa_der_s;
PlateLe_flangex_Ob_f = IF PlateLe_Contact THEN PlateLe_c*(PlateLe_sx_Orel-
PlateLe_sx_Orel0)+PlateLe_d*PlateLe_der_sx_Orel ELSE 0;
Spool_flangex_Oa_s = Spool_s-0.5*Spool_L;
SpringRi_sx_Orel = PotLe_flangex_Oa_s-Spool_flangex_Oa_s;
SpringRi_Contact = Less(SpringRi_sx_Orel,"SpringRi.s_rel", SpringRi_sx_Orel0,
"SpringRi.s_rel0", 1);
Spool_der_s = Spool_v;
Spool_flangex_Oa_der_s = Spool_der_s;
SpringRi_der_sx_Orel = PotLe_flangex_Oa_der_s-Spool_flangex_Oa_der_s;
SpringRi_flangex_Ob_f = IF SpringRi_Contact THEN SpringRi_c*(SpringRi_sx_Orel-
SpringRi_sx_Orel0)+SpringRi_d*SpringRi_der_sx_Orel ELSE 0;
PotLe_flangex_Oa_f =  -(PlateLe_flangex_Ob_f+SpringRi_flangex_Ob_f);
Spring_flangex_Ob_s = PotRi_s-0.5*PotRi_L;
Spring_flangex_Oa_s = PotLe_s+0.5*PotLe_L;
Spring_sx_Orel = Spring_flangex_Ob_s-Spring_flangex_Oa_s;
Spring_f = Spring_c*(Spring_sx_Orel-Spring_sx_Orel0);
PotLe_der_v = divmacro(PotLe_flangex_Oa_f+Spring_f,"PotLe.flange_a.f+Spring.f",
PotLe_m,"PotLe.m");
Friction_der_sx_Orel = Spool_flangex_Oa_der_s;
Friction_flangex_Ob_f = Friction_d*Friction_der_sx_Orel;
PotRi_flangex_Ob_s = PotRi_s+0.5*PotRi_L;
Rod_s = PotRi_flangex_Ob_s-0.5*Rod_L;
SpringLe_flangex_Oa_s = Rod_s-0.5*Rod_L;
SpringLe_sx_Orel = Spool_flangex_Oa_s-SpringLe_flangex_Oa_s;
SpringLe_Contact = Less(SpringLe_sx_Orel,"SpringLe.s_rel", SpringLe_sx_Orel0,
"SpringLe.s_rel0", 2);
PotRi_der_s = PotRi_v;
PotRi_flangex_Ob_der_s = PotRi_der_s;
Rod_der_s = PotRi_flangex_Ob_der_s;
SpringLe_flangex_Oa_der_s = Rod_der_s;
SpringLe_der_sx_Orel = Spool_flangex_Oa_der_s-SpringLe_flangex_Oa_der_s;
Rod_flangex_Ob_f =  -(IF SpringLe_Contact THEN SpringLe_c*(SpringLe_sx_Orel-
SpringLe_sx_Orel0)+SpringLe_d*SpringLe_der_sx_Orel ELSE 0);
Spool_flangex_Oa_f = SpringRi_flangex_Ob_f-Friction_flangex_Ob_f+
Rod_flangex_Ob_f;
Sine1_outPort_signal_1 = Sine1_px_Ooffset_1+(IF LessTime(Sine1_px_OstartTime_1,
0) THEN 0 ELSE Sine1_px_Oamplitude_1*sin(6.28318530717959*Sine1_px_OfreqHz_1*(
Time-Sine1_px_OstartTime_1)+Sine1_px_Ophase_1));
Spool_der_v = divmacro(Spool_flangex_Oa_f+Sine1_outPort_signal_1,
"Spool.flange_a.f+Sine1.outPort.signal[1]",Spool_m,"Spool.m");
PlateRi_sx_Orel = PlateRi_flangex_Ob_s-PotRi_flangex_Ob_s;
PlateRi_Contact = Less(PlateRi_sx_Orel,"PlateRi.s_rel", PlateRi_sx_Orel0,
"PlateRi.s_rel0", 3);
PlateRi_der_sx_Orel =  -PotRi_flangex_Ob_der_s;
Housing_flangex_Oa_f = IF PlateRi_Contact THEN PlateRi_c*(PlateRi_sx_Orel-
PlateRi_sx_Orel0)+PlateRi_d*PlateRi_der_sx_Orel ELSE 0;
PotRi_flangex_Ob_f = Housing_flangex_Oa_f-Rod_flangex_Ob_f;
PotRi_der_v = divmacro(PotRi_flangex_Ob_f-Spring_f,"PotRi.flange_b.f-Spring.f",
PotRi_m,"PotRi.m");

AcceptedSection1

AcceptedSection2
Sine1_y_1 = Sine1_outPort_signal_1;
Spool_flangex_0b_s = Spool_s+0.5*Spool_L;
FixedLe_flangex_0b_f = PlateLe_flangex_0b_f-Housing_flangex_0a_f+
Friction_flangex_0b_f;
Friction_sx_0rel = Spool_flangex_0a_s-FixedLe_s0;

DefaultSection
InitialSection
/* No equations */
DefaultSection
InitializeData(1)
EndTranslatedEquations

#include <dsblock6.c>

PreNonAlias(0)
StartNonAlias(0)
DeclareVariable("SpringLe.flange_a.s", "absolute position of flange [m]", 0, \
0.0,0.0,0.0,0,0)
DeclareVariable("SpringLe.flange_a.der(s)", "der(absolute position of flange) [m/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareVariable("SpringLe.s_rel", "relative distance (= flange_b.s - flange_a.s) [m]"\
, 0, 0.0,1E+100,0.0,0,0)
DeclareVariable("SpringLe.der(s_rel)", "der(relative distance (= flange_b.s - flange_a.s)) [m/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareParameter("SpringLe.s_rel0", "unstretched spring length [m]", 0, 0, \
0.0,0.0,0.0,0,0)
DeclareParameter("SpringLe.c", "spring constant [N/m]", 1, 1000000.0, 0.0,1E+100\
,0.0,0,0)
DeclareParameter("SpringLe.d", "damping constant [N/ (m/s)]", 2, 250, 0.0,1E+100\
,0.0,0,0)
DeclareVariable("SpringLe.Contact", "false, if s_rel > l ", false, 0.0,0.0,0.0,0\
,0)
DeclareVariable("SpringRi.flange_b.f", "cut force directed into flange [N]", 0, \
0.0,0.0,0.0,0,0)
DeclareVariable("SpringRi.s_rel", "relative distance (= flange_b.s - flange_a.s) [m]"\
, 0, 0.0,1E+100,0.0,0,0)
DeclareVariable("SpringRi.der(s_rel)", "der(relative distance (= flange_b.s - flange_a.s)) [m/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareParameter("SpringRi.s_rel0", "unstretched spring length [m]", 3, 0, \
0.0,0.0,0.0,0,0)
DeclareParameter("SpringRi.c", "spring constant [N/m]", 4, 1000000.0, 0.0,1E+100\
,0.0,0,0)
DeclareParameter("SpringRi.d", "damping constant [N/ (m/s)]", 5, 250, 0.0,1E+100\
,0.0,0,0)
DeclareVariable("SpringRi.Contact", "false, if s_rel > l ", false, 0.0,0.0,0.0,0\
,0)
DeclareState("Spool.s", "absolute position of center of component (s = flange_a.s + L/2 = flange_b.s - L/2) [m]"\
, 0, 0, 0.0,0.0,0.0,0,0)
DeclareDerivative("Spool.der(s)", "der(absolute position of center of component (s = flange_a.s + L/2 = flange_b.s - L/2)) [m/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareParameter("Spool.L", "length of component from left flange to right flange (= flange_b.s - flange_a.s) [m]"\
, 6, 0.19, 0.0,0.0,0.0,0,0)
DeclareVariable("Spool.flange_a.s", "absolute position of flange [m]", 0, \
0.0,0.0,0.0,0,0)
DeclareVariable("Spool.flange_a.der(s)", "der(absolute position of flange) [m/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareVariable("Spool.flange_a.f", "cut force directed into flange [N]", 0, \
0.0,0.0,0.0,0,0)
DeclareVariable("Spool.flange_b.s", "absolute position of flange [m]", 0, \
0.0,0.0,0.0,0,0)
DeclareParameter("Spool.m", "mass of the sliding mass [kg]", 7, 0.15, 0.0,1E+100\
,0.0,0,0)
DeclareState("Spool.v", "absolute velocity of component [m/s]", 1, 0, 0.0,0.0,\
0.0,0,0)
DeclareDerivative("Spool.der(v)", "der(absolute velocity of component) [m/s/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareParameter("FixedLe.s0", "fixed offset position of housing [m]", 8, (\
-0.0965), 0.0,0.0,0.0,0,0)
DeclareVariable("FixedLe.flange_b.s", "absolute position of flange [m]", 0, \
0.0,0.0,0.0,0,1)
DeclareVariable("FixedLe.flange_b.f", "cut force directed into flange [N]", 0, \
0.0,0.0,0.0,0,0)
DeclareState("PotLe.s", "absolute position of center of component (s = flange_a.s + L/2 = flange_b.s - L/2) [m]"\
, 2, (-0.094), 0.0,0.0,0.0,0,0)
DeclareDerivative("PotLe.der(s)", "der(absolute position of center of component (s = flange_a.s + L/2 = flange_b.s - L/2)) [m/s]"\
, 0, 0.0,0.0,0.0,0,0)
```

```
DeclareParameter("PotLe.L", "length of component from left flange to right flange (= flange_b.s - flange_a.s) [m]"\
, 9, 0.002, 0.0,0.0,0.0,0,0)
DeclareVariable("PotLe.flange_a.s", "absolute position of flange [m]", 0, \
0.0,0.0,0.0,0,0)
DeclareVariable("PotLe.flange_a.der(s)", "der(absolute position of flange) [m/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareVariable("PotLe.flange_a.f", "cut force directed into flange [N]", 0, \
0.0,0.0,0.0,0,0)
DeclareParameter("PotLe.m", "mass of the sliding mass [kg]", 10, 0.01, 0.0,\
1E+100,0.0,0,0)
DeclareState("PotLe.v", "absolute velocity of component [m/s]", 3, 0, 0.0,0.0,\
0.0,0,0)
DeclareDerivative("PotLe.der(v)", "der(absolute velocity of component) [m/s/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareState("PotRi.s", "absolute position of center of component (s = flange_a.s + L/2 = flange_b.s - L/2) [m]"\
, 4, (-0.0655), 0.0,0.0,0.0,0,0)
DeclareDerivative("PotRi.der(s)", "der(absolute position of center of component (s = flange_a.s + L/2 = flange_b.s - L/2)) [m/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareParameter("PotRi.L", "length of component from left flange to right flange (= flange_b.s - flange_a.s) [m]"\
, 11, 0.002, 0.0,0.0,0.0,0,0)
DeclareVariable("PotRi.flange_b.s", "absolute position of flange [m]", 0, \
0.0,0.0,0.0,0,0)
DeclareVariable("PotRi.flange_b.der(s)", "der(absolute position of flange) [m/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareVariable("PotRi.flange_b.f", "cut force directed into flange [N]", 0, \
0.0,0.0,0.0,0,0)
DeclareParameter("PotRi.m", "mass of the sliding mass [kg]", 12, 0.01, 0.0,\
1E+100,0.0,0,0)
DeclareState("PotRi.v", "absolute velocity of component [m/s]", 5, 0, 0.0,0.0,\
0.0,0,0)
DeclareDerivative("PotRi.der(v)", "der(absolute velocity of component) [m/s/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareVariable("Spring.flange_a.s", "absolute position of flange [m]", 0, \
0.0,0.0,0.0,0,0)
DeclareVariable("Spring.flange_b.s", "absolute position of flange [m]", 0, \
0.0,0.0,0.0,0,0)
DeclareVariable("Spring.s_rel", "relative distance (= flange_b.s - flange_a.s) [m]"\
, 0, 0.0,1E+100,0.0,0,0)
DeclareVariable("Spring.f", "forcee between flanges (positive in direction of flange axis R) [N]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareParameter("Spring.s_rel0", "unstretched spring length [m]", 13, 0.025, \
0.0,1E+100,0.0,0,0)
DeclareParameter("Spring.c", "spring constant  [N/m]", 14, 20000.0, 0.0,1E+100,\
0.0,0,0)
DeclareVariable("PlateLe.flange_a.s", "absolute position of flange [m]", 0, \
0.0,0.0,0.0,0,1)
DeclareVariable("PlateLe.flange_b.f", "cut force directed into flange [N]", 0, \
0.0,0.0,0.0,0,0)
DeclareVariable("PlateLe.s_rel", "relative distance (= flange_b.s - flange_a.s) [m]"\
, 0, 0.0,1E+100,0.0,0,0)
DeclareVariable("PlateLe.der(s_rel)", "der(relative distance (= flange_b.s - flange_a.s)) [m/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareParameter("PlateLe.s_rel0", "unstretched spring length [m]", 15, 0.0015, \
0.0,0.0,0.0,0,0)
DeclareParameter("PlateLe.c", "spring constant [N/m]", 16, 1000000.0, 0.0,1E+100\
,0.0,0,0)
DeclareParameter("PlateLe.d", "damping constant [N/ (m/s)]", 17, 250, 0.0,1E+100\
,0.0,0,0)
DeclareVariable("PlateLe.Contact", "false, if s_rel > l ", false, 0.0,0.0,0.0,0,\
0)
DeclareVariable("PlateRi.flange_b.s", "absolute position of flange [m]", 0, \
0.0,0.0,0.0,0,1)
DeclareVariable("PlateRi.s_rel", "relative distance (= flange_b.s - flange_a.s) [m]"\
, 0, 0.0,1E+100,0.0,0,0)
DeclareVariable("PlateRi.der(s_rel)", "der(relative distance (= flange_b.s - flange_a.s)) [m/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareParameter("PlateRi.s_rel0", "unstretched spring length [m]", 18, 0.0015, \
0.0,0.0,0.0,0,0)
DeclareParameter("PlateRi.c", "spring constant [N/m]", 19, 1000000.0, 0.0,1E+100\
,0.0,0,0)
DeclareParameter("PlateRi.d", "damping constant [N/ (m/s)]", 20, 250, 0.0,1E+100\
,0.0,0,0)
DeclareVariable("PlateRi.Contact", "false, if s_rel > l ", false, 0.0,0.0,0.0,0,\
0)
DeclareVariable("Rod.s", "absolute position of center of component (s = flange_a.s + L/2 = flange_b.s - L/2) [m]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareVariable("Rod.der(s)", "der(absolute position of center of component (s = flange_a.s + L/2 = flange_b.s - L/2)) [m/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareParameter("Rod.L", "length of component from left flange to right flange (= flange_b.s - flange_a.s) [m]"\
, 21, 0.0275, 0.0,0.0,0.0,0,0)
DeclareVariable("Rod.flange_b.f", "cut force directed into flange [N]", 0, \
0.0,0.0,0.0,0,0)
```

```
DeclareVariable("Friction.flange_a.s", "absolute position of flange [m]", 0, \
0.0,0.0,0.0,0,1)
DeclareVariable("Friction.flange_b.f", "cut force directed into flange [N]", 0, \
0.0,0.0,0.0,0,0)
DeclareVariable("Friction.s_rel", "relative distance (= flange_b.s - flange_a.s) [m]"\
, 0, 0.0,1E+100,0.0,0,0)
DeclareVariable("Friction.der(s_rel)", "der(relative distance (= flange_b.s - flange_a.s)) [m/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareParameter("Friction.d", "damping constant [N/ (m/s)] [N/ (m/s)]", 22, \
2500, 0.0,1E+100,0.0,0,0)
DeclareVariable("Force1.inPort.n", "Dimension of signal vector", 1, 0.0,0.0,0.0,\
0,1)
DeclareVariable("Sine1.nout", "Number of outputs", 1, 1.0,1E+100,0.0,0,1)
DeclareVariable("Sine1.outPort.n", "Dimension of signal vector", 1, 0.0,0.0,0.0,\
0,1)
DeclareVariable("Sine1.outPort.signal[1]", "Real output signals", 0, 0.0,0.0,0.0\
,0,0)
DeclareVariable("Sine1.y[1]", "", 0, 0.0,0.0,0.0,0,0)
DeclareParameter("Sine1.amplitude[1]", "Amplitudes of sine waves", 23, 150, \
0.0,0.0,0.0,0,0)
DeclareParameter("Sine1.freqHz[1]", "Frequencies of sine waves [Hz]", 24, 0.01, \
0.0,0.0,0.0,0,0)
DeclareParameter("Sine1.phase[1]", "Phases of sine waves [rad]", 25, 0, 0.0,0.0,\
0.0,0,0)
DeclareParameter("Sine1.offset[1]", "Offsets of output signals", 26, 0, 0.0,0.0,\
0.0,0,0)
DeclareParameter("Sine1.startTime[1]", "Output = offset for time < startTime [s]"\
, 27, 0, 0.0,0.0,0.0,0,0)
DeclareVariable("Sine1.pi", "", 3.14159265358979, 0.0,0.0,0.0,0,1)
DeclareVariable("Sine1.p_amplitude[1]", "", 0, 0.0,0.0,0.0,0,1)
DeclareVariable("Sine1.p_freqHz[1]", "", 0, 0.0,0.0,0.0,0,1)
DeclareVariable("Sine1.p_phase[1]", "", 0, 0.0,0.0,0.0,0,1)
DeclareVariable("Sine1.p_offset[1]", "", 0, 0.0,0.0,0.0,0,1)
DeclareVariable("Sine1.p_startTime[1]", "[s]", 0, 0.0,0.0,0.0,0,1)
DeclareVariable("Housing.s", "absolute position of center of component (s = flange_a.s + L/2 = flange_b.s - L/2) [m]"\
, 0, 0.0,0.0,0.0,0,1)
DeclareParameter("Housing.L", "length of component from left flange to right flange (= flange_b.s - flange_a.s) [m]"\
, 28, 0.0305, 0.0,0.0,0.0,0,0)
DeclareVariable("Housing.flange_a.s", "absolute position of flange [m]", 0, \
0.0,0.0,0.0,0,1)
DeclareVariable("Housing.flange_a.f", "cut force directed into flange [N]", 0, \
0.0,0.0,0.0,0,0)
DeclareVariable("Housing.flange_b.s", "absolute position of flange [m]", 0, \
0.0,0.0,0.0,0,1)
EndNonAlias(0)

PreAlias(0)
StartAlias(0)
DeclareAlias("SpringLe.flange_a.f", "cut force directed into flange [N]", "\
Rod.flange_b.f", 1, 5, 36)
DeclareAlias("SpringLe.flange_b.s", "absolute position of flange [m]", "\
Spool.flange_a.s", 1, 5, 9)
DeclareAlias("SpringLe.flange_b.f", "cut force directed into flange [N]", "\
Rod.flange_b.f", -1, 5, 36)
DeclareAlias("SpringLe.f", "forcee between flanges (positive in direction of flange axis R) [N]"\
, "Rod.flange_b.f", -1, 5, 36)
DeclareAlias("SpringLe.v_rel", "relative velocity between flange L and R [m/s]"\
, "SpringLe.der(s_rel)", 1, 5, 3)
DeclareAlias("SpringRi.flange_a.s", "absolute position of flange [m]", "\
Spool.flange_a.s", 1, 5, 9)
DeclareAlias("SpringRi.flange_a.f", "cut force directed into flange [N]", "\
SpringRi.flange_b.f", -1, 5, 5)
DeclareAlias("SpringRi.flange_b.s", "absolute position of flange [m]", "\
PotLe.flange_a.s", 1, 5, 15)
DeclareAlias("SpringRi.f", "forcee between flanges (positive in direction of flange axis R) [N]"\
, "SpringRi.flange_b.f", 1, 5, 5)
DeclareAlias("SpringRi.v_rel", "relative velocity between flange L and R [m/s]"\
, "SpringRi.der(s_rel)", 1, 5, 7)
DeclareAlias("Spool.flange_b.f", "cut force directed into flange [N]", "\
Sine1.outPort.signal[1]", 1, 5, 44)
DeclareAlias("Spool.a", "absolute acceleration of component [m/s2]", "\
Spool.der(v)", 1, 6, 1)
DeclareAlias("PotLe.flange_b.s", "absolute position of flange [m]", "\
Spring.flange_a.s", 1, 5, 21)
DeclareAlias("PotLe.flange_b.f", "cut force directed into flange [N]", "Spring.f\
", 1, 5, 24)
DeclareAlias("PotLe.a", "absolute acceleration of component [m/s2]", "\
PotLe.der(v)", 1, 6, 3)
DeclareAlias("PotRi.flange_a.s", "absolute position of flange [m]", "\
Spring.flange_b.s", 1, 5, 22)
DeclareAlias("PotRi.flange_a.f", "cut force directed into flange [N]", "Spring.f\
", -1, 5, 24)
```

```
DeclareAlias("PotRi.a", "absolute acceleration of component [m/s2]", "\
PotRi.der(v)", 1, 6, 5)
DeclareAlias("Spring.flange_a.f", "cut force directed into flange [N]", "\
Spring.f", -1, 5, 24)
DeclareAlias("Spring.flange_b.f", "cut force directed into flange [N]", "\
Spring.f", 1, 5, 24)
DeclareAlias("PlateLe.flange_a.f", "cut force directed into flange [N]", "\
PlateLe.flange_b.f", -1, 5, 26)
DeclareAlias("PlateLe.flange_b.s", "absolute position of flange [m]", "\
PotLe.flange_a.s", 1, 5, 15)
DeclareAlias("PlateLe.f", "forcee between flanges (positive in direction of flange axis R) [N]"\
, "PlateLe.flange_b.f", 1, 5, 26)
DeclareAlias("PlateLe.v_rel", "relative velocity between flange L and R [m/s]"\
, "PlateLe.der(s_rel)", 1, 5, 28)
DeclareAlias("PlateRi.flange_a.s", "absolute position of flange [m]", "\
PotRi.flange_b.s", 1, 5, 18)
DeclareAlias("PlateRi.flange_a.f", "cut force directed into flange [N]", "\
Housing.flange_a.f", -1, 5, 54)
DeclareAlias("PlateRi.flange_b.f", "cut force directed into flange [N]", "\
Housing.flange_a.f", 1, 5, 54)
DeclareAlias("PlateRi.f", "forcee between flanges (positive in direction of flange axis R) [N]"\
, "Housing.flange_a.f", 1, 5, 54)
DeclareAlias("PlateRi.v_rel", "relative velocity between flange L and R [m/s]"\
, "PlateRi.der(s_rel)", 1, 5, 32)
DeclareAlias("Rod.flange_a.s", "absolute position of flange [m]", "\
SpringLe.flange_a.s", 1, 5, 0)
DeclareAlias("Rod.flange_a.f", "cut force directed into flange [N]", "\
Rod.flange_b.f", -1, 5, 36)
DeclareAlias("Rod.flange_b.s", "absolute position of flange [m]", "\
PotRi.flange_b.s", 1, 5, 18)
DeclareAlias("Friction.flange_a.f", "cut force directed into flange [N]", "\
Friction.flange_b.f", -1, 5, 38)
DeclareAlias("Friction.flange_b.s", "absolute position of flange [m]", "\
Spool.flange_a.s", 1, 5, 9)
DeclareAlias("Friction.f", "forcee between flanges (positive in direction of flange axis R) [N]"\
, "Friction.flange_b.f", 1, 5, 38)
DeclareAlias("Friction.v_rel", "relative velocity between flange L and R [m/s]"\
, "Friction.der(s_rel)", 1, 5, 40)
DeclareAlias("Force1.f", "driving force [N]", "Sine1.outPort.signal[1]", 1, 5, \
44)
DeclareAlias("Force1.flange_b.s", "absolute position of flange [m]", "\
Spool.flange_b.s", 1, 5, 12)
DeclareAlias("Force1.flange_b.f", "cut force directed into flange [N]", "\
Sine1.outPort.signal[1]", -1, 5, 44)
DeclareAlias("Force1.inPort.signal[1]", "Real input signals", "Sine1.outPort.signal[1]\
", 1, 5, 44)
DeclareAlias("Housing.flange_b.f", "cut force directed into flange [N]", "\
Housing.flange_a.f", -1, 5, 54)
EndAlias(0)
#define NX_    6
#define NX2_   0
#define NU_    0
#define NY_    0
#define NW_    56
#define NP_    29
#define NI_    0
#define NRel_  4
#define NTim_  1
#define NSamp_ 0
#define NCons_ 0
#define NA_    41
#define SizePre_ 0
#define SizeEq_ 0
#define SizeDelay_ 0
#define QNLmax_ 0
#define MAXAux 0
#define NrDymolaTimers_ 0
#define NWhen_ 0

#include <dsblock5.c>

StartDataBlock
EndDataBlock
```

Appendix C

Parallel code for the PreLoad example

This appendix shows the parallel code from the `PreLoad` example produced by the parallelization tool for two processors using the FTD method.

```
#include <mpi.h>
void proc0(long *idemand, long *icall_, double *time
 ,double X_[], double XD_[], double U_[], double DP_[]
 ,long IP_[], long LP_[]
, double F_[], double Y_[], double W_[], double QZ_[]
, double duser_[], long iuser_[]
 ,long luser_[],long *QiErr);
#ifndef __DSBPART_MPI_MACROS
#define __DSBPART_MPI_MACROS
#define MSEND(buf,count,type,proc,tag) MPI_Send(buf,count,type,proc,tag,MPI_COMM_WORLD)
#define MRECV(buf,count,type,proc,tag) MPI_Recv(buf,count,type,proc,tag,MPI_COMM_WORLD,&status)
#define BARRIER MPI_Barrier(MPI_COMM_WORLD)
#define KILLCOMMAND 99
#define SENDKILL {int __i; killbuf[0]=0;for(i=1;i<2;i++) MSEND(&killbuf,18+2*(NRel_+1),MPI_LONG,i,0);}
#define _BreakFunction(nr) {void proc_##nr(void); proc_##nr();} } void proc_##nr(void) {
#endif
double sendvar,recvvar,killbuf;
MPI_Status status;
MPI_Request request;
int myrank,packedsize;
long tempi[11];
double tempr[48];
/* DSblock model generated by Dymola from Modelica model Modelica.Mechanics.Translational.Examples.PreLoad
 */

#include <matrixop.h>
/* Prototypes for functions used in model */
/* Codes used in model */
/* DSblock C-code: */

#include <moutil.c>
#include <dsblock1.c>

/* Define variable names. */


#define SpringLe_flangex_0a_s  Variable(0)
#define SpringLe_flangex_0a_der_s  Variable(1)
#define SpringLe_sx_0rel  Variable(2)
#define SpringLe_der_sx_0rel  Variable(3)
#define SpringLe_sx_0rel0  Parameter(0)
#define SpringLe_c  Parameter(1)
#define SpringLe_d  Parameter(2)
#define SpringLe_Contact  Variable(4)
```

```
#define SpringRi_flangex_0b_f  Variable(5)
#define SpringRi_sx_0rel  Variable(6)
#define SpringRi_der_sx_0rel  Variable(7)
#define SpringRi_sx_0rel0  Parameter(3)
#define SpringRi_c  Parameter(4)
#define SpringRi_d  Parameter(5)
#define SpringRi_Contact  Variable(8)
#define Spool_s  State(0)
#define Spool_der_s  Derivative(0)
#define Spool_L  Parameter(6)
#define Spool_flangex_0a_s  Variable(9)
#define Spool_flangex_0a_der_s  Variable(10)
#define Spool_flangex_0a_f  Variable(11)
#define Spool_flangex_0b_s  Variable(12)
#define Spool_m  Parameter(7)
#define Spool_v  State(1)
#define Spool_der_v  Derivative(1)
#define FixedLe_s0  Parameter(8)
#define FixedLe_flangex_0b_s  Variable(13)
#define FixedLe_flangex_0b_f  Variable(14)
#define PotLe_s  State(2)
#define PotLe_der_s  Derivative(2)
#define PotLe_L  Parameter(9)
#define PotLe_flangex_0a_s  Variable(15)
#define PotLe_flangex_0a_der_s  Variable(16)
#define PotLe_flangex_0a_f  Variable(17)
#define PotLe_m  Parameter(10)
#define PotLe_v  State(3)
#define PotLe_der_v  Derivative(3)
#define PotRi_s  State(4)
#define PotRi_der_s  Derivative(4)
#define PotRi_L  Parameter(11)
#define PotRi_flangex_0b_s  Variable(18)
#define PotRi_flangex_0b_der_s  Variable(19)
#define PotRi_flangex_0b_f  Variable(20)
#define PotRi_m  Parameter(12)
#define PotRi_v  State(5)
#define PotRi_der_v  Derivative(5)
#define Spring_flangex_0a_s  Variable(21)
#define Spring_flangex_0b_s  Variable(22)
#define Spring_sx_0rel  Variable(23)
#define Spring_f  Variable(24)
#define Spring_sx_0rel0  Parameter(13)
#define Spring_c  Parameter(14)
#define PlateLe_flangex_0a_s  Variable(25)
#define PlateLe_flangex_0b_f  Variable(26)
#define PlateLe_sx_0rel  Variable(27)
#define PlateLe_der_sx_0rel  Variable(28)
#define PlateLe_sx_0rel0  Parameter(15)
#define PlateLe_c  Parameter(16)
#define PlateLe_d  Parameter(17)
#define PlateLe_Contact  Variable(29)
#define PlateRi_flangex_0b_s  Variable(30)
#define PlateRi_sx_0rel  Variable(31)
#define PlateRi_der_sx_0rel  Variable(32)
#define PlateRi_sx_0rel0  Parameter(18)
#define PlateRi_c  Parameter(19)
#define PlateRi_d  Parameter(20)
#define PlateRi_Contact  Variable(33)
#define Rod_s  Variable(34)
#define Rod_der_s  Variable(35)
#define Rod_L  Parameter(21)
#define Rod_flangex_0b_f  Variable(36)
#define Friction_flangex_0a_s  Variable(37)
#define Friction_flangex_0b_f  Variable(38)
#define Friction_sx_0rel  Variable(39)
#define Friction_der_sx_0rel  Variable(40)
#define Friction_d  Parameter(22)
#define Force1_inPort_n  Variable(41)
#define Sine1_nout  Variable(42)
#define Sine1_outPort_n  Variable(43)
#define Sine1_outPort_signal_1  Variable(44)
#define Sine1_y_1  Variable(45)
#define Sine1_amplitude  &Parameter(23)
#define Sine1_amplitude_1  Parameter(23)
#define Sine1_freqHz  &Parameter(24)
#define Sine1_freqHz_1  Parameter(24)
#define Sine1_phase  &Parameter(25)
#define Sine1_phase_1  Parameter(25)
#define Sine1_offset  &Parameter(26)
#define Sine1_offset_1  Parameter(26)
#define Sine1_startTime  &Parameter(27)
```

```
#define Sine1_startTime_1  Parameter(27)
#define Sine1_pi  Variable(46)
#define Sine1_px_0amplitude  &Variable(47)
#define Sine1_px_0amplitude_1  Variable(47)
#define Sine1_px_0freqHz  &Variable(48)
#define Sine1_px_0freqHz_1  Variable(48)
#define Sine1_px_0phase  &Variable(49)
#define Sine1_px_0phase_1  Variable(49)
#define Sine1_px_0offset  &Variable(50)
#define Sine1_px_0offset_1  Variable(50)
#define Sine1_px_0startTime  &Variable(51)
#define Sine1_px_0startTime_1  Variable(51)
#define Housing_s  Variable(52)
#define Housing_L  Parameter(28)
#define Housing_flangex_0a_s  Variable(53)
#define Housing_flangex_0a_f  Variable(54)
#define Housing_flangex_0b_s  Variable(55)

TranslatedEquations

InitialSection
Sine1_nout = 1;
Sine1_outPort_n = 1;
Sine1_pi = 3.14159265358979;
Force1_inPort_n = 1;
BoundParameterSection
Sine1_px_0amplitude_1 = Sine1_amplitude_1;
Sine1_px_0freqHz_1 = Sine1_freqHz_1;
Sine1_px_0phase_1 = Sine1_phase_1;
Sine1_px_0offset_1 = Sine1_offset_1;
Sine1_px_0startTime_1 = Sine1_startTime_1;
Housing_s = FixedLe_s0+0.5*Housing_L;
PlateRi_flangex_0b_s = Housing_s+0.5*Housing_L;
Friction_flangex_0a_s = FixedLe_s0;
FixedLe_flangex_0b_s = FixedLe_s0;
Housing_flangex_0a_s = FixedLe_s0;
PlateLe_flangex_0a_s = FixedLe_s0;
Housing_flangex_0b_s = PlateRi_flangex_0b_s;
InitialSection
DefaultSection
InitializeData(0)

OutputSection

DynamicsSection

/* schedule using 2 processors */

/* Processor 0 */

proc0(idemand_,icall_,time,X_,XD_,U_,DP_,IP_,LP_,F_,Y_,W_,QZ_,duser_,iuser_,luser_,QiErr);
AcceptedSection1


AcceptedSection2
Sine1_y_1 = Sine1_outPort_signal_1;
Spool_flangex_0b_s = Spool_s+0.5*Spool_L;
FixedLe_flangex_0b_f = PlateLe_flangex_0b_f-Housing_flangex_0a_f+
Friction_flangex_0b_f;
Friction_sx_0rel = Spool_flangex_0a_s-FixedLe_s0;

DefaultSection
InitialSection
/* No equations */
DefaultSection
InitializeData(1)
EndTranslatedEquations

#include <dsblock6.c>

PreNonAlias(0)
StartNonAlias(0)
DeclareVariable("SpringLe.flange_a.s", "absolute position of flange [m]", 0, \
0.0,0.0,0.0,0,0)
DeclareVariable("SpringLe.flange_a.der(s)", "der(absolute position of flange) [m/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareVariable("SpringLe.s_rel", "relative distance (= flange_b.s - flange_a.s) [m]"\
, 0, 0.0,1E+100,0.0,0,0)
DeclareVariable("SpringLe.der(s_rel)", "der(relative distance (= flange_b.s - flange_a.s)) [m/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareParameter("SpringLe.s_rel0", "unstretched spring length [m]", 0, 0, \
0.0,0.0,0.0,0,0)
```

```
DeclareParameter("SpringLe.c", "spring constant [N/m]", 1, 1000000.0, 0.0,1E+100\
,0.0,0,0)
DeclareParameter("SpringLe.d", "damping constant [N/ (m/s)]", 2, 250, 0.0,1E+100\
,0.0,0,0)
DeclareVariable("SpringLe.Contact", "false, if s_rel > l ", false, 0.0,0.0,0.0,0\
,0)
DeclareVariable("SpringRi.flange_b.f", "cut force directed into flange [N]", 0, \
0.0,0.0,0.0,0,0)
DeclareVariable("SpringRi.s_rel", "relative distance (= flange_b.s - flange_a.s) [m]"\
, 0, 0.0,1E+100,0.0,0,0)
DeclareVariable("SpringRi.der(s_rel)", "der(relative distance (= flange_b.s - flange_a.s)) [m/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareParameter("SpringRi.s_rel0", "unstretched spring length [m]", 3, 0, \
0.0,0.0,0.0,0,0)
DeclareParameter("SpringRi.c", "spring constant [N/m]", 4, 1000000.0, 0.0,1E+100\
,0.0,0,0)
DeclareParameter("SpringRi.d", "damping constant [N/ (m/s)]", 5, 250, 0.0,1E+100\
,0.0,0,0)
DeclareVariable("SpringRi.Contact", "false, if s_rel > l ", false, 0.0,0.0,0.0,0\
,0)
DeclareState("Spool.s", "absolute position of center of component (s = flange_a.s + L/2 = flange_b.s - L/2) [m]"\
, 0, 0, 0.0,0.0,0.0,0,0)
DeclareDerivative("Spool.der(s)", "der(absolute position of center of component (s = flange_a.s + L/2 = flange_b.s - L/2)) [m/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareParameter("Spool.L", "length of component from left flange to right flange (= flange_b.s - flange_a.s) [m]"\
, 6, 0.19, 0.0,0.0,0.0,0,0)
DeclareVariable("Spool.flange_a.s", "absolute position of flange [m]", 0, \
0.0,0.0,0.0,0,0)
DeclareVariable("Spool.flange_a.der(s)", "der(absolute position of flange) [m/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareVariable("Spool.flange_a.f", "cut force directed into flange [N]", 0, \
0.0,0.0,0.0,0,0)
DeclareVariable("Spool.flange_b.s", "absolute position of flange [m]", 0, \
0.0,0.0,0.0,0,0)
DeclareParameter("Spool.m", "mass of the sliding mass [kg]", 7, 0.15, 0.0,1E+100\
,0.0,0,0)
DeclareState("Spool.v", "absolute velocity of component [m/s]", 1, 0, 0.0,0.0,\
0.0,0,0)
DeclareDerivative("Spool.der(v)", "der(absolute velocity of component) [m/s/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareParameter("FixedLe.s0", "fixed offset position of housing [m]", 8, (\
-0.0965), 0.0,0.0,0.0,0,0)
DeclareVariable("FixedLe.flange_b.s", "absolute position of flange [m]", 0, \
0.0,0.0,0.0,0,1)
DeclareVariable("FixedLe.flange_b.f", "cut force directed into flange [N]", 0, \
0.0,0.0,0.0,0,0)
DeclareState("PotLe.s", "absolute position of center of component (s = flange_a.s + L/2 = flange_b.s - L/2) [m]"\
, 2, (-0.094), 0.0,0.0,0.0,0,0)
DeclareDerivative("PotLe.der(s)", "der(absolute position of center of component (s = flange_a.s + L/2 = flange_b.s - L/2)) [m/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareParameter("PotLe.L", "length of component from left flange to right flange (= flange_b.s - flange_a.s) [m]"\
, 9, 0.002, 0.0,0.0,0.0,0,0)
DeclareVariable("PotLe.flange_a.s", "absolute position of flange [m]", 0, \
0.0,0.0,0.0,0,0)
DeclareVariable("PotLe.flange_a.der(s)", "der(absolute position of flange) [m/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareVariable("PotLe.flange_a.f", "cut force directed into flange [N]", 0, \
0.0,0.0,0.0,0,0)
DeclareParameter("PotLe.m", "mass of the sliding mass [kg]", 10, 0.01, 0.0,\
1E+100,0.0,0,0)
DeclareState("PotLe.v", "absolute velocity of component [m/s]", 3, 0, 0.0,0.0,\
0.0,0,0)
DeclareDerivative("PotLe.der(v)", "der(absolute velocity of component) [m/s/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareState("PotRi.s", "absolute position of center of component (s = flange_a.s + L/2 = flange_b.s - L/2) [m]"\
, 4, (-0.0655), 0.0,0.0,0.0,0,0)
DeclareDerivative("PotRi.der(s)", "der(absolute position of center of component (s = flange_a.s + L/2 = flange_b.s - L/2)) [m/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareParameter("PotRi.L", "length of component from left flange to right flange (= flange_b.s - flange_a.s) [m]"\
, 11, 0.002, 0.0,0.0,0.0,0,0)
DeclareVariable("PotRi.flange_b.s", "absolute position of flange [m]", 0, \
0.0,0.0,0.0,0,0)
DeclareVariable("PotRi.flange_b.der(s)", "der(absolute position of flange) [m/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareVariable("PotRi.flange_b.f", "cut force directed into flange [N]", 0, \
0.0,0.0,0.0,0,0)
DeclareParameter("PotRi.m", "mass of the sliding mass [kg]", 12, 0.01, 0.0,\
1E+100,0.0,0,0)
DeclareState("PotRi.v", "absolute velocity of component [m/s]", 5, 0, 0.0,0.0,\
0.0,0,0)
DeclareDerivative("PotRi.der(v)", "der(absolute velocity of component) [m/s/s]"\
, 0, 0.0,0.0,0.0,0,0)
```

```
DeclareVariable("Spring.flange_a.s", "absolute position of flange [m]", 0, \
0.0,0.0,0.0,0,0)
DeclareVariable("Spring.flange_b.s", "absolute position of flange [m]", 0, \
0.0,0.0,0.0,0,0)
DeclareVariable("Spring.s_rel", "relative distance (= flange_b.s - flange_a.s) [m]"\
, 0, 0.0,1E+100,0.0,0,0)
DeclareVariable("Spring.f", "forcee between flanges (positive in direction of flange axis R) [N]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareParameter("Spring.s_rel0", "unstretched spring length [m]", 13, 0.025, \
0.0,1E+100,0.0,0,0)
DeclareParameter("Spring.c", "spring constant  [N/m]", 14, 20000.0, 0.0,1E+100,\
0.0,0,0)
DeclareVariable("PlateLe.flange_a.s", "absolute position of flange [m]", 0, \
0.0,0.0,0.0,0,1)
DeclareVariable("PlateLe.flange_b.f", "cut force directed into flange [N]", 0, \
0.0,0.0,0.0,0,0)
DeclareVariable("PlateLe.s_rel", "relative distance (= flange_b.s - flange_a.s) [m]"\
, 0, 0.0,1E+100,0.0,0,0)
DeclareVariable("PlateLe.der(s_rel)", "der(relative distance (= flange_b.s - flange_a.s)) [m/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareParameter("PlateLe.s_rel0", "unstretched spring length [m]", 15, 0.0015, \
0.0,0.0,0.0,0,0)
DeclareParameter("PlateLe.c", "spring constant [N/m]", 16, 1000000.0, 0.0,1E+100\
,0.0,0,0)
DeclareParameter("PlateLe.d", "damping constant [N/ (m/s)]", 17, 250, 0.0,1E+100\
,0.0,0,0)
DeclareVariable("PlateLe.Contact", "false, if s_rel > l ", false, 0.0,0.0,0.0,0,\
0)
DeclareVariable("PlateRi.flange_b.s", "absolute position of flange [m]", 0, \
0.0,0.0,0.0,0,1)
DeclareVariable("PlateRi.s_rel", "relative distance (= flange_b.s - flange_a.s) [m]"\
, 0, 0.0,1E+100,0.0,0,0)
DeclareVariable("PlateRi.der(s_rel)", "der(relative distance (= flange_b.s - flange_a.s)) [m/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareParameter("PlateRi.s_rel0", "unstretched spring length [m]", 18, 0.0015, \
0.0,0.0,0.0,0,0)
DeclareParameter("PlateRi.c", "spring constant [N/m]", 19, 1000000.0, 0.0,1E+100\
,0.0,0,0)
DeclareParameter("PlateRi.d", "damping constant [N/ (m/s)]", 20, 250, 0.0,1E+100\
,0.0,0,0)
DeclareVariable("PlateRi.Contact", "false, if s_rel > l ", false, 0.0,0.0,0.0,0,\
0)
DeclareVariable("Rod.s", "absolute position of center of component (s = flange_a.s + L/2 = flange_b.s - L/2) [m]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareVariable("Rod.der(s)", "der(absolute position of center of component (s = flange_a.s + L/2 = flange_b.s - L/2)) [m/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareParameter("Rod.L", "length of component from left flange to right flange (= flange_b.s - flange_a.s) [m]"\
, 21, 0.0275, 0.0,0.0,0.0,0,0)
DeclareVariable("Rod.flange_b.f", "cut force directed into flange [N]", 0, \
0.0,0.0,0.0,0,0)
DeclareVariable("Friction.flange_a.s", "absolute position of flange [m]", 0, \
0.0,0.0,0.0,0,1)
DeclareVariable("Friction.flange_b.f", "cut force directed into flange [N]", 0, \
0.0,0.0,0.0,0,0)
DeclareVariable("Friction.s_rel", "relative distance (= flange_b.s - flange_a.s) [m]"\
, 0, 0.0,1E+100,0.0,0,0)
DeclareVariable("Friction.der(s_rel)", "der(relative distance (= flange_b.s - flange_a.s)) [m/s]"\
, 0, 0.0,0.0,0.0,0,0)
DeclareParameter("Friction.d", "damping constant [N/ (m/s)] [N/ (m/s)]", 22, \
2500, 0.0,1E+100,0.0,0,0)
DeclareVariable("Force1.inPort.n", "Dimension of signal vector", 1, 0.0,0.0,0.0,\
0,1)
DeclareVariable("Sine1.nout", "Number of outputs", 1, 1.0,1E+100,0.0,0,1)
DeclareVariable("Sine1.outPort.n", "Dimension of signal vector", 1, 0.0,0.0,0.0,\
0,1)
DeclareVariable("Sine1.outPort.signal[1]", "Real output signals", 0, 0.0,0.0,0.0\
,0,0)
DeclareVariable("Sine1.y[1]", "", 0, 0.0,0.0,0.0,0,0)
DeclareParameter("Sine1.amplitude[1]", "Amplitudes of sine waves", 23, 150, \
0.0,0.0,0.0,0,0)
DeclareParameter("Sine1.freqHz[1]", "Frequencies of sine waves [Hz]", 24, 0.01, \
0.0,0.0,0.0,0,0)
DeclareParameter("Sine1.phase[1]", "Phases of sine waves [rad]", 25, 0, 0.0,0.0,\
0.0,0,0)
DeclareParameter("Sine1.offset[1]", "Offsets of output signals", 26, 0, 0.0,0.0,\
0.0,0,0)
DeclareParameter("Sine1.startTime[1]", "Output = offset for time < startTime [s]"\
, 27, 0, 0.0,0.0,0.0,0,0)
DeclareVariable("Sine1.pi", "", 3.14159265358979, 0.0,0.0,0.0,0,1)
DeclareVariable("Sine1.p_amplitude[1]", "", 0, 0.0,0.0,0.0,0,1)
DeclareVariable("Sine1.p_freqHz[1]", "", 0, 0.0,0.0,0.0,0,1)
DeclareVariable("Sine1.p_phase[1]", "", 0, 0.0,0.0,0.0,0,1)
DeclareVariable("Sine1.p_offset[1]", "", 0, 0.0,0.0,0.0,0,1)
DeclareVariable("Sine1.p_startTime[1]", "[s]", 0, 0.0,0.0,0.0,0,1)
DeclareVariable("Housing.s", "absolute position of center of component (s = flange_a.s + L/2 = flange_b.s - L/2) [m]"\
, 0, 0.0,0.0,0.0,0,1)
DeclareParameter("Housing.L", "length of component from left flange to right flange (= flange_b.s - flange_a.s) [m]"\
, 28, 0.0305, 0.0,0.0,0.0,0,0)
DeclareVariable("Housing.flange_a.s", "absolute position of flange [m]", 0, \
0.0,0.0,0.0,0,1)
DeclareVariable("Housing.flange_a.f", "cut force directed into flange [N]", 0, \
0.0,0.0,0.0,0,0)
DeclareVariable("Housing.flange_b.s", "absolute position of flange [m]", 0, \
0.0,0.0,0.0,0,1)
EndNonAlias(0)

PreAlias(0)
StartAlias(0)
DeclareAlias("SpringLe.flange_a.f", "cut force directed into flange [N]", "\
Rod.flange_b.f", 1, 5, 36)
DeclareAlias("SpringLe.flange_b.s", "absolute position of flange [m]", "\
Spool.flange_a.s", 1, 5, 9)
DeclareAlias("SpringLe.flange_b.f", "cut force directed into flange [N]", "\
Rod.flange_b.f", -1, 5, 36)
DeclareAlias("SpringLe.f", "forcee between flanges (positive in direction of flange axis R) [N]"\
, "Rod.flange_b.f", -1, 5, 36)
DeclareAlias("SpringLe.v_rel", "relative velocity between flange L and R [m/s]"\
, "SpringLe.der(s_rel)", 1, 5, 3)
DeclareAlias("SpringRi.flange_a.s", "absolute position of flange [m]", "\
Spool.flange_a.s", 1, 5, 9)
DeclareAlias("SpringRi.flange_a.f", "cut force directed into flange [N]", "\
SpringRi.flange_b.f", -1, 5, 5)
DeclareAlias("SpringRi.flange_b.s", "absolute position of flange [m]", "\
PotLe.flange_a.s", 1, 5, 15)
DeclareAlias("SpringRi.f", "forcee between flanges (positive in direction of flange axis R) [N]"\
, "SpringRi.flange_b.f", 1, 5, 5)
DeclareAlias("SpringRi.v_rel", "relative velocity between flange L and R [m/s]"\
, "SpringRi.der(s_rel)", 1, 5, 7)
DeclareAlias("Spool.flange_b.f", "cut force directed into flange [N]", "\
Sine1.outPort.signal[1]", 1, 5, 44)
DeclareAlias("Spool.a", "absolute acceleration of component [m/s2]", "\
Spool.der(v)", 1, 6, 1)
DeclareAlias("PotLe.flange_b.s", "absolute position of flange [m]", "\
Spring.flange_a.s", 1, 5, 21)
DeclareAlias("PotLe.flange_b.f", "cut force directed into flange [N]", "Spring.f\
", 1, 5, 24)
DeclareAlias("PotLe.a", "absolute acceleration of component [m/s2]", "\
PotLe.der(v)", 1, 6, 3)
DeclareAlias("PotRi.flange_a.s", "absolute position of flange [m]", "\
Spring.flange_b.s", 1, 5, 22)
DeclareAlias("PotRi.flange_a.f", "cut force directed into flange [N]", "Spring.f\
", -1, 5, 24)
DeclareAlias("PotRi.a", "absolute acceleration of component [m/s2]", "\
PotRi.der(v)", 1, 6, 5)
DeclareAlias("Spring.flange_a.f", "cut force directed into flange [N]", "\
Spring.f", -1, 5, 24)
DeclareAlias("Spring.flange_b.f", "cut force directed into flange [N]", "\
Spring.f", 1, 5, 24)
DeclareAlias("PlateLe.flange_a.f", "cut force directed into flange [N]", "\
PlateLe.flange_b.f", -1, 5, 26)
DeclareAlias("PlateLe.flange_b.s", "absolute position of flange [m]", "\
PotLe.flange_a.s", 1, 5, 15)
DeclareAlias("PlateLe.f", "forcee between flanges (positive in direction of flange axis R) [N]"\
, "PlateLe.flange_b.f", 1, 5, 26)
DeclareAlias("PlateLe.v_rel", "relative velocity between flange L and R [m/s]"\
, "PlateLe.der(s_rel)", 1, 5, 28)
DeclareAlias("PlateRi.flange_a.s", "absolute position of flange [m]", "\
PotRi.flange_b.s", 1, 5, 18)
DeclareAlias("PlateRi.flange_a.f", "cut force directed into flange [N]", "\
Housing.flange_a.f", -1, 5, 54)
DeclareAlias("PlateRi.flange_b.f", "cut force directed into flange [N]", "\
Housing.flange_a.f", 1, 5, 54)
DeclareAlias("PlateRi.f", "forcee between flanges (positive in direction of flange axis R) [N]"\
, "Housing.flange_a.f", 1, 5, 54)
DeclareAlias("PlateRi.v_rel", "relative velocity between flange L and R [m/s]"\
, "PlateRi.der(s_rel)", 1, 5, 32)
DeclareAlias("Rod.flange_a.s", "absolute position of flange [m]", "\
SpringLe.flange_a.s", 1, 5, 0)
DeclareAlias("Rod.flange_a.f", "cut force directed into flange [N]", "\
Rod.flange_b.f", -1, 5, 36)
DeclareAlias("Rod.flange_b.s", "absolute position of flange [m]", "\
PotRi.flange_b.s", 1, 5, 18)
DeclareAlias("Friction.flange_a.f", "cut force directed into flange [N]", "\
Friction.flange_b.f", -1, 5, 38)
```

```
DeclareAlias("Friction.flange_b.s", "absolute position of flange [m]", "\
Spool.flange_a.s", 1, 5, 9)
DeclareAlias("Friction.f", "forcee between flanges (positive in direction of flange axis R) [N]"\
, "Friction.flange_b.f", 1, 5, 38)
DeclareAlias("Friction.v_rel", "relative velocity between flange L and R [m/s]"\
, "Friction.der(s_rel)", 1, 5, 40)
DeclareAlias("Force1.f", "driving force [N]", "Sine1.outPort.signal[1]", 1, 5, \
44)
DeclareAlias("Force1.flange_b.s", "absolute position of flange [m]", "\
Spool.flange_b.s", 1, 5, 12)
DeclareAlias("Force1.flange_b.f", "cut force directed into flange [N]", "\
Sine1.outPort.signal[1]", -1, 5, 44)
DeclareAlias("Force1.inPort.signal[1]", "Real input signals", "Sine1.outPort.signal[1]\
", 1, 5, 44)
DeclareAlias("Housing.flange_b.f", "cut force directed into flange [N]", "\
Housing.flange_a.f", -1, 5, 54)
EndAlias(0)
#define NX_    6
#define NX2_   0
#define NU_    0
#define NY_    0
#define NW_    56
#define NP_    29
#define NI_    0
#define NRel_  4
#define NTim_  1
#define NSamp_ 0
#define NCons_ 0
#define NA_    41
#define SizePre_ 0
#define SizeEq_ 0
#define SizeDelay_ 0
#define QNLmax_ 0
#define MAXAux 0
#define NrDymolaTimers_ 0
#define NWhen_ 0

#include <dsblock5.c>

StartDataBlock
EndDataBlock
extern double _mpimsendbuf0[];
extern double _mpimrecvbuf0[];
extern double _mpimsendbuf1[];
extern double _mpimrecvbuf1[];
/* Processor 0 */

void proc0(long *idemand, long *icall_, double *time
 ,double X_[], double XD_[], double U_[], double DP_[]
 ,long IP_[], long LP_[]
, double F_[], double Y_[], double W_[], double QZ_[]
, double duser_[], long iuser_[]
 ,long luser_[],long *QiErr) {
double dlocbuf[7+2*NRel_+1+(3*(NRel_+1))+SizePre_+2*SizeEq_];
long llocbuf[18+2*(NRel_+1)];
extern int inJacobian_;
int i,j;
llocbuf[0]=1;llocbuf[1]=*idemand; llocbuf[2]=*icall_; llocbuf[3]=*QiErr;
llocbuf[4]=Init; llocbuf[5]=Event;llocbuf[6]=PrintEvent;llocbuf[7]=AnyEvent;
llocbuf[8]=Iter;
llocbuf[9]=solverHandleEq_;
llocbuf[10]=inJacobian_;
llocbuf[11]=QInfRev
;llocbuf[12]=QiOpt
;llocbuf[13]=QNnl
;llocbuf[14]=Qinfo
;llocbuf[15]=QNLnr
;llocbuf[16]=QBase
;llocbuf[17]=NewParameters
MSEND(&llocbuf,18,MPI_LONG,1,0);
/* Send to 1 */
i=0;
_mpimsendbuf1[i++]=1.0;
_mpimsendbuf1[i++]=*time;
_mpimsendbuf1[i++]=EPS_;
_mpimsendbuf1[i++]=Time;
_mpimsendbuf1[i++]=EqRemember1Time_;
_mpimsendbuf1[i++]=EqRemember2Time_;
_mpimsendbuf1[i++]=Qtol;
for(j=0;j<SizePre_;j++) _mpimsendbuf1[i++]=QPre_[j];
for(j=0;j<SizeEq_;j++) _mpimsendbuf1[i++]=EqRemember1_[j];
for(j=0;j<SizeEq_;j++) _mpimsendbuf1[i++]=EqRemember2_[j];
```

```
_mpimsendbuf1[0+7+SizePre_+2*SizeEq_]=Spool_m;
_mpimsendbuf1[1+7+SizePre_+2*SizeEq_]=Sine1_px_0offset_1;
_mpimsendbuf1[2+7++SizePre_+2*SizeEq_]=Sine1_px_0amplitude_1;
_mpimsendbuf1[3+7+SizePre_+2*SizeEq_]=Sine1_px_0startTime_1;
_mpimsendbuf1[4+7+SizePre_+2*SizeEq_]=Friction_d;
_mpimsendbuf1[5+7+SizePre_+2*SizeEq_]=SpringLe_sx_0rel0;
_mpimsendbuf1[6+7+SizePre_+2*SizeEq_]=SpringRi_sx_0rel0;
_mpimsendbuf1[7+7+SizePre_+2*SizeEq_]=Sine1_px_0phase_1;
_mpimsendbuf1[8+7+SizePre_+2*SizeEq_]=SpringLe_d;
_mpimsendbuf1[9+7+SizePre_+2*SizeEq_]=SpringLe_c;
_mpimsendbuf1[10+7+SizePre_+2*SizeEq_]=SpringRi_d;
_mpimsendbuf1[11+7+SizePre_+2*SizeEq_]=SpringRi_c;
_mpimsendbuf1[12+7+SizePre_+2*SizeEq_]=Sine1_px_0freqHz_1;
_mpimsendbuf1[13+7+SizePre_+2*SizeEq_]=Spool_v;
_mpimsendbuf1[14+7+SizePre_+2*SizeEq_]=Spool_s;
_mpimsendbuf1[15+7+SizePre_+2*SizeEq_]=PotLe_v;
_mpimsendbuf1[16+7+SizePre_+2*SizeEq_]=PotLe_s;
_mpimsendbuf1[17+7+SizePre_+2*SizeEq_]=Rod_L;
_mpimsendbuf1[18+7+SizePre_+2*SizeEq_]=Spool_L;
_mpimsendbuf1[19+7+SizePre_+2*SizeEq_]=PotLe_L;
_mpimsendbuf1[20+7+SizePre_+2*SizeEq_]=PotRi_v;
_mpimsendbuf1[21+7+SizePre_+2*SizeEq_]=PotRi_s;
_mpimsendbuf1[22+7+SizePre_+2*SizeEq_]=PotRi_L;
MSEND(&_mpimsendbuf1,23+7+SizePre_+2*SizeEq_,MPI_DOUBLE,1,134849120);
tempr[27] = 0.5;
 tempr[26] =  tempr[27] *  PotRi_L;
tempr[25] = 0.5;
tempr[46] = 0.5;
tempr[40] = 0.5;
 PotRi_flangex_0b_s =  PotRi_s +  tempr[26];
 tempr[24] =  tempr[25] *  Rod_L;
tempr[23] = 0.5;
 PotRi_der_s =   PotRi_v;
 tempr[45] =  tempr[46] *  PotLe_L;
 tempr[39] =  tempr[40] *  Spool_L;
 Rod_s =  PotRi_flangex_0b_s -  tempr[24];
 tempr[22] =  tempr[23] *  Rod_L;
 PotRi_flangex_0b_der_s =   PotRi_der_s;
 PotLe_der_s =   PotLe_v;
 PotLe_flangex_0a_s =  PotLe_s -  tempr[45];
 Spool_flangex_0a_s =  Spool_s -  tempr[39];
 Spool_der_s =   Spool_v;
 SpringLe_flangex_0a_s =  Rod_s -  tempr[22];
 Rod_der_s =   PotRi_flangex_0b_der_s;
 PlateLe_sx_0rel =  PotLe_flangex_0a_s -  FixedLe_s0;
 PotLe_flangex_0a_der_s =   PotLe_der_s;
 SpringRi_sx_0rel =  PotLe_flangex_0a_s -  Spool_flangex_0a_s;
 Spool_flangex_0a_der_s =   Spool_der_s;
 SpringLe_sx_0rel =  Spool_flangex_0a_s -  SpringLe_flangex_0a_s;
 SpringLe_flangex_0a_der_s =   Rod_der_s;
 PlateLe_der_sx_0rel =   PotLe_flangex_0a_der_s;
 tempr[44] =  PlateLe_sx_0rel -  PlateLe_sx_0rel0;
 SpringRi_der_sx_0rel =  PotLe_flangex_0a_der_s -  Spool_flangex_0a_der_s;
 tempr[38] =  SpringRi_sx_0rel -  SpringRi_sx_0rel0;
tempr[33] = 0.5;
tempr[31] = 0.5;
 SpringLe_der_sx_0rel =  Spool_flangex_0a_der_s -  SpringLe_flangex_0a_der_s;
 tempr[21] =  SpringLe_sx_0rel -  SpringLe_sx_0rel0;
 PlateRi_sx_0rel =  PlateRi_flangex_0b_s -  PotRi_flangex_0b_s;
tempi[10] = 0;
 tempr[43] =  PlateLe_c *  tempr[44];
 tempr[42] =  PlateLe_d *  PlateLe_der_sx_0rel;
tempi[8] = 1;
 tempr[37] =  SpringRi_c *  tempr[38];
 tempr[36] =  SpringRi_d *  SpringRi_der_sx_0rel;
 tempr[32] =  tempr[33] *  PotRi_L;
 tempr[30] =  tempr[31] *  PotLe_L;
tempi[6] = 2;
 tempr[20] =  SpringLe_c *  tempr[21];
 tempr[19] =  SpringLe_d *  SpringLe_der_sx_0rel;
 PlateRi_der_sx_0rel = -  PotRi_flangex_0b_der_s;
 tempr[5] =  PlateRi_sx_0rel -  PlateRi_sx_0rel0;
PlateLe_Contact =PlateLe_sx_0rel < PlateLe_sx_0rel0;
 tempr[41] =  tempr[43] +  tempr[42];
tempi[9] = 0;
SpringRi_Contact =SpringRi_sx_0rel < SpringRi_sx_0rel0;
 tempr[35] =  tempr[37] +  tempr[36];
tempi[7] = 0;
 Spring_flangex_0b_s =  PotRi_s -  tempr[32];
 Spring_flangex_0a_s =  PotLe_s +  tempr[30];
SpringLe_Contact =SpringLe_sx_0rel < SpringLe_sx_0rel0;
 tempr[18] =  tempr[20] +  tempr[19];
```

```
tempi[5] = 0;
tempi[2] = 3;
 tempr[4] =  PlateRi_c *  tempr[5];
 tempr[3] =  PlateRi_d *  PlateRi_der_sx_0rel;
 PlateLe_flangex_0b_f = IF  PlateLe_Contact THEN  tempr[41] ELSE  tempi[9];
 SpringRi_flangex_0b_f = IF  SpringRi_Contact THEN  tempr[35] ELSE  tempi[7];
 Spring_sx_0rel =  Spring_flangex_0b_s -  Spring_flangex_0a_s;
 tempr[17] = IF  SpringLe_Contact THEN  tempr[18] ELSE  tempi[5];
PlateRi_Contact =PlateRi_sx_0rel < PlateRi_sx_0rel0;
 tempr[2] =  tempr[4] +  tempr[3];
tempi[1] = 0;
 tempr[34] =  PlateLe_flangex_0b_f +  SpringRi_flangex_0b_f;
 tempr[29] =  Spring_sx_0rel -  Spring_sx_0rel0;
 Rod_flangex_0b_f = -  tempr[17];
 Housing_flangex_0a_f = IF  PlateRi_Contact THEN  tempr[2] ELSE  tempi[1];
 PotLe_flangex_0a_f = -  tempr[34];
 Spring_f =  Spring_c *  tempr[29];
 PotRi_flangex_0b_f =  Housing_flangex_0a_f -  Rod_flangex_0b_f;
 tempr[28] =  PotLe_flangex_0a_f +  Spring_f;
 tempr[1] =  PotRi_flangex_0b_f -  Spring_f;
 PotLe_der_v = divmacro( tempr[28],  "PotLe.flange_a.f+Spring.f",  PotLe_m,  "PotLe.m");
 PotRi_der_v = divmacro( tempr[1],  "PotRi.flange_b.f-Spring.f",  PotRi_m,  "PotRi.m");
/* RecvToMaster */
/* Recv to end proc 1 */
MRECV(&_mpimrecvbuf0,26,MPI_DOUBLE,1,39280);
Spool_der_v=_mpimrecvbuf0[0];
Sine1_outPort_signal_1=_mpimrecvbuf0[1];
Spool_flangex_0a_f=_mpimrecvbuf0[2];
Rod_flangex_0b_f=_mpimrecvbuf0[3];
Friction_flangex_0b_f=_mpimrecvbuf0[4];
SpringRi_flangex_0b_f=_mpimrecvbuf0[5];
SpringLe_Contact=_mpimrecvbuf0[6];
Friction_der_sx_0rel=_mpimrecvbuf0[7];
SpringRi_Contact=_mpimrecvbuf0[8];
SpringLe_der_sx_0rel=_mpimrecvbuf0[9];
SpringRi_der_sx_0rel=_mpimrecvbuf0[10];
SpringLe_flangex_0a_der_s=_mpimrecvbuf0[11];
SpringLe_sx_0rel=_mpimrecvbuf0[12];
Spool_flangex_0a_der_s=_mpimrecvbuf0[13];
SpringRi_sx_0rel=_mpimrecvbuf0[14];
PotLe_flangex_0a_der_s=_mpimrecvbuf0[15];
Rod_der_s=_mpimrecvbuf0[16];
SpringLe_flangex_0a_s=_mpimrecvbuf0[17];
Spool_der_s=_mpimrecvbuf0[18];
Spool_flangex_0a_s=_mpimrecvbuf0[19];
PotLe_flangex_0a_s=_mpimrecvbuf0[20];
PotLe_der_s=_mpimrecvbuf0[21];
PotRi_flangex_0b_der_s=_mpimrecvbuf0[22];
Rod_s=_mpimrecvbuf0[23];
PotRi_der_s=_mpimrecvbuf0[24];
PotRi_flangex_0b_s=_mpimrecvbuf0[25];
leave:
finish:
BARRIER;
}
/* Processor 1 */

void proc1() {
long llocbuf[18+2*(NRel_+1)];
extern int inJacobian_;
long *idemand_=&llocbuf[1];
long *icall_=&llocbuf[2];
long *QiErr=&llocbuf[3];
long * QL_=&llocbuf[18];
long * Qenable_=&llocbuf[18+NRel_+1];
double *time=&_mpimrecvbuf1[1];
double X_[NX_],XD_[NX_];
double U_[NU_],Y_[NY_];
double W_[NW_],DP_[NP_];
double F_[NX_];
double *QZ_=&_mpimrecvbuf1[7];
double *QRel_=&_mpimrecvbuf1[7+2*NRel_+1];
double *Qp_=&_mpimrecvbuf1[7+2*NRel_+1+NRel_+1];
double *Qn_=&_mpimrecvbuf1[7+2*NRel_+1+NRel_+1+NRel_+1];
double *QPre_=&_mpimrecvbuf1[7+2*NRel_+1+NRel_+1+NRel_+1+NRel_+1];
double *EqRemember1_=&_mpimrecvbuf1[7+2*NRel_+1+NRel_+1+NRel_+1+NRel_+1+SizePre_];
double *EqRemember2_=&_mpimrecvbuf1[7+2*NRel_+1+NRel_+1+NRel_+1+NRel_+1+SizePre_+SizeEq_];
long *IP_;
bool *LP_;
MRECV(&llocbuf,18+2*(NRel_+1),MPI_LONG,0,0);
Init = llocbuf[4];
Event = llocbuf[5];
```

```
PrintEvent = llocbuf[6];
AnyEvent = llocbuf[7];
Iter = llocbuf[8];
solverHandleEq_ = llocbuf[9];
inJacobian_ = llocbuf[10];
QInfRev=llocbuf[11]
;QiOpt=llocbuf[12];
QNnl=llocbuf[13];
Qinfo=llocbuf[14];
QNLnr=llocbuf[15];
QBase=llocbuf[16];
NewParameters = llocbuf[17];
if ( llocbuf[0] == 0) { MPI_Finalize(); exit(0);}
MRECV(&_mpimrecvbuf1,23+7+2*NRel_+1+(3*(NRel_+1))+SizePre_+2*SizeEq_,MPI_DOUBLE,0,134849120);
EPS_ = _mpimrecvbuf1[2];
Time = _mpimrecvbuf1[3];
EqRemember1Time_=_mpimrecvbuf1[4];
EqRemember2Time_=_mpimrecvbuf1[5];
Qtol=_mpimrecvbuf1[6];
Spool_m=_mpimrecvbuf1[0+7+SizePre_+2*SizeEq_];
Sine1_px_0offset_1=_mpimrecvbuf1[1+7+SizePre_+2*SizeEq_];
Sine1_px_0amplitude_1=_mpimrecvbuf1[2+7+SizePre_+2*SizeEq_];
Sine1_px_0startTime_1=_mpimrecvbuf1[3+7+SizePre_+2*SizeEq_];
Friction_d=_mpimrecvbuf1[4+7+SizePre_+2*SizeEq_];
SpringLe_sx_0rel0=_mpimrecvbuf1[5+7+SizePre_+2*SizeEq_];
SpringRi_sx_0rel0=_mpimrecvbuf1[6+7+SizePre_+2*SizeEq_];
Sine1_px_0phase_1=_mpimrecvbuf1[7+7+SizePre_+2*SizeEq_];
SpringLe_d=_mpimrecvbuf1[8+7+SizePre_+2*SizeEq_];
SpringLe_c=_mpimrecvbuf1[9+7+SizePre_+2*SizeEq_];
SpringRi_d=_mpimrecvbuf1[10+7+SizePre_+2*SizeEq_];
SpringRi_c=_mpimrecvbuf1[11+7+SizePre_+2*SizeEq_];
Sine1_px_0freqHz_1=_mpimrecvbuf1[12+7+SizePre_+2*SizeEq_];
Spool_v=_mpimrecvbuf1[13+7+SizePre_+2*SizeEq_];
Spool_s=_mpimrecvbuf1[14+7+SizePre_+2*SizeEq_];
PotLe_v=_mpimrecvbuf1[15+7+SizePre_+2*SizeEq_];
PotLe_s=_mpimrecvbuf1[16+7+SizePre_+2*SizeEq_];
Rod_L=_mpimrecvbuf1[17+7+2+SizePre_+2*SizeEq_];
Spool_L=_mpimrecvbuf1[18+7+SizePre_+2*SizeEq_];
PotLe_L=_mpimrecvbuf1[19+7+SizePre_+2*SizeEq_];
PotRi_v=_mpimrecvbuf1[20+7+SizePre_+2*SizeEq_];
PotRi_s=_mpimrecvbuf1[21+7+SizePre_+2*SizeEq_];
PotRi_L=_mpimrecvbuf1[22+7+SizePre_+2*SizeEq_];
tempr[27] = 0.5;
 tempr[26] =  tempr[27] *  PotRi_L;
tempr[25] = 0.5;
tempr[46] = 0.5;
tempr[40] = 0.5;
 PotRi_flangex_0b_s =  PotRi_s +  tempr[26];
 tempr[24] =  tempr[25] *  Rod_L;
tempr[23] = 0.5;
 PotRi_der_s =   PotRi_v;
 tempr[45] =  tempr[46] *  PotLe_L;
 tempr[39] =  tempr[40] *  Spool_L;
 Rod_s =  PotRi_flangex_0b_s -  tempr[24];
 tempr[22] =  tempr[23] *  Rod_L;
 PotRi_flangex_0b_der_s =   PotRi_der_s;
 PotLe_flangex_0a_s =  PotLe_s -  tempr[45];
 PotLe_der_s =   PotLe_v;
 Spool_flangex_0a_s =  Spool_s -  tempr[39];
 Spool_der_s =   Spool_v;
 SpringLe_flangex_0a_s =  Rod_s -  tempr[22];
 Rod_der_s =   PotRi_flangex_0b_der_s;
tempr[14] = 6.28318530717959;
 PotLe_flangex_0a_der_s =   PotLe_der_s;
 SpringRi_sx_0rel =  PotLe_flangex_0a_s -  Spool_flangex_0a_s;
 Spool_flangex_0a_der_s =   Spool_der_s;
 SpringLe_sx_0rel =  Spool_flangex_0a_s -  SpringLe_flangex_0a_s;
 SpringLe_flangex_0a_der_s =   Rod_der_s;
 tempr[13] =  tempr[14] *  Sine1_px_0freqHz_1;
 tempr[12] =  Time -  Sine1_px_0startTime_1;
 SpringRi_der_sx_0rel =  PotLe_flangex_0a_der_s -  Spool_flangex_0a_der_s;
 tempr[38] =  SpringRi_sx_0rel -  SpringRi_sx_0rel0;
 SpringLe_der_sx_0rel =  Spool_flangex_0a_der_s -  SpringLe_flangex_0a_der_s;
 tempr[21] =  SpringLe_sx_0rel -  SpringLe_sx_0rel0;
 tempr[11] =  tempr[13] *  tempr[12];
tempi[8] = 1;
 tempr[37] =  SpringRi_c *  tempr[38];
 tempr[36] =  SpringRi_d *  SpringRi_der_sx_0rel;
tempi[6] = 2;
 tempr[20] =  SpringLe_c *  tempr[21];
 tempr[19] =  SpringLe_d *  SpringLe_der_sx_0rel;
 tempr[10] =  tempr[11] +  Sine1_px_0phase_1;
```

```
SpringRi_Contact =SpringRi_sx_0rel < SpringRi_sx_0rel0;
 tempr[35] =  tempr[37] +  tempr[36];
tempi[7] = 0;
 Friction_der_sx_0rel =   Spool_flangex_0a_der_s;
SpringLe_Contact =SpringLe_sx_0rel < SpringLe_sx_0rel0;
 tempr[18] =  tempr[20] +  tempr[19];
tempi[5] = 0;
tempi[4] = 0;
 tempr[9] = sin( tempr[10]);
 SpringRi_flangex_0b_f = IF  SpringRi_Contact THEN  tempr[35] ELSE  tempi[7];
 Friction_flangex_0b_f =  Friction_d *  Friction_der_sx_0rel;
 tempr[17] = IF  SpringLe_Contact THEN  tempr[18] ELSE  tempi[5];
 tempr[15] = LessTime( Sine1_px_0startTime_1,  tempi[4]);
tempi[3] = 0;
 tempr[8] =  Sine1_px_0amplitude_1 *  tempr[9];
 Rod_flangex_0b_f = -  tempr[17];
 tempr[16] =  SpringRi_flangex_0b_f -  Friction_flangex_0b_f;
 tempr[7] = IF  tempr[15] THEN  tempi[3] ELSE  tempr[8];
 Spool_flangex_0a_f =  tempr[16] +  Rod_flangex_0b_f;
 Sine1_outPort_signal_1 =  Sine1_px_0offset_1 +  tempr[7];
 tempr[6] =  Spool_flangex_0a_f +  Sine1_outPort_signal_1;
 Spool_der_v = divmacro( tempr[6],  "Spool.flange_a.f+Sine1.outPort.signal[1]",  Spool_m,  "Spool.m");
/* SendToMaster */
_mpimsendbuf1[0]=Spool_der_v;
_mpimsendbuf1[1]=Sine1_outPort_signal_1;
_mpimsendbuf1[2]=Spool_flangex_0a_f;
_mpimsendbuf1[3]=Rod_flangex_0b_f;
_mpimsendbuf1[4]=Friction_flangex_0b_f;
_mpimsendbuf1[5]=SpringRi_flangex_0b_f;
_mpimsendbuf1[6]=SpringLe_Contact;
_mpimsendbuf1[7]=Friction_der_sx_0rel;
_mpimsendbuf1[8]=SpringRi_Contact;
_mpimsendbuf1[9]=SpringLe_der_sx_0rel;
_mpimsendbuf1[10]=SpringRi_der_sx_0rel;
_mpimsendbuf1[11]=SpringLe_flangex_0a_der_s;
_mpimsendbuf1[12]=SpringLe_sx_0rel;
_mpimsendbuf1[13]=Spool_flangex_0a_der_s;
_mpimsendbuf1[14]=SpringRi_sx_0rel;
_mpimsendbuf1[15]=PotLe_flangex_0a_der_s;
_mpimsendbuf1[16]=Rod_der_s;
_mpimsendbuf1[17]=SpringLe_flangex_0a_s;
_mpimsendbuf1[18]=Spool_der_s;
_mpimsendbuf1[19]=Spool_flangex_0a_s;
_mpimsendbuf1[20]=PotLe_flangex_0a_s;
_mpimsendbuf1[21]=PotLe_der_s;
_mpimsendbuf1[22]=PotRi_flangex_0b_der_s;
_mpimsendbuf1[23]=Rod_s;
_mpimsendbuf1[24]=PotRi_der_s;
_mpimsendbuf1[25]=PotRi_flangex_0b_s;
MSEND(&_mpimsendbuf1,26,MPI_DOUBLE,0,39280);
leave:
finish:
BARRIER;
}

int main (int argc, char *argv[])
{
 int val,i,flag;
 long killbuf[18+2*(NRel_+1)];
 MPI_Init(&argc,&argv);
 MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
 MPI_Pack_size(1,MPI_DOUBLE,MPI_COMM_WORLD,&packedsize);
 if (myrank == 0) {
  val = dymosimMain(argc,argv);
  SENDKILL;
  MPI_Finalize();
   return val;
 }
 dymmdp_(); /* Set up machine constants for NL solving*/
 while (1) {
   switch(myrank) {
   case 1:
    proc1();
    break;
   };
 }
}
double _mpimsendbuf0[26];
double _mpimrecvbuf0[26];
double _mpimsendbuf1[26];
double _mpimrecvbuf1[26];
```

Bibliography

[1] Aisee tool, http://www.aisee.com.

[2] 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.

[3] N. Andersson. Licentiate thesis: *Compilation of Mathematical Models to Parallel Code.* Department of Computer and Information Science, Linköpings Universitet, Sweden, 1996.

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

[5] Peter Aronsson, Peter Fritzson. Static Scheduling of Sequential Java Programs for Multi-Processors. In *JOSES (Java Optimization Strategies for Embedded Systems) Workshop, at ETAPS, 2-6 April, Genova, Italy*, 2001.

[6] M. Ayed, J-L Gaudiot. An efficient heuristic for code partitioning. *Parallel Computing*, 26:399–426, 2000.

[7] L. Blackford and J. Choi and A. Cleary and E. D'Azevedo and J. Demmel and I. Dhillon and J. Dongarra and S. Hammarling and G. Henry and A. Petitet and K. Stanley and D. Walker and R. Whaley. Scalapack users' guide, 1997.

[8] Thomas L. Casavant and Jon G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *Software Engineering*, 14(2):141–154, 1988.

[9] E. Coffman Jr. and R. Graham. Optimal scheduling for two processor systems. *Acta Informatica*, vol. 1(no. 3):200–213, 1972.

[10] 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.

[11] S. Pande S. Darbha. A Robust Compile Time Method for Scheduling Task Parallelism on Distributed Memory Machines. In *Proceedings of PACT'96, October 20-23, Boston, Massachutetts, USA*, pages 156–162, 1996.

[12] 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.

[13] 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.

[14] *Dymola, http://www.dynasim.se.*

[15] H. Elmqvist, M. Otter, and F. Cellier. Inline Integration: A New Mixed Symbolic /Numeric Approach for Solving Differential– Algebraic Equation Systems, Keynote Address, Proc. ESM'95, European Simulation Multiconference, Prague, Czech Republic, June 5–8, 1995.

[16] 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.

[17] 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.

[18] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10th ACM Symposium on Theory of Computing, San Diego, Ca USA*, 1978.

[19] I. Foster. *Designing and Building Parallel Programs.* Addison-Wesley, 1995.

[20] D. Fritzson and P. Nordling. Adaptive scheduling strategy optimizer for parallel roller bearing simulation. *Future Generation Computer Systems*, 16, 2000.

[21] 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.

[22] I.S Duff, A.M Erisman and J.K Reid". *Direct Methods for Sparse Matrices.* Oxford Science Publications, 1989.

[23] B. Kruatrachue and T. Lewis. Grain Size Determination for Parallel Processing. *IEEE Software*, vol. 5(no. 1):23–32, 1988.

[24] B. Kruatrachue. *Static Task Scheduling and Grain Packing in Parallel Processor Systems.* PhD thesis, Dept. of Electrical and Computer Engineering, Oregon State University, 1987.

[25] 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.

[26] 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.

[27] 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.

[28] Zhen Liu. Worst-case analysis of scheduling heuristics of parallel systems. *Parallel Computing*, 24:863–891, 1998.

[29] 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.

[30] M. Gustavsson. Personal Communication. PELAB, Linköping University, Sweden, 2000.

[31] *MathModelica, http://www.mathcore.com.*

[32] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, 1994.

[33] A Migdalas, P. M. Pardalos, and S Storoy. *Parallel Computing in Optimization.* Kluwer Academic Press, 1997.

[34] *The Modelica Language, http://www.modelica.org.*

[35] Modelica Association. *The Modelica Language Specification Version 2.0*, March 2002. http://www.modelica.org.

[36] Myrinet, http://www.myrinet.com.

[37] 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.

[38] 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.

[39] C. S. Park, S. B. Choi. Multiprocessor Scheduling Algorithm Utilizing Linear Clustering of Directed Acyclic Graphs. In *Parallel and Distributed Systems, Proceedings ICPADS'97 December 11-13, Seoul, Korea*, 1997.

[40] V. Paxson. GNU Flex Manual, Version 2.5.3, Free Software Foundation, 1996.

[41] 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.

[42] A. Radulescu and A. van Gemund. On the complexity of list scheduling algorithms for distributed-memory systems. In *ACM International Conference on Supercomputing, Rhodes, Greece*, 1999.

[43] 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.

[44] S. Chingchit, M. Kumar, L.N. Bhuyan. A flexible Clustering and Scheduling Scheme for Efficient Parallel Computation. In *Proceedings, Parallel and Distributed Processing*, pages 500–505. IEEE Computer, 1999. 12-16 April, San Juan, Puerto Rico.

[45] Scali - Scalable Linux Systems, http://www.scali.com.

[46] 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.

[47] 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.

[48] J. D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, 1975.

[49] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, 1989.

[50] L. G- Valliant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[51] VCG tool, http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html.

[52] L. Viklund, J. Herber, and P. Fritzson. The Implementation of Object-Math - a High-Level Programming Environment for Scientific Computing. In *In Proc of CC-92: International Workshop on Compiler Construction, Paderborn, Germany, October 5-7*, 1992.

[53] 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.

[54] 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.

[55] 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.

LINKÖPINGS UNIVERSITET

Avdelning, Institution
Division, department

Institutionen för datavetenskap

Department of Computer and Information Science

Datum
Date

20020404

Språk
Language

[✓ Svenska/Swedish] — no:
☐ Svenska/Swedish
☒ Engelska/English
☐ ____________

Rapporttyp
Report: category

☒ Licentiatavhandling
☐ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ ____________

ISBN 91-7373-280-X

ISRN LiU-Tek-Lic- 2002:06

Serietitel och serienummer
Title of series, numbering

ISSN 0280-7971

Linköping Studies in Science and Technology

Thesis No. 933

URL för elektronisk version

Titel
Title

Automatic Parallelization of Simulation Code from Equation Based Simulation Languages

Författare
Author

Peter Aronsson

Sammandrag
Abstract

Modern state-of-the-art equation based object oriented modeling languages such as Modelica have enabled easy modeling of large and complex physical systems. When such complex models are to be simulated, simulation tools typically perform a number of optimizations on the underlying set of equations in the modeled system, with the goal of gaining better simulation performance by decreasing the equation system size and complexity. The tools then typically generate efficient code to obtain fast execution of the simulations. However, with increasing complexity of modeled systems the number of equations and variables are increasing. Therefore, to be able to simulate these large complex systems in an efficient way parallel computing can be exploited.

This thesis presents the work of building an automatic parallelization tool that produces an efficient parallel version of the simulation code by building a data dependency graph (task graph) from the simulation code and applying efficient scheduling and clustering algorithms on the task graph. Various scheduling and clustering algorithms, adapted for the requirements from this type of simulation code, have been implemented and evaluated. The scheduling and clustering algorithms presented and evaluated can also be used for functional dataflow languages in general, since the algorithms work on a task graph with dataflow edges between nodes.

Results are given in form of speedup measurements and task graph statistics produced by the tool. The conclusion drawn is that some of the algorithms investigated and adapted in this work give reasonable measured speedup results for some specific Modelica models, e.g. a model of a thermofluid pipe gave a speedup of about 2.5 on 8 processors in a PC-cluster. However, future work lies in finding a good algorithm that works well in general.

Nyckelord
Keywords

Clustering, Modelica, Scheduling, Parallellization

Department of Computer and Information Science
Linköpings universitet

Linköping Studies in Science and Technology
Faculty of Arts and Sciences - Licentiate Theses

No 17	**Vojin Plavsic:** Interleaved Processing of Non-Numerical Data Stored on a Cyclic Memory. (Available at: FOA, Box 1165, S-581 11 Linköping, Sweden. FOA Report B30062E)
No 28	**Arne Jönsson, Mikael Patel:** An Interactive Flowcharting Technique for Communicating and Realizing Algorithms, 1984.
No 29	**Johnny Eckerland:** Retargeting of an Incremental Code Generator, 1984.
No 48	**Henrik Nordin:** On the Use of Typical Cases for Knowledge-Based Consultation and Teaching, 1985.
No 52	**Zebo Peng:** Steps Towards the Formalization of Designing VLSI Systems, 1985.
No 60	**Johan Fagerström:** Simulation and Evaluation of Architecture based on Asynchronous Processes, 1985.
No 71	**Jalal Maleki:** ICONStraint, A Dependency Directed Constraint Maintenance System, 1987.
No 72	**Tony Larsson:** On the Specification and Verification of VLSI Systems, 1986.
No 73	**Ola Strömfors:** A Structure Editor for Documents and Programs, 1986.
No 74	**Christos Levcopoulos:** New Results about the Approximation Behavior of the Greedy Triangulation, 1986.
No 104	**Shamsul I. Chowdhury: S**tatistical Expert Systems - a Special Application Area for Knowledge-Based Computer Methodology, 1987.
No 108	**Rober Bilos:** Incremental Scanning and Token-Based Editing, 1987.
No 111	**Hans Block:** SPORT-SORT Sorting Algorithms and Sport Tournaments, 1987.
No 113	**Ralph Rönnquist:** Network and Lattice Based Approaches to the Representation of Knowledge, 1987.
No 118	**Mariam Kamkar, Nahid Shahmehri:** Affect-Chaining in Program Flow Analysis Applied to Queries of Programs, 1987.
No 126	**Dan Strömberg:** Transfer and Distribution of Application Programs, 1987.
No 127	**Kristian Sandahl:** Case Studies in Knowledge Acquisition, Migration and User Acceptance of Expert Systems, 1987.
No 139	**Christer Bäckström:** Reasoning about Interdependent Actions, 1988.
No 140	**Mats Wirén:** On Control Strategies and Incrementality in Unification-Based Chart Parsing, 1988.
No 146	**Johan Hultman:** A Software System for Defining and Controlling Actions in a Mechanical System, 1988.
No 150	**Tim Hansen:** Diagnosing Faults using Knowledge about Malfunctioning Behavior, 1988.
No 165	**Jonas Löwgren:** Supporting Design and Management of Expert System User Interfaces, 1989.
No 166	**Ola Petersson:** On Adaptive Sorting in Sequential and Parallel Models, 1989.
No 174	**Yngve Larsson:** Dynamic Configuration in a Distributed Environment, 1989.
No 177	**Peter Åberg:** Design of a Multiple View Presentation and Interaction Manager, 1989.
No 181	**Henrik Eriksson:** A Study in Domain-Oriented Tool Support for Knowledge Acquisition, 1989.
No 184	**Ivan Rankin:** The Deep Generation of Text in Expert Critiquing Systems, 1989.
No 187	**Simin Nadjm-Tehrani:** Contributions to the Declarative Approach to Debugging Prolog Programs, 1989.
No 189	**Magnus Merkel:** Temporal Information in Natural Language, 1989.
No 196	**Ulf Nilsson:** A Systematic Approach to Abstract Interpretation of Logic Programs, 1989.
No 197	**Staffan Bonnier:** Horn Clause Logic with External Procedures: Towards a Theoretical Framework, 1989.
No 203	**Christer Hansson:** A Prototype System for Logical Reasoning about Time and Action, 1990.
No 212	**Björn Fjellborg:** An Approach to Extraction of Pipeline Structures for VLSI High-Level Synthesis, 1990.
No 230	**Patrick Doherty:** A Three-Valued Approach to Non-Monotonic Reasoning, 1990.
No 237	**Tomas Sokolnicki:** Coaching Partial Plans: An Approach to Knowledge-Based Tutoring, 1990.
No 250	**Lars Strömberg:** Postmortem Debugging of Distributed Systems, 1990.
No 253	**Torbjörn Näslund:** SLDFA-Resolution - Computing Answers for Negative Queries, 1990.
No 260	**Peter D. Holmes:** Using Connectivity Graphs to Support Map-Related Reasoning, 1991.
No 283	**Olof Johansson:** Improving Implementation of Graphical User Interfaces for Object-Oriented Knowledge-Bases, 1991.
No 298	**Rolf G Larsson:** Aktivitetsbaserad kalkylering i ett nytt ekonomisystem, 1991.
No 318	**Lena Srömbäck:** Studies in Extended Unification-Based Formalism for Linguistic Description: An Algorithm for Feature Structures with Disjunction and a Proposal for Flexible Systems, 1992.
No 319	**Mikael Pettersson:** DML-A Language and System for the Generation of Efficient Compilers from Denotational Specification, 1992.
No 326	**Andreas Kågedal:** Logic Programming with External Procedures: an Implementation, 1992.
No 328	**Patrick Lambrix:** Aspects of Version Management of Composite Objects, 1992.
No 333	**Xinli Gu:** Testability Analysis and Improvement in High-Level Synthesis Systems, 1992.
No 335	**Torbjörn Näslund:** On the Role of Evaluations in Iterative Development of Managerial Support Sytems, 1992.
No 348	**Ulf Cederling:** Industrial Software Development - a Case Study, 1992.
No 352	**Magnus Morin:** Predictable Cyclic Computations in Autonomous Systems: A Computational Model and Implementation, 1992.
No 371	**Mehran Noghabai:** Evaluation of Strategic Investments in Information Technology, 1993.
No 378	**Mats Larsson:** A Transformational Approach to Formal Digital System Design, 1993.
No 380	**Johan Ringström:** Compiler Generation for Parallel Languages from Denotational Specifications, 1993.
No 381	**Michael Jansson:** Propagation of Change in an Intelligent Information System, 1993.
No 383	**Jonni Harrius:** An Architecture and a Knowledge Representation Model for Expert Critiquing Systems, 1993.
No 386	**Per Österling:** Symbolic Modelling of the Dynamic Environments of Autonomous Agents, 1993.
No 398	**Johan Boye:** Dependency-based Groudness Analysis of Functional Logic Programs, 1993.

No 402	**Lars Degerstedt:** Tabulated Resolution for Well Founded Semantics, 1993.
No 406	**Anna Moberg:** Satellitkontor - en studie av kommunikationsmönster vid arbete på distans, 1993.
No 414	**Peter Carlsson:** Separation av företagsledning och finansiering - fallstudier av företagsledarutköp ur ett agent-teoretiskt perspektiv, 1994.
No 417	**Camilla Sjöström:** Revision och lagreglering - ett historiskt perspektiv, 1994.
No 436	**Cecilia Sjöberg:** Voices in Design: Argumentation in Participatory Development, 1994.
No 437	**Lars Viklund:** Contributions to a High-level Programming Environment for a Scientific Computing, 1994.
No 440	**Peter Loborg:** Error Recovery Support in Manufacturing Control Systems, 1994.
FHS 3/94	**Owen Eriksson:** Informationssystem med verksamhetskvalitet - utvärdering baserat på ett verksamhetsinriktat och samskapande perspektiv, 1994.
FHS 4/94	**Karin Pettersson:** Informationssystemstrukturering, ansvarsfördelning och användarinflytande - En komparativ studie med utgångspunkt i två informationssystemstrategier, 1994.
No 441	**Lars Poignant:** Informationsteknologi och företagsetablering - Effekter på produktivitet och region, 1994.
No 446	**Gustav Fahl:** Object Views of Relational Data in Multidatabase Systems, 1994.
No 450	**Henrik Nilsson:** A Declarative Approach to Debugging for Lazy Functional Languages, 1994.
No 451	**Jonas Lind:** Creditor - Firm Relations: an Interdisciplinary Analysis, 1994.
No 452	**Martin Sköld:** Active Rules based on Object Relational Queries - Efficient Change Monitoring Techniques, 1994.
No 455	**Pär Carlshamre:** A Collaborative Approach to Usability Engineering: Technical Communicators and System Developers in Usability-Oriented Systems Development, 1994.
FHS 5/94	**Stefan Cronholm:** Varför CASE-verktyg i systemutveckling? - En motiv- och konsekvensstudie avseende arbetssätt och arbetsformer, 1994.
No 462	**Mikael Lindvall:** A Study of Traceability in Object-Oriented Systems Development, 1994.
No 463	**Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av Sandviks förvärv av Bahco Verktyg, 1994.
No 464	**Hans Olsén:** Collage Induction: Proving Properties of Logic Programs by Program Synthesis, 1994.
No 469	**Lars Karlsson:** Specification and Synthesis of Plans Using the Features and Fluents Framework, 1995.
No 473	**Ulf Söderman:** On Conceptual Modelling of Mode Switching Systems, 1995.
No 475	**Choong-ho Yi:** Reasoning about Concurrent Actions in the Trajectory Semantics, 1995.
No 476	**Bo Lagerström:** Successiv resultatavräkning av pågående arbeten. - Fallstudier i tre byggföretag, 1995.
No 478	**Peter Jonsson:** Complexity of State-Variable Planning under Structural Restrictions, 1995.
FHS 7/95	**Anders Avdic:** Arbetsintegrerad systemutveckling med kalkylkprogram, 1995.
No 482	**Eva L Ragnemalm:** Towards Student Modelling through Collaborative Dialogue with a Learning Companion, 1995.
No 488	**Eva Toller:** Contributions to Parallel Multiparadigm Languages: Combining Object-Oriented and Rule-Based Programming, 1995.
No 489	**Erik Stoy:** A Petri Net Based Unified Representation for Hardware/Software Co-Design, 1995.
No 497	**Johan Herber:** Environment Support for Building Structured Mathematical Models, 1995.
No 498	**Stefan Svenberg:** Structure-Driven Derivation of Inter-Lingual Functor-Argument Trees for Multi-Lingual Generation, 1995.
No 503	**Hee-Cheol Kim:** Prediction and Postdiction under Uncertainty, 1995.
FHS 8/95	**Dan Fristedt:** Metoder i användning - mot förbättring av systemutveckling genom situationell metodkunskap och metodanalys, 1995.
FHS 9/95	**Malin Bergvall:** Systemförvaltning i praktiken - en kvalitativ studie avseende centrala begrepp, aktiviteter och ansvarsroller, 1995.
No 513	**Joachim Karlsson:** Towards a Strategy for Software Requirements Selection, 1995.
No 517	**Jakob Axelsson:** Schedulability-Driven Partitioning of Heterogeneous Real-Time Systems, 1995.
No 518	**Göran Forslund:** Toward Cooperative Advice-Giving Systems: The Expert Systems Experience, 1995.
No 522	**Jörgen Andersson:** Bilder av småföretagares ekonomistyrning, 1995.
No 538	**Staffan Flodin:** Efficient Management of Object-Oriented Queries with Late Binding, 1996.
No 545	**Vadim Engelson:** An Approach to Automatic Construction of Graphical User Interfaces for Applications in Scientific Computing, 1996.
No 546	**Magnus Werner :** Multidatabase Integration using Polymorphic Queries and Views, 1996.
FiF-a 1/96	**Mikael Lind:** Affärsprocessinriktad förändringsanalys - utveckling och tillämpning av synsätt och metod, 1996.
No 549	**Jonas Hallberg:** High-Level Synthesis under Local Timing Constraints, 1996.
No 550	**Kristina Larsen:** Förutsättningar och begränsningar för arbete på distans - erfarenheter från fyra svenska företag. 1996.
No 557	**Mikael Johansson:** Quality Functions for Requirements Engineering Methods, 1996.
No 558	**Patrik Nordling:** The Simulation of Rolling Bearing Dynamics on Parallel Computers, 1996.
No 561	**Anders Ekman:** Exploration of Polygonal Environments, 1996.
No 563	**Niclas Andersson:** Compilation of Mathematical Models to Parallel Code, 1996.
No 567	**Johan Jenvald:** Simulation and Data Collection in Battle Training, 1996.
No 575	**Niclas Ohlsson:** Software Quality Engineering by Early Identification of Fault-Prone Modules, 1996.
No 576	**Mikael Ericsson:** Commenting Systems as Design Support—A Wizard-of-Oz Study, 1996.
No 587	**Jörgen Lindström:** Chefers användning av kommunikationsteknik, 1996.
No 589	**Esa Falkenroth:** Data Management in Control Applications - A Proposal Based on Active Database Systems, 1996.
No 591	**Niclas Wahllöf:** A Default Extension to Description Logics and its Applications, 1996.
No 595	**Annika Larsson:** Ekonomisk Styrning och Organisatorisk Passion - ett interaktivt perspektiv, 1997.
No 597	**Ling Lin:** A Value-based Indexing Technique for Time Sequences, 1997.

No 598	**Rego Granlund:** C^{3}Fire - A Microworld Supporting Emergency Management Training, 1997.
No 599	**Peter Ingels:** A Robust Text Processing Technique Applied to Lexical Error Recovery, 1997.
No 607	**Per-Arne Persson:** Toward a Grounded Theory for Support of Command and Control in Military Coalitions, 1997.
No 609	**Jonas S Karlsson:** A Scalable Data Structure for a Parallel Data Server, 1997.
FiF-a 4	**Carita Åbom:** Videomötesteknik i olika affärssituationer - möjligheter och hinder, 1997.
FiF-a 6	**Tommy Wedlund**: Att skapa en företagsanpassad systemutvecklingsmodell - genom rekonstruktion, värdering och vidareutveckling i T50-bolag inom ABB, 1997.
No 615	**Silvia Coradeschi**: A Decision-Mechanism for Reactive and Coordinated Agents, 1997.
No 623	**Jan Ollinen:** Det flexibla kontorets utveckling på Digital - Ett stöd för multiflex? 1997.
No 626	**David Byers:** Towards Estimating Software Testability Using Static Analysis, 1997.
No 627	**Fredrik Eklund:** Declarative Error Diagnosis of GAPLog Programs, 1997.
No 629	**Gunilla Ivefors:** Krigsspel coh Informationsteknik inför en oförutsägbar framtid, 1997**.**
No 631	**Jens-Olof Lindh:** Analysing Traffic Safety from a Case-Based Reasoning Perspective, 1997
No 639	**Jukka Mäki-Turja:**. Smalltalk - a suitable Real-Time Language, 1997.
No 640	**Juha Takkinen**: CAFE: Towards a Conceptual Model for Information Management in Electronic Mail, 1997.
No 643	**Man Lin**: Formal Analysis of Reactive Rule-based Programs, 1997.
No 653	**Mats Gustafsson**: Bringing Role-Based Access Control to Distributed Systems, 1997.
FiF-a 13	**Boris Karlsson:** Metodanalys för förståelse och utveckling av systemutvecklingsverksamhet. Analys och värdering av systemutvecklingsmodeller och dess användning, 1997.
No 674	**Marcus Bjäreland:** Two Aspects of Automating Logics of Action and Change - Regression and Tractability, 1998.
No 676	**Jan Håkegård**: Hiera rchical Test Architecture and Board-Level Test Controller Synthesis, 1998.
No 668	**Per-Ove Zetterlund**: Normering av svensk redovisning - En studie av tillkomsten av Redovisningsrådets rekommendation om koncernredovisning (RR01:91), 1998.
No 675	**Jimmy Tjäder**: Projektledaren & planen - en studie av projektledning i tre installations- och systemutvecklingsprojekt, 1998.
FiF-a 14	**Ulf Melin**: Informationssystem vid ökad affärs- och processorientering - egenskaper, strategier och utveckling, 1998.
No 695	**Tim Heyer**: COMPASS: Introduction of Formal Methods in Code Development and Inspection, 1998.
No 700	**Patrik Hägglund:** Programming Languages for Computer Algebra, 1998.
FiF-a 16	**Marie-Therese Christiansson:** Inter-organistorisk verksamhetsutveckling - metoder som stöd vid utveckling av partnerskap och informationssystem, 1998.
No 712	**Christina Wennestam:** Information om immateriella resurser. Investeringar i forskning och utveckling samt i personal inom skogsindustrin, 1998.
No 719	**Joakim Gustafsson:** Extending Temporal Action Logic for Ramification and Concurrency, 1998.
No 723	**Henrik André-Jönsson:** Indexing time-series data using text indexing methods, 1999.
No 725	**Erik Larsson:** High-Level Testability Analysis and Enhancement Techniques, 1998.
No 730	**Carl-Johan Westin:** Informationsförsörjning: en fråga om ansvar - aktiviteter och uppdrag i fem stora svenska organisationers operativa informationsförsörjning, 1998.
No 731	**Åse Jansson:** Miljöhänsyn - en del i företags styrning, 1998.
No 733	**Thomas Padron-McCarthy:** Performance-Polymorphic Declarative Queries, 1998.
No 734	**Anders Bäckström:** Värdeskapande kreditgivning - Kreditriskhantering ur ett agentteoretiskt perspektiv, 1998.
FiF-a 21	**Ulf Seigerroth:** Integration av förändringsmetoder - en modell för välgrundad metodintegration, 1999.
FiF-a 22	**Fredrik Öberg:** Object-Oriented Frameworks - A New Strategy for Case Tool Development, 1998.
No 737	**Jonas Mellin:** Predictable Event Monitoring, 1998.
No 738	**Joakim Eriksson:** Specifying and Managing Rules in an Active Real-Time Database System, 1998.
FiF-a 25	**Bengt E W Andersson:** Samverkande informationssystem mellan aktörer i offentliga åtaganden - En teori om aktörsarenor i samverkan om utbyte av information, 1998.
No 742	**Pawel Pietrzak:** Static Incorrectness Diagnosis of CLP (FD), 1999.
No 748	**Tobias Ritzau:** Real-Time Reference Counting in RT-Java, 1999.
No 751	**Anders Ferntoft:** Elektronisk affärskommunikation - kontaktkostnader och kontaktprocesser mellan kunder och leverantörer på producentmarknader,1999.
No 752	**Jo Skåmedal:** Arbete på distans och arbetsformens påverkan på resor och resmönster, 1999.
No 753	**Johan Alvehus:** Mötets metaforer. En studie av berättelser om möten, 1999.
No 754	**Magnus Lindahl:** Bankens villkor i låneavtal vid kreditgivning till högt belånade företagsförvärv: En studie ur ett agentteoretiskt perspektiv, 2000.
No 766	**Martin V. Howard:** Designing dynamic visualizations of temporal data, 1999.
No 769	**Jesper Andersson:** Towards Reactive Software Architectures, 1999.
No 775	**Anders Henriksson:** Unique kernel diagnosis, 1999.
FiF-a 30	**Pär J. Ågerfalk:** Pragmatization of Information Systems - A Theoretical and Methodological Outline, 1999.
No 787	**Charlotte Björkegren:** Learning for the next project - Bearers and barriers in knowledge transfer within an organisation, 1999.
No 788	**Håkan Nilsson:** Informationsteknik som drivkraft i granskningsprocessen - En studie av fyra revisionsbyråer, 2000.
No 790	**Erik Berglund:** Use-Oriented Documentation in Software Development, 1999.
No 791	**Klas Gäre:** Verksamhetsförändringar i samband med IS-införande, 1999.
No 800	**Anders Subotic:** Software Quality Inspection, 1999.
No 807	**Svein Bergum**: Managerial communication in telework, 2000.

No 809	**Flavius Gruian:** Energy-Aware Design of Digital Systems, 2000.
FiF-a 32	**Karin Hedström:** Kunskapsanvändning och kunskapsutveckling hos verksamhetskonsulter - Erfarenheter från ett FOU-samarbete, 2000.
No 808	**Linda Askenäs:** Affärssystemet - En studie om teknikens aktiva och passiva roll i en organisation, 2000.
No 820	**Jean Paul Meynard:** Control of industrial robots through high-level task programming, 2000.
No 823	**Lars Hult:** Publika Gränsytor - ett designexempel, 2000.
No 832	**Paul Pop:** Scheduling and Communication Synthesis for Distributed Real-Time Systems, 2000.
FiF-a 34	**Göran Hultgren:** Nätverksinriktad Förändringsanalys - perspektiv och metoder som stöd för förståelse och utveckling av affärsrelationer och informationssystem, 2000.
No 842	**Magnus Kald:** The role of management control systems in strategic business units, 2000.
No 844	**Mikael Cäker:** Vad kostar kunden? Modeller för intern redovisning, 2000.
FiF-a 37	**Ewa Braf**: Organisationers kunskapsverksamheter - en kritisk studie av ”knowledge management”, 2000.
FiF-a 40	**Henrik Lindberg:** Webbaserade affärsprocesser - Möjligheter och begränsningar, 2000.
FiF-a 41	**Benneth Christiansson:** Att komponentbasera informationssystem - Vad säger teori och praktik?, 2000.
No. 854	**Ola Pettersson:** Deliberation in a Mobile Robot, 2000.
No 863	**Dan Lawesson**: Towards Behavioral Model Fault Isolation for Object Oriented Control Systems, 2000.
No 881	**Johan Moe:** Execution Tracing of Large Distributed Systems, 2001.
No 882	**Yuxiao Zhao:** XML-based Frameworks for Internet Commerce and an Implementation of B2B e-procurement, 2001.
No 890	**Annika Flycht-Eriksson:** Domain Knowledge Management inInformation-providing Dialogue systems, 2001.
Fif-a 47	**Per-Arne Segerkvis**t: Webbaserade imaginära organisationers samverkansformer, 2001.
No 894	**Stefan Svarén:** Styrning av investeringar i divisionaliserade företag - Ett koncernperspektiv, 2001.
No 906	**Lin Han**: Secure and Scalable E-Service Software Delivery, 2001.
No 917	**Emma Hansson:** Optionsprogram för anställda - en studie av svenska börsföretag, 2001.
Fif-a-49	**Stefan Holgersson:** IT-system och filtrering av verksamhetskunskap - kvalitetsproblem vid analyser och beslutsfattande som bygger på uppgifter hämtade från polisens IT-system, 2001.
Fif-a-51	**Per Oscarsson:**Informationssäkerhet i verksamheter - begrepp och modeller som stöd för förståelse av informationssäkerhet och dess hantering, 2001.
No 919	**Luis Alejandro Cortes:** A Petri Net Based Modeling and Verification Technique for Real-Time Embedded Systems, 2001.
No 915	**Niklas Sandell:** Redovisning i skuggan av en bankkris - Värdering av fastigheter. 2001.
No 931	**Fredrik Elg:** Ett dynamiskt perspektiv på individuella skillnader av heuristisk kompetens, intelligens, mentala modeller, mål och konfidens i kontroll av mikrovärlden Moro, 2002.
No 933	**Peter Aronsson:** Automatic Parallelization of Simulation Code from Equation Based Simulation Languages, 2002.