Aronsson P., Fritzson P.:

Multiprocessor Scheduling of Simulation Code From Modelica Models

2nd International Modelica Conference, Proceedings, pp. 331-338

Paper presented at the 2nd International Modelica Conference, March 18-19, 2002,
Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR), Oberpfaffenhofen, Germany.

All papers of this workshop can be downloaded from
http://www.Modelica.org/Conference2002/papers.shtml

Program Committee:

- Martin Otter, Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR), Institut für Robotik und Mechatronik, Oberpfaffenhofen, Germany (chairman of the program committee).
- Hilding Elmqvist, Dynasim AB, Lund, Sweden.
- Peter Fritzson, PELAB, Department of Computer and Information Science, Linköping University, Sweden.

Local organizers:

Martin Otter, Astrid Jaschinski, Christian Schweiger, Erika Woeller, Johann Bals,
Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR), Institut für Robotik und
Mechatronik, Oberpfaffenhofen, Germany

Multiprocessor Scheduling of Simulation Code From Modelica Models

Peter Aronsson, Peter Fritzson
PELAB, The Programming Environment lab
Department of Computer and Information Science
Linköping University
SE-581 83 Linköping, Sweden
{petar,petfr}@ida.liu.se

Abstract

Modern object oriented modeling techniques, such as the Modelica modeling language, are increasing the capability to model and simulate systems of large size and complexity. Simulation of such large and complex systems is computationally very expensive. The use of parallel computers for simulation of Modelica models is one approach of handling simulation of such large and complex systems within reasonable time limits. This paper presents an automatic parallelization tool that translates the sequential simulation code generated from a Modelica compiler, Dymola, into a parallel version that can be executed on a parallel computer. The paper also presents several scheduling and clustering techniques used by the tool to partition the simulation code onto several processors. One of these techniques, called FTDT-Full Task Duplication Technique, gives a measured speedup of 2.5 on an 8 processor PC-cluster. However, future work includes developing better scheduling and clustering algorithms to further improve the results.

1 Summary

Object oriented equation based modeling languages such as Modelica enable simulation of large complex systems. However, with growing complexity of modeled systems, the need for parallelization becomes increasingly important in order to keep simulation time within reasonable limits.

The first step in a Modelica compilation results in a system of differential and algebraic equations. The Modelica compiler typically performs optimizations on this system of equations to reduce its size. Other optimizations on the equation system are also performed to for instance reduce the index of the system to make it easier to solve numerically, and break algebraic loops to enable generation of more efficient code. Finally, sequential C code is generated from the optimized set of equations, containing assignment statements with arithmetic expressions, function calls, and subsystems of equations that are solved using a variety of solution techniques. This simulation code is then combined with a numerical solver to simulate the model.

This paper presents an automatic parallelization method and tool that builds a task graph from the optimized sequential code produced by the Dymola commercial Modelica compiler. Earlier work indicated that the task graph should be built at the expression level, resulting in a large fine grained task graph. The reason for building the task graph at the lowest level is to reveal all possible parallelism in the task graph. The fine granularity of the task graph means that the communication costs between tasks in the graph are typically much larger than the execution costs of the tasks. Hence, the scheduling algorithms need to take this into consideration to be able to produce an efficient parallel schedule of the task graph.

Several scheduling algorithms have been studied and implemented for this problem, like the TDS algorithm, which is a task duplication based scheduling algorithm. We have also investigated clustering algorithms which have the goal of clustering nodes for better computation/communication tradeoff. However the standard algorithms found in the literature gave poor result due to the special properties of the tasks graphs generated from the optimized equations converted to C-code emitted by the Dymola Modelica compiler.

There are some scheduling algorithms, specially designed for targeting simulation code, like for instance the algorithm presented in [21]. However, that algorithm is not suitable for our purposes since it is mainly

designed for coarse grained task graphs and can not handle fine grained task graphs well. The reason for obtaining good speedup in that case is that the used architecture had a reasonable fast communication network compared to the slow processor speed. However, this relation between communication speed and processor speed is not valid today, and is in fact degrading in the future, since the processor speed is increasing much faster than the communication speed.

Yet another approach, where task duplication is always used, FTDT - Full Task Duplication Technique, shows speedup results for some examples, including a thermofluid pipe model which gives a speedup of about 2.5 on 8 processors running on a PC-cluster.

Future work include designing and developing better clustering and scheduling algorithms well suited for the simulation code generated from optimized systems of equations.

2 Introduction

Modelica is an acausal, object-oriented, equation based modeling language for modeling and simulation of large and complex multi-domain systems [14, 8]. Modelica was designed by an international team of researchers, whose joint effort has resulted in a general language for design of models of physical multi-domain systems. Modelica has influences from a number of earlier object oriented modeling languages, for instance Dymola [7] and ObjectMath [9].

A Modelica compiler flattens the object oriented structure of the model into a system of differential algebraic equations (DAE) which during simulation is solved using a standard DAE solver. This code is often very time consuming to execute, and there is a great need for parallel execution, especially for demanding applications like hardware-in-the-loop simulation.

The flat set of equations produced by a Modelica compiler is typically sparse, and there is a large opportunity for optimization. A simulation tool with support for the Modelica language would typically perform optimizations on the equation set to reduce the number of equations. One such tool is Dymola [6], another is MathModelica [13].

The problem presented in this paper is to parallelize the calculation of the states (the state variables and their derivatives) in each time step of the solver. The code for this calculation consists of assignments of numerical expressions, e.g. addition or multiplication operations, to different variables. But it can also contain function calls, for instance to solve an equation system or to compute `sin(x)` for a value `x`, which are computationally more heavy tasks. The Dymola simulation tool produces this kind of code. Hence we can use Dymola as a front end for our automatic parallelization tool.

To parallelize the simulation we first build a task graph, $G = (V, E)$ where each task $v \in V$ corresponds to a simple binary operation, or a function call. A data dependency edge ($e \in E$) is present between two task nodes v_1, v_2 iff v_2 uses the result from v_1. This is represented in the task graph by the edge $e = (v_1, v_2)$. Each task is assigned an execution cost which corresponds to a normalized execution time of the task, and each edge is assigned a communication cost corresponding to a normalized communication time between the tasks if they execute on different processors. Figure 1 illustrates how a task graph can be represented graphically. Each node is divided by a horizontal line. Above the line a unique task label/number is given and below the line is the execution cost of the task. Near each edge is the communication cost for the edge given. The goal of a scheduling or clustering algorithm is to minimize the execution time of the parallel program. This often means that the communication between processors must be kept low, since interprocessor communication is very expensive. When two tasks execute on the same processor, the communication cost between them is reduced to zero.

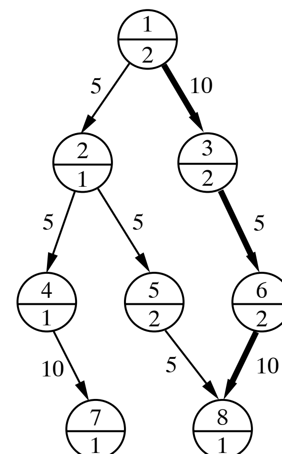


Figure 1: Task graph with communication and execution costs.

Scheduling and partitioning of task graphs of the kind described above have been studied thoroughly in the past three decades. There exist a plethora of different scheduling and partitioning algorithms in the literature for different kinds of task graphs, considering different aspects of the scheduling problem. The

general problem of scheduling task graphs for a multiprocessor system has been proven to be NP complete [15].

The rest of the paper is organized as follows: Section 3 gives a short summary of related work. Section 4 presents our contribution to parallelizing simulation code. In section 5 we give some results of our contribution, followed by a discussion and future work in section 6.

3 Related Work

A large number of scheduling and partitioning algorithms have been presented in the literature. Some of them use list scheduling techniques and heuristics [3, 11, 5, 2, 10, 22]. A list scheduler keeps a list of tasks that are ready to be scheduled, i.e. all its predecessors have already been scheduled. In each step it selects one of the tasks in the list, by some heuristic, and assigns it to a suitable processor, and updates the list.

Another technique is called critical path scheduling [17]. The critical path of a task graph (DAG) is the path having the largest sum of communication and execution cost. The algorithm calculates the critical path, extracts it from the task graph and assign it to a processor. After this operation, a new critical path is found in the remaining task graph, which is then scheduled to the next processor, and so on. One property of critical path scheduling algorithms is that the number of available processors is assumed to be unbounded, because of the nature of the algorithm.

Yet another approach to scheduling of task graphs is to first apply a task clustering algorithm and thereafter schedule the clusters for a fixed number of processors. A task clustering algorithm results in a cluster partition of the task graph. A cluster is a set of nodes designated to execute on the same processor. Thus, the communication costs for edges between nodes belonging to the same cluster are reduced to zero. A low complexity task clustering algorithm is the DSC algorithm [19]. It has a complexity of $O(n \cdot log(n))$.

An alternative approach to task clustering is to apply task merging algorithms [12]. The difference between a task clustering algorithm and a task merging algorithm is that in the task clustering case, tasks are not merged, i.e. the communication of data is still performed for each individual task in the cluster. But for the task merging case, the tasks are merged such that the new task resulting from the merge receives all its data *before* the computational work of the task, and sends all the resulting data to other tasks *after* the computational work has been performed.

Due to the merging property of a task merging algorithm, the resulting task graph will have a higher granularity value, i.e. the communication to computation ratio will increase. Thus, after a task merging algorithm has been applied any standard scheduling algorithm that works better for coarse grained task graphs can successfully be applied.

An orthogonal feature in scheduling algorithms is task duplication. Task duplication scheduling algorithms rely on task duplication as a mean of reducing communication cost. However, the decision if a task should be duplicated or not introduces additional complexity to the algorithm, pushing the complexity up in the range $O(n^3)$ to $O(n^4)$ for task graphs with n nodes.

4 Scheduling of Simulation Code

An overview of the automatic parallelization tool presented in this paper is given in Figure 2. The figure illustrates both how the sequential executable and the parallel executable that are built.

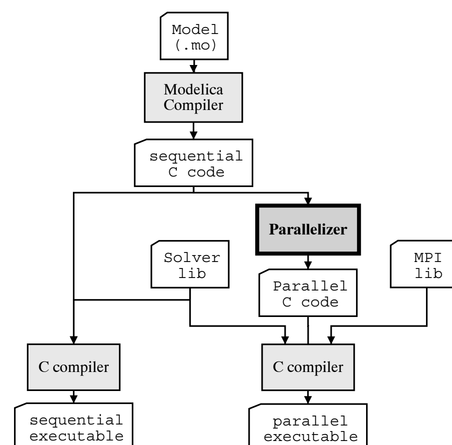


Figure 2: An overview of the parallelization tool and its environment.

Simulation code generated from Modelica mostly consists of a large number of assignments of expressions with arithmetic operations to variables. Some of the variables are needed by the DAE solver to calculate the next state, hence they must be sent to the processor running the solver. Other variables are merely temporary variables whose value can be discarded after the final use.

The simulation code is parsed, and a fine grained task graph is built, see the structure of the tool in Figure 3. The generated graph, which has the properties

of a DAG (Directed Acyclic Graph), can be very large. A typical application (e.g. a thermo-fluid model of a pipe, discretized to 100 pieces), with an integration time of around 10 milli seconds, has a task graph with 30000 nodes. The size of each node can also vary a lot. For instance, when the simulation code originates from a DAE, certain nodes represent an equation system that have to be solved in each iteration if they can not be solved statically at compile time. These equation systems can be linear or non-linear. In the linear case, any standard equation solver could be used, even parallel solvers. In the non-linear case, fixed point iteration is used. In both cases, the solving of each equation system is represented as a single node in the task graph. Such a node can have a large execution time in comparison to other nodes (like an addition or a multiplication of two scalar floating point values).

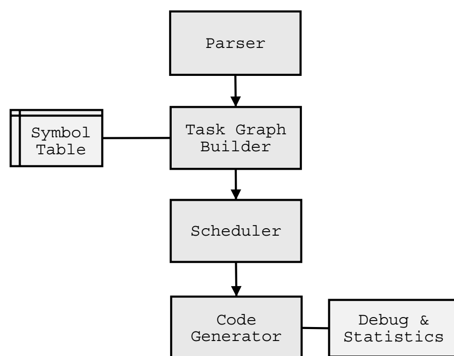


Figure 3: The internal architecture of the parallelization tool.

The task graph generated directly from the simulation code is not suitable for scheduling to multiprocessors. There are several reasons for this, the major reason is that the task graph is too fine grained for applying a standard scheduling algorithm. Many scheduling algorithms are designed for coarse grained task graphs. The granularity of a task graph is the relation between the communication cost between tasks and the execution cost of the tasks. There are several scheduling algorithms that can handle fine grained task graphs as well as coarse grained task graphs. One such category of algorithms is non-linear clustering algorithms [19, 20]. These algorithms consider putting siblings[1] into the same cluster to reduce communication cost. A problem with some of these algorithms is that the complexity is too high for the large task graphs generated by our tool.

[1] A sibling s, to a task n is defined as a node where n and s has a common predecessor.

A second problem with the task graphs generated is that in order to keep the task graph small, the implementation does not allow a task to contain several operations. For instance, a task can not contain both a multiplication and a function call. The simulation code can also contain Modelica `when` statements, which are equivalent to a special form of `if` statements without `else` branch. These need to be considered as one task, since if the condition of the `when` statement is true, all statements included in the `when` statement should be executed. An alternative would be to replicate the guard for each statement in the `when` statement. This is however not implemented yet, since usually the when statements are small in size and the need of splitting them up is low.

To solve the problems above, a second task graph is built, with references into the original task graph. The implementation of the second task graph makes it possible to cluster tasks into larger ones, thus increasing the granularity of the task graph. The first task graph is kept, since it is needed later for generating code. The two task graphs are illustrated in Figure 4.

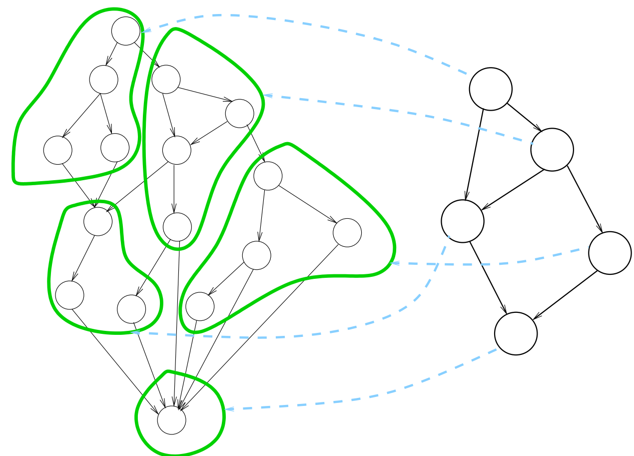


Figure 4: The two task graphs built from the simulation code.

The second level task graph can also be used to cluster tasks together using a task clustering or task merging algorithm. An algorithm for merging task similar to the algorithm found in [16] has been implemented in our tool, except that our algorithm deals with removing cycles. The algorithm constructs a cluster incrementally, starting with a single node n, taken from a list l of all nodes sorted by level in descending order.

The algorithms first examine the children of the node n. When all children have been clustered, the algorithm continues searching for a node that has a child (not in the cluster) with in-degree one and in-

cludes them as well, see figure 5. The next choice of including nodes is the set of parent nodes to the node *n*. After that, siblings to *n* are chosen, followed by an arbitrary node in the list of nodes.

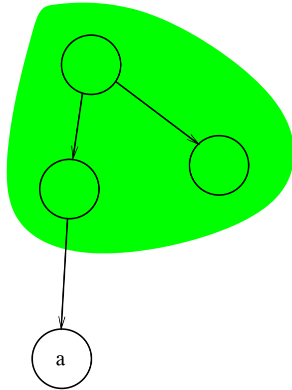


Figure 5: Node *a* with in-degree one is included in the cluster

To prevent the algorithm from producing cycles in the clustered task graph, a function that detects cycles is called, which includes all nodes forming a cycle from a cluster. Such a cycle is illustrated in Figure 6. The figure shows that by clustering nodes *a* and *b* together and not including *c* in the same task, the resulting task graph will be cyclic, thus removing the property of a DAG making it impossible to schedule.

When a cycle is detected several approaches can be taken. Either the complete cycle is added to the cluster, which can in the worst case make a cluster too large. Another alternative is to remove the node from the cluster causing the cycle, and begin a new cluster. This on the other hand might cause some clusters to be too small.

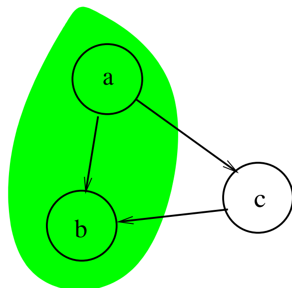


Figure 6: A cluster that forms a cycle in the resulting task graph.

Once the coarse grained task graph is built we can use standard scheduling algorithms found in the literature. In our implementation we have used a scheduling algorithm called TDS [1], which is a critical path scheduling algorithm with task duplication. For coarse grained graphs it produces the optimal solution. However, the number of processors needed by the algorithm is not fixed. Thus, to use the algorithm in practice a phase following the TDS algorithm has to be introduced. This phase limits the number of processors by merging task lists of different processors.

Finally, a simple method called Full Task Duplication Technique (FTDT) is implemented in the parallelization tool. It collects clusters by collecting all parents for each exit node (i.e. a node without any successor) into clusters. These clusters are then merged in a load balancing manner until the number of clusters match the required number of processors. This method is only useful if the task granularity (communication to execution time ratio) is very high, i.e. the average cost of communication is much larger than the average cost of execution in the task graph. However, since it applies *full* duplication it represents an upper limit on the possible speedup for task graph with very high communication costs and can thus be useful in some specific cases.

5 Results

The first results without the pre-clustering (or task merging) phase implemented showed that the TDS algorithm did not work well on fine grained tasks, even with an unlimited number of processors. Most examples did not produce speedups at all. The major reason was that the early implementation did not optimize the sending of messages between tasks, by sending and receiving larger chunks of data. In practice, each scalar produced its own MPI send and receive call.

But also the limitations of the TDS algorithm on fine grained task graphs (i.e. task graphs with high communication costs) had an effect on the result. The TDS algorithm is a linear critical path scheduling algorithm, i.e. it never schedules two siblings onto the same processor. This means that the TDS algorithm exploits all available parallelism, even if the communication cost is very high. Thus it does not work well on graphs with high granularity.

When using the modified task merging algorithm described above, the results were also not showing a speedup > 1. The main reason for not giving good results in this case was that the task merging algorithm did not succeed in both merging the tasks to increase the granularity and still reveal enough parallelism in the task graph such that speedup > 1 could be ob-

tained. Thus, the resulting task graph contained too many dependencies such that the only possible partition was a sequential partition for one processor.

The Full Task Duplication Technique did however produce reasonable results. Figure 7 gives computed speedup values from the task graphs generated from the simulation code for a thermofluid pipe model with three different discretization values. Figure 8 gives the measured speedup results for the same models when the parallelized simulation code is executed on a parallel PC-cluster with a SCI communication network [18].

Figure 9 presents computed speedup values from the task graph for the robot example (the r3 robot) found in the Modelica Standard Library.

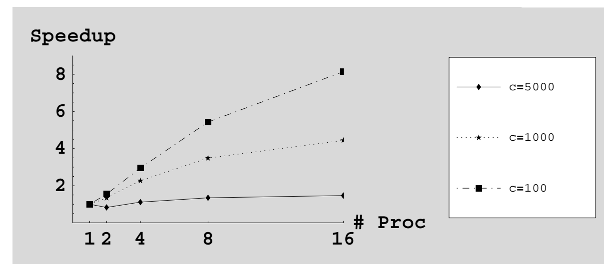
6 Discussion and Future Work

It is clear that the simulation code emitted from compilers of equation based simulation languages can be highly optimized and very irregular code. Hence, this code is not trivial to parallelize. The scheduling algorithms found in the literature are not suitable for fine grained task graphs of the magnitude produced by our tool. Therefore, a pre-clustering phase is needed. Also, the increasingly gap between processor speed and communication speed will demand better clustering and task merging algorithms in order to provide good speedup results in the future.

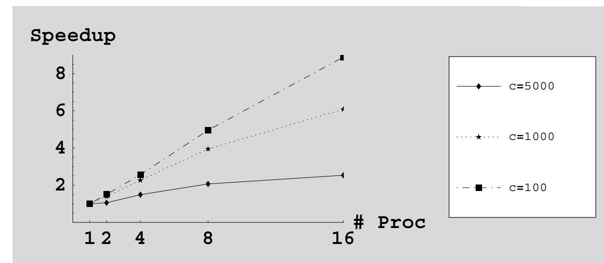
Due to the large task graphs, caused by the large simulation code files, the clustering algorithm must be of low complexity but still use for instance task duplication to reduce communication cost.

The results could be further improved by applying task duplication to the pre-cluster algorithm. Since each task can be very small, extensive task duplication could be used to reduce the communication in the clustered task graph. However, the demand for a low complexity algorithm does not allow an advanced task duplication scheme. Future work is to investigate what kind of task duplication could be considered in the pre-clustering phase. Clearly, it could improve the results significantly.

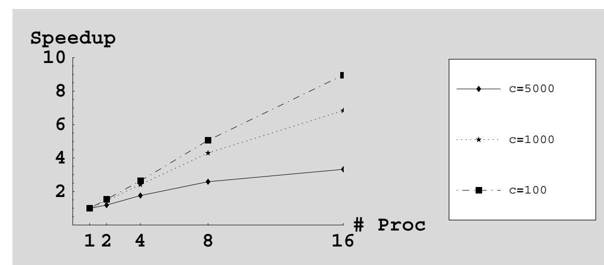
The results from the robot example did not produce good speedup. However, when using mixed mode and inline integration the amount of parallelism in the task graph increased. Therefore, future work includes a deeper investigation on larger models both using mixed mode and inline integration and without those optimizations. Future work also includes an investigation on the effects of different optimizations performed on the system of equations regarding the amount of parallelism in the simulation code.



(a) Thermofluid pipe with 50 discretization points.



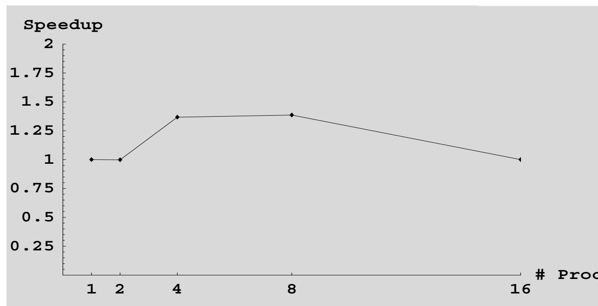
(b) Thermofluid pipe with 100 discretization points.



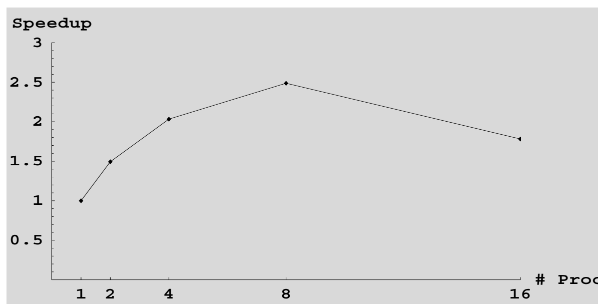
(c) Thermofluid pipe with 150 discretization points.

Figure 7: Computed speedup figures for different communication costs c using the FTD method on the Thermofluid pipe.

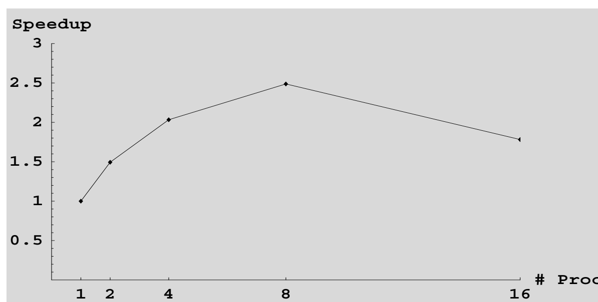
Future work on the scheduling and clustering problem for fine grained task graphs is also needed. Perhaps a more accurate parallel machine model is needed, like for instance the Logp parallel programming model [4].This would make the model more accurate, giving better estimates of the gained speedup. However, such an extension of the computational model would also increase the complexity of the scheduling and clustering algorithms.



(a) Thermofluid pipe with 50 discretization points.



(b) Thermofluid pipe with 100 discretization points.

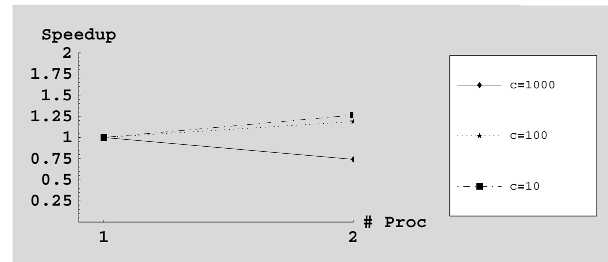


(c) Thermofluid pipe with 150 discretization points.

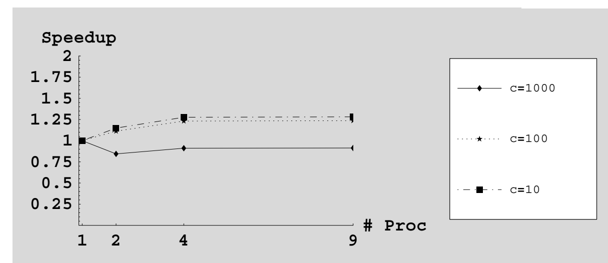
Figure 8: Measured speedup figures when executing on a PC-cluster with SCI network interface using the FTD method on Thermofluid pipe.

7 Acknowledgments

This has been supported by the Modelica Tools project in the Complex Systems framework supported by NUTEK (Swedish Board for Technical Development), and the EU-IST project RealSim.



(a) Mechanical robot model



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

Figure 9: Computed speedup figures for different communication costs, *c*, using the FTD method on the robot example.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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 par-

allel computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.

[5] 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.

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

[7] H. Elmqvist. *A Structured Model Language for Large Continuous Systems*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1978.

[8] P. Fritzson, V. Engelson. Modelica - A Unified Object-Oriented Language for System Modeling and Simulation. In *Proceedings of the 12th European conference on Object-Oriented Programming, LNCS*. Springer Verlag, 1998.

[9] P. Fritzson, L. Viklund, J. Herber, and D. Fritzson. High-level mathematical modeling and programming. *IEEE Software*, vol. 12(no. 4):77–87, July 1995.

[10] 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.

[11] J.J. Hwang, Y.C. Chow, F.D. Anger, C.Y. Lee. Scheduling Precedence Graphs in Systems with Interprocessor Communication Times. *Journal on Computing*, vol. 18(vol. 2), 1989.

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

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

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

[15] R.L. Graham, L.E. Lawler, J.K. Lenstra and A.H. Kan. Optimization an Approximation in Deterministic Sequencing and Scheduling: A Survey. *Annals of Discrete Mathematics*, pages 287–326, 1979.

[16] 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.

[17] S. Darbha, D. P. Agrawal. Optimal Scheduling Algorithm for Distributed-Memory Machines. *Transactions on Parallel and Distributed Systems*, vol. 9(no. 1), 1998.

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

[19] 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.

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

[21] 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.

[22] 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.