Clustering and Scheduling of Simulation Code Generated from Equation Based Simulation Languages

Peter Aronsson, Peter Fritzson
Dept. of Computer and Information Science,
Linköpings Universitet, SE-581 83 Linköping, Sweden

E-mail: (petar,petfr)@ida.liu.se

Abstract

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 Modelica compilation results in an Ordiniary Differential Equation system or a Differential Algebraic Equation system, depending on the specific Modelica model. The Modelica compiler typically performs optimizations on this system of equations to reduce its size. The optimized code consists of simple arithmetic operations, assignments, and function calls.

This paper presents an automatic parallelization tool that builds a task graph from the optimized sequential code produced by a commercial Modelica compiler. Various scheduling algorithms have been implemented, as well as specific enhancements to cluster nodes for better computation/communication tradeoff. Finally, the tool generates simulation code, in a master-slave fashion, using MPI.

Keywords: Scheduling, Clustering, Modelica, simulation, large complex system, TDS

1. 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 calculate the sine of a value, 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 is present between two tasks 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. The goal is to minimize the execution time of the parallel program. This often means that the communication between processors must be kept low, since inter processor communication is very expensive. When two tasks execute on the same processor, the communication cost between them is reduced to zero.

Scheduling and partitioning of such task graphs described above has been studied thoroughly in the past three decades. There exists 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 multi-processor system is proven to be NP complete [15].

The rest of the paper is organized as follows: Section 2 gives a short summary of related work. Section 3 presents our contribution of parallelizing simulation code. In section 4 we give some results of our contribution, followed by a discussion and future work in section 5.

2. Related Work

A large number of scheduling and partitioning algorithms have been presented in the literature. Some of them use a list scheduling technique and heuristics [3, 10, 12, 5, 4, 11]. 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.

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.

3. Scheduling of Simulation Code

Simulation code generated from Modelica mostly consist 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. This graph, which has the properties of a DAG, can be very large. A typical application (a thermo-fluid model of a pipe, discreticised 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, an equation system has to be solved in each iteration if it can not be solved statically at compile time. This equation system 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 the 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).

The task graph generated 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 tasks. The granularity of a task graph is the relation between the communication cost between tasks and the execution cost of tasks. There are several scheduling algorithms that can handle fine grained tasks as well as coarse grained tasks. One such category of algorithms is non-linear clustering algorithms [18, 1]. These algorithms consider putting siblings[1] cluster to reduce communication cost. A problem with these algorithms is that the complexity is too high for the large task graphs generated in this application area.

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 can be seen as a `if` statement 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` clause should be executed. An alternative would be to replicate the guard for each statement in the `when` clause. 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 1.

The second coarse grained task graph is built by clustering tasks together to form larger tasks. Here we use a very simple algorithm, see algorithm 1. This algorithm is similar

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

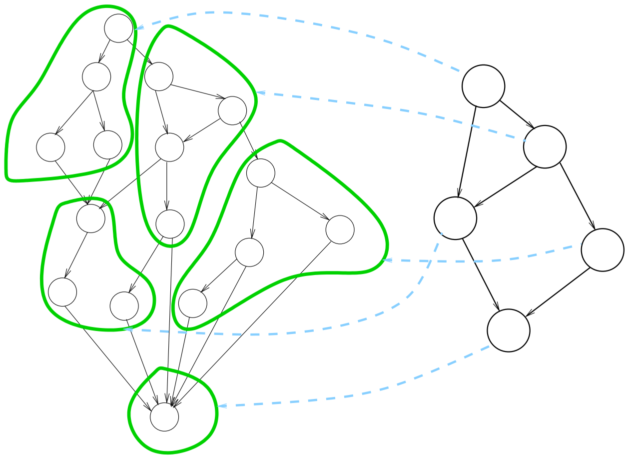

Figure 1. The two task graphs built from the simulation code.

to an algorithm that can be found in [16], 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 examines 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 includes them as well, see figure 2. The next choice of including nodes is parent nodes to *n*. After that, siblings to *n* are chosen, followed by an arbitrary node in the list of nodes.

To prevent the algorithm of 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 3. 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 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. A third option is to duplicate the node causing the cycle. This can be advantageous if the node causing the cycle is small, compared to the total cost of all nodes on the path causing the cycle.

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 [2], which is a critical path scheduling algorithm with task duplication. For coarse grained graphs it produces

Algorithm 1 **buildCluster(n:***node***, l:***list of nodes***, size:***integer***)**

cluster = addNode(n)
t = calcSize(cluster)
while $t < size$ **do**
  if $c \in Child(n)$ **and** $c \notin cluster$ **then**
    cluster = addNode(c)
    cluster = removeCycles(cluster)
    l = removeNode(l,c)
    t = calcSize(cluster)
    break
  end if
  if $c \in cluster$ **and** $|Child(c)| = 1$ **and** $Child(c) \notin cluster$ **then**
    cluster = addNode(Child(c))
    l = removeNode(l,c)
    t = calcSize(cluster)
    break
  end if
  if $c \in Sibling(n)$ **and** $c \notin cluster$ **then**
    cluster = addNode(Child(c))
    l = removeNode(l,c)
    t = calcSize(cluster)
    break
  end if
  if $c \notin cluster$ **and** $c \in l$ **then**
    cluster = addNode(Child(c))
    l = removeNode(l,c)
    t = calcSize(cluster)
    break
  end if
end while

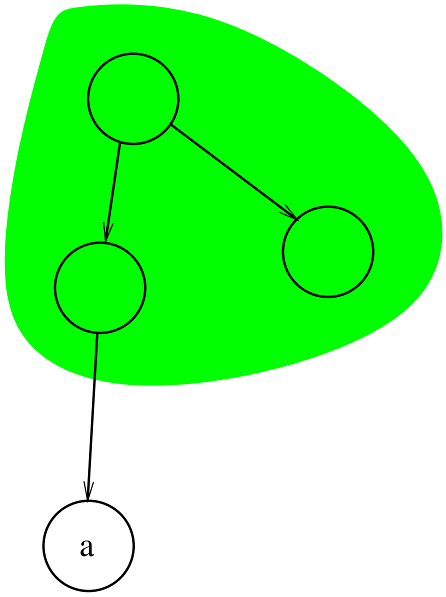

Figure 2. Node a with in-degree one is included in the cluster

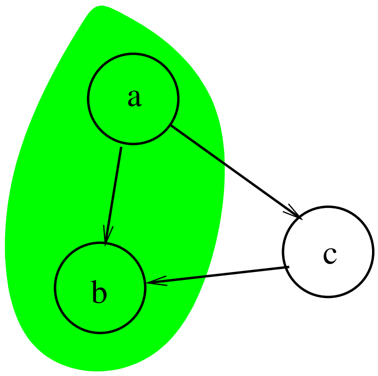

Figure 3. A cluster that forms a cycle in the resulting task graph

the optimal solution. However the number of processors needed is not fixed. Thus, to use the algorithm in practice a phase following the TDS algorithm has been introduced. This phase limits the number of processors by merging task lists of different processors.

4. Results

The first results without the pre-clustering phase implemented showed that the TDS algorithm did not produce well on fine grained tasks, even with 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 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 fine grained task graphs.

5. Discussion and Future Work

It is clear that the simulation code from equation based simulation languages produces highly optimized and very irregular code. Hence, it is not trivial to parallelize. The scheduling algorithms found in literature are not suited for fine grained task graphs of the magnitude produced by our tool. Therefore, a pre-clustering phase is needed.

Due to the large task graphs, caused by the large simulation code files, the clustering algorithm must be of low complexity. For instance, Sarkars approach of zeroing an edge (removing the edge and combine the two tasks associated with the edge) as long as the parallel time of the task graph does not increase [1], would not work in practice on our task graphs. The cost of calculating the parallel time of the task graph is too large. Therefore, a more efficient approach like ours is needed.

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 on 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.

Future work also includes a thorough comparison of scheduling algorithms in the tool.

The long term future work includes investigating the need of dynamic scheduling as a complement or a replacement for the static scheduling. Dynamic scheduling could be useful for instance when modeling a hybrid system, where both discrete and continuous variables are simulated. Discrete events is represented in the simulation code by Modelica when statements, discussed earlier. Typical for such systems are that at a given time one or several discrete events can happen. These events can affect the number of states and variables in the system, thus the system can at such an event totally change behavior. To be able to parallelize such systems, dynamic scheduling is needed.

Future work also includes parallelization of code using inline solvers and mixed mode integration. The system can be partitioned into parts, each with its own inline solvers, which reduces the dependencies between these parts.

6. Acknowledgments

This work started with support from the Modelica Tools project in Nutek, Komplexa Tekniska System and continues in the EU project RealSim.

References

[1] *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, 1999.

[2] 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.

[3] 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.

[4] 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.

[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*, 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. 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] Min-You Wu, Daniel D. Gajski. Hypertool: A Programming Aid for Message-Passing Systems. *Transactions on Parallel and Distributed Systems*, vol. 1(no. 3), 1990.

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

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

[15] R.L. Graham, L.E. Lawler, J.K. Lenstra and A.H. Kan. Optimization an Approximatioin 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. *Proceedings, Parallel and Distributed Processing*, pages 500–505, 1999.

[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] 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.