A Strategy for Parallel Simulation of Declarative Object-Oriented Models of Generalized Physical Networks

Francesco Casella[1]

[1]Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy,
francesco.casella@polimi.it

Abstract

For several years now, most of the growth of computing power has been made possible by exploiting parallel CPUs on the same chip; unfortunately, state-of-the-art software tools for the simulation of declarative, object-oriented models still generate single-threaded simulation code, showing an increasingly disappointing performance. This paper presents a simple strategy for the efficient computation of the right-hand-side of the ordinary differential equations resulting from the causalization of object-oriented models, which is often the computational bottleneck of the executable simulation code. It is shown how this strategy can be particularly effective in the case of generalized physical networks, i.e., system models built by the connection of components storing certain quantities and of components describing the flow of such quantities between them.

Keywords Parallel simulation, Declarative modelling, Structural analysis

1. Introduction

For several years now, most of the growth of computing power predicted by Moore's law has been made possible by exploiting parallel CPUs on the same chip; this trend is likely to continue for many years in the future. Significant speed-up in the simulation of declarative, object-oriented models will require to exploit the availability of parallel processing units.

With reference to the Modelica community, there are several attempts in this direction reported in the literature, mostly from Linköping University PELAB. One possible approach (see, e.g., [2, 8, 9]) is to analyse the mutual dependencies among the equations and variables of the system by means of graph analysis, eventually providing some kind of optimal scheduling of tasks to solve them in parallel, while avoiding idle times and bottlenecks in the computation.

A completely different approach, also pioneered at PELAB, is based on Transmission Line Modelling (TLM) [11, 13]. The basic idea is that physical interactions can often be modelled by means of components that represent wave propagation in finite time (e.g. pressure waves in hydraulic systems, elastic waves in mechanical systems, electromagnetic waves in electrical systems). It is then possible to split large system models into several smaller subsystems, that only interact through TLM components. This allows to simulate each sub-system in parallel for the duration of the TLM delay, as its behaviour will only depend on the past history of connected sub-systems.

This approach is interesting because it is based on physical first principles, introducing no approximations; however, the values of transmission line delays in most systems are quite small, thus limiting the maximum length of the integration time step allowed for the simulation. Moreover, this approach critically depends on the good judgement of the modeller, that must introduce appropriate TLM components all over the system model in order to obtain a performance benefit.

In spite of the above-mentioned studies, the state-of-the-art software tools for the simulation of declarative, object-oriented models still generate single-threaded simulation code, as of today. This results in an increasingly disappointing performance, as the number of cores available on standard desktop workstations or even laptops roughly doubles every two years, while the simulation speed basically remains the same.

The goal of this paper is to show that, for a fairly large class of object-oriented models of physical systems, a simple strategy for parallel simulation can be envisioned, which is expected to provide large speed-up ratios when using many-cores CPUs, and which does not depend on the accurate estimation of the computation and communication delay to obtain good performance. Since this strategy is very easy to implement and test, it is hoped that it quickly finds its way into mainstream object-oriented simulation tools, thus improving the state of the art in this field.

The paper is organised as follows. Section 2 contains the statement of the problem and a discussion of related work. The algorithm to partition the solution of the model

5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. 19 April, 2013, University of Nottingham, UK.

EOOLT 2013 website:
http://www.eoolt.org/2013/

into independent tasks that can be executed in parallel is described in detail in Section 3. In Section 4, models of generalized physical networks are introduced, and the results of the partitioning algorithm are discussed. Section 5 briefly discusses the task scheduling problem, while Section 6 concludes the paper with final remarks and indications for future work.

2. Problem Statement and Related Work

The starting point of this analysis is an equation-based, object-oriented model of a dynamical system, e.g. written in Modelica. For the sake of conciseness, the analysis is limited to continuous-time systems, though it could be easily extended to hybrid systems with event handling and clocked variables.

After flattening, the model is transformed into a set of Differential-Algebraic Equations (DAEs)

$$F(x, \dot{x}, v, t) = 0, \tag{1}$$

where $F(\cdot)$ is a vector-valued function, x is the vector of variables appearing under derivative sign in the model, v is the vector of all other algebraic variables, and t is the time.

A commonly adopted strategy for the numerical simulation of such systems is to first transform the DAEs (1) into Ordinary Differential equations (ODEs), i.e., solving equations (1) for the state derivatives $\dot{x}$ and for the other variables v as a function of the states and of time:

$$\dot{x} = f(x, t) \tag{2}$$

$$v = g(x, t). \tag{3}$$

By defining the vector z of unknowns as

$$z = \begin{bmatrix} \dot{x} \\ v \end{bmatrix}, \tag{4}$$

the ODEs (2)-(3) can be formulated as

$$z = h(x, t) \tag{5}$$

Sophisticated numerical and symbolic manipulation techniques (see [5] for a comprehensive review) are employed to generate efficient code to compute $h(x, t)$, which will then be called by the ODE integration algorithm at each simulation step. Implicit integration algorithms will also require every now and then the computation of the Jacobian $\frac{\partial h(x,t)}{\partial x}$, which might be performed either symbolically or numerically [4, 3]. This code is then linked to standard routines for numerical integration of differential equations, such as DASSL or the routines from the Sundials suite, thus generating an executable simulation code.

In this context, it is in principle possible to exploit parallelism in the computation of $h(x, t)$, in the computation of $\frac{\partial h(x,t)}{\partial x}$, and in the algorithm for numerical integration, which might, e.g., employ parallel algorithms to solve the implicit equations required to compute the next state value at each time step. This paper focuses on those problems in which the computation of $h(x, t)$ takes the lion's share of the simulation time, e.g., because it involves the computation of cumbersome functions to evaluate the properties of a fluid in an energy conversion system model. Of course, it is also possible to combine the approach presented here with the parallel computation of the Jacobian (which is fairly trivial if done numerically) and of the numerical integration algorithms, but this is outside the scope of the present work.

The algorithm presented in the next section follows the same principle that was first put forward in [1], and later on further developed in the Modelica context in [2]: exploiting the dependencies between the different equations (and parts thereof) to determine the order in which the systems has to be solved and which systems that can be solved in parallel. However, it tries to do so in a simpler way, by exploiting the mathematical structure of generalized network models.

More specifically, [2] first represents the algorithm to compute $h(x, t)$ with the finest possible granularity: each node in the dependency graph is a single term in the right-hand-side expressions of those equations that can be solved explicitly for (x, t), or a system of implicit equations for those who can't. Subsequently, these atomic tasks are merged into larger tasks, taking into account execution and communication costs, in order to minimize the overall execution time and maximize the parallel speed-up ratio. The merging algorithms are fairly involved, and their result critically depends on those costs, which are often hard to estimate reliably. Moreover, this kind of analysis seems to fit well the computational model of networked systems (e.g., clusters of workstations, which were popular at the time of that work), where communication delays are significant when compared to execution times, making a clever merging of tasks mandatory for good performance. Results obtained by the application of these techniques to a few representative test models were reported in [2] and subsequent related work, but no analysis was ever attempted to understand what is the typical structure of the dependencies in different classes of physical system models, in order to understand how much they can benefit in general from the application of this parallelization technique. Unfortunately, even though these algorithm were implemented in earlier versions of the OpenModelica compilers, they are currently no longer supported, which prevents trying them on real-life problems that can only be handled by more recent versions of the compiler.

The aim and scope of this paper are somewhat different. First of all, the underlying computational model is that of multiple-core CPUs with shared memory, in which communication delays tend to be small or negligible compared to execution times, at least as long as all the variables of the model can be kept within the on-chip cache memory at all times. This seems a feasible proposition for models of moderately large size: a system with 10000 variables (after optimizations such as alias elimination) in double precision requires only 80 kilobytes of shared cache memory. Second, it is shown that a fairly large class of object-oriented models, namely generalized physical networks, has a dependency structure that can be very well exploited by a sim-

ple parallelization algorithm which does not critically depend on the accurate estimation of execution times, but can guarantee nearly optimal allocation of parallel resources, as long as the number of nodes in the network is much larger than the number of parallel processing units.

3. An Algorithm for Parallel Solution of Equations from Declarative Models

The proposed algorithm is now outlined in detail.

1. Build an Equations-Variables (E-V) digraph, where every E-node corresponds to an equation in (1), every V-node correspond to a scalar unknown variable in z, and an edge exists between an E-node and a V-node if the unknown variable shows up in the equation.
2. Find a complete matching between E and V nodes (see [7] for a review of suitable algorithms); if this is not possible because the DAE system has index greater than one, apply Pantelides' algorithm [12] and the Dummy Derivative algorithm [10] until the system is reduced to index 1 and a complete matching can be found.
3. Transform the E-V digraph into a directed graph by first replacing each non-matching edge with an arc going from the E-node to the V-node, then by collapsing each V-node with its matching E-node.
4. Run Tarjan's algorithm [6] on the directed graph to locate its strongly connected components, corresponding to systems of algebraic equations that need to be solved simultaneously for their matching unknown variables.
5. Collapse each set of nodes making up a strong components into one macro-node.
6. Let $i = 1$
7. Search for all the sinks in the graph and collect them in the set S_i; these correspond to equations (or to systems of implicit equations) that can be solved independently of each other.
8. Delete all nodes in S_i from the directed graph, as well as all arcs connected to them.
9. If there are still nodes in the graph, increase i by one and goto Step 7.

When the algorithm shown above terminates, all the equations and systems of implicit equations of the system will be collected in the sets S_i.

Proof: after executing Step 5, the directed graph has no closed cycles left in it, because each and every strong components has been collapsed into a single macro-node; therefore, there exists at least one sink in the graph. Removing nodes without outgoing arcs does not create cycles, so that at each iteration at least one node is removed from the graph, until there will be none left, QED.

Note that state-of-the-art Modelica tools already perform Steps 1 to 4, so that the addition to the tool code in order to implement the proposed strategy is minimal.

The result of this analysis can also be visualized in terms of the Block Lower Triangular (BLT) representation of the incidence matrix, see Figure 1. Each set S_i corresponds to a block diagonal square matrix in the BLT matrix, marked in red in Figure 1, where every block on the diagonal corresponds to a strong component of the system of equations. As the ordering induced by the directed graph is partial, there exist many different BLT transformations of the original system corresponding to the same graph.

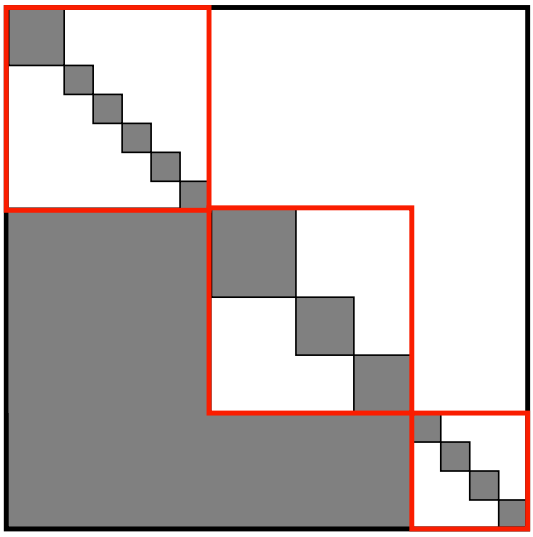

Figure 1. The incidence matrix in BLT form with S_i sets.

All the equations or systems of equations showing up in each set S_i can now be solved independently on parallel cores. Before moving to the solution of set S_{i+1}, it is necessary to wait that all equations belonging to the set S_i have been solved.

The latter requirement can in general create bottlenecks, e.g., $N-1$ cores might stand idle for a long time, waiting for the N^{th} one to complete its task. However, if the number of nodes is much larger than the number of cores and there is no single node whose execution time is disproportionately longer than that of all the others, on average the impact of such situations on the overall execution time will be small. It will be shown in the next Section that this is precisely the case of large generalized physical networks.

4. Application to Generalized Network Models

Many physical models can be built by connecting storage components and flow components, see Figures 2-3. The former ones describe the storage of certain quantities, by means of dynamic balance equations; the latter instead describe the flow of those quantities between different components, which is governed by the difference of some potential variable at the two boundaries. Two examples will be detailed in this section: thermal networks and thermo-hydraulic networks.

4.1 Thermal Networks

Thermal networks describe the flow of heat between bodies having different temperature. In this case the stored quantity is thermal energy, which is conveniently described by temperature state variables, while the flow components

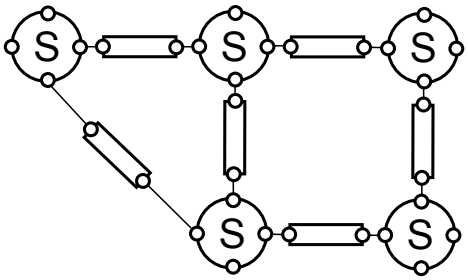

Figure 2. A physical network model with flow components connected to storage components only.

compute the thermal power flow based on the temperature at the two boundaries.

Thermal storage components are described by energy balance equations:

$$C(T_i)\frac{dT_i}{dt} = \sum_j Q_{i,j}, \tag{6}$$

where T_i is the temperature of the i-th storage component, $C(T)$ is the thermal capacitance, and $Q_{i,j}$ are the heat flows entering the i-th component. For simplicity, assume all thermal power flows can be modelled by constant thermal conductances:

$$Q_i = G_i(T_{i,a} - T_{i,b}) \tag{7}$$

where G_i is the thermal conductance of the i-th flow component and $T_{i,a}, T_{i,b}$ are the two boundary temperatures.

Assuming that each thermal flow component is directly connected to two storage components, as in Figure 2, once all alias variables have been eliminated, the equations (6) will be matched to their corresponding temperature derivatives, while the equations (7) will be matched to their corresponding heat flows. There will be no strong components in the E-V graph, corresponding to a strictly lower triangular BLT form of the incidence matrix. If the algorithm presented in Section 3 is now applied, the set S_1 will contain all the flow equations (7), each of which can be solved independently, and the set S_2 will contain all the storage equations (6), each of which can be solved independently once the heat flows have been computed by the equations in S_1.

With reference to the thermal network in Fig. 2, the directed graphs at each algorithm iteration are shown in Fig.

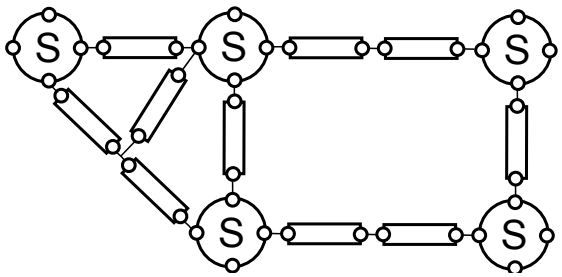

Figure 3. A physical network model with flow components directly connected to each other.

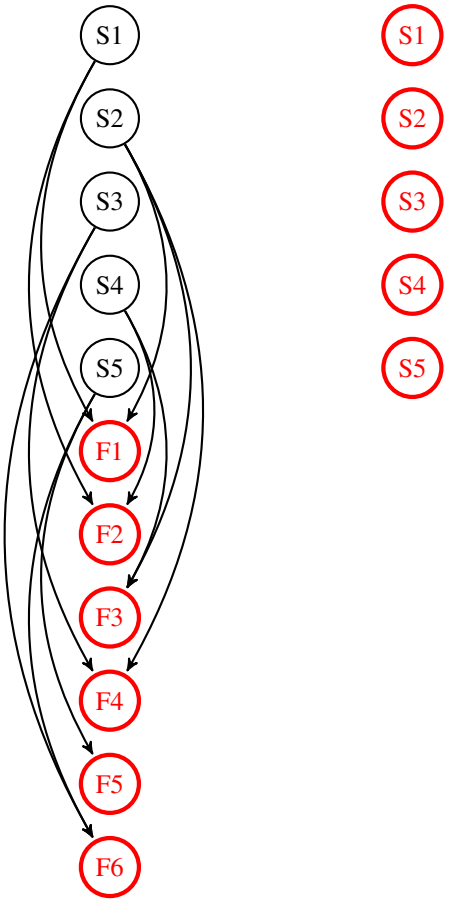

Figure 4. Iterations of the parallelization algorithm: thermal network

4. Nodes marked with the letter S represent storage equations (6), nodes marked with the letter F represent thermal flow equations (7); thick-bordered red nodes correspond to the set S_i at the i-th iteration.

In case of more complex connection topologies, where the heat flow components are directly connected to other heat flow components as in Figure 3, there will be strong components in the E-V graph, corresponding to sets of algebraic equations that must be solved simultaneously to determine the heat flows and the intermediate temperatures, which in this case are not known state variables. Consequently, the set S_1 will also contain the corresponding macro-nodes.

4.2 Thermo-Hydraulic Systems

Thermo-hydraulic networks describe the flow of mass and thermal energy between different components representing the storage of mass and thermal energy in finite volumes of the system, by means of flow components describing the mass flow and the heat flow (e.g., due to convective heat transfer) between different volumes. In this case, the stored quantities are mass and energy, which can be described, e.g., by pressure and temperature state variables. Mass flow rates are determined by the pressure difference between the boundaries of flow components, and also by the upstream

properties of the fluid (e.g., the density). Heat flows are determined by thermal conductances, as in the previous sub-section.

Storage components are described by mass and energy balance equations:

$$\begin{bmatrix} e_i & h_i & \rho_i & \frac{\partial \rho_i}{\partial p} & \frac{\partial \rho_i}{\partial T} & \frac{\partial e_i}{\partial p} & \frac{\partial e_i}{\partial T} \end{bmatrix} = f(p_i, T_i) \quad (8)$$

$$M_i = \rho_i V_i \quad (9)$$

$$\frac{dM_i}{dt} = \sum_j w_{i,j} \quad (10)$$

$$\frac{dE_i}{dt} = \sum_j w_{i,j} h_{i,j} + \sum_j Q_{i,j} \quad (11)$$

$$\frac{dM_i}{dt} = \frac{\partial \rho_i}{\partial p}\frac{dp_i}{dt} + \frac{\partial \rho_i}{\partial T}\frac{dT_i}{dt} \quad (12)$$

$$\frac{dE_i}{dt} = \left(\frac{\partial e_i}{\partial p}\frac{dp_i}{dt} + \frac{\partial e_i}{\partial T}\frac{dT_i}{dt}\right) M_i + e_i \frac{dM_i}{dt} \quad (13)$$

where $e_i, h_i, \rho_i, p_i, T_i$ are the specific internal energy, specific enthalpy, density, pressure, and temperature of the fluid contained in the i-th component, V_i is the volume of the component, M_i is the mass of the fluid contained in the component, $w_{i,j}$ are the mass flow rates entering the component, $h_{i,j}$ the associated upstream specific enthalpies, and $Q_{i,j}$ the heat flows entering the component.

Mass flow components determine the mass flow rate as a function of the boundary pressures and of the upstream density:

$$w_i = w(p_{i,a}, p_{i,b}, \rho_i), \quad (14)$$

while heat flows components are the same as in the previous sub-section.

Assuming that mass flow and heat flow components are always connected between two storage components, once all alias variables have been eliminated, equations (8) will be matched to all the properties on their left-hand-side, equation (9) will be matched to M_i, equation (10) will be matched to $\frac{dM_i}{dt}$, equation (11) will be matched to $\frac{dE_i}{dt}$, the pairs of equations (12)-(13) will be matched to dp_i/dt and dT_i/dt, forming a strong component of two variables and equations for each i, equations (14) will be matched to the mass flow rates w_i, and equations (7) will be matched to the heat flows Q_i.

After the algorithm illustrated in Section 3 has been applied, the set S_1 will contain all the fluid property equations (8) and the thermal flow equations (7), each of which can be computed independently. The set S_2 will contain the equations (9), (14), which can be solved independently. The set S_3 will contain the equations (10) and (11), each of which can be solved independently. Finally, the set S_4 will contain the macro-nodes corresponding to the systems (12)-(13), each of which can be solved independently to compute the state derivatives.

With reference to a simple system composed of three storage components, connected in series by two mass flow components, with the flow direction from the first to the last storage component, the directed graphs shown in Fig. 5 are obtained at each iteration. Equations (8) are marked with P, (9) with M, (14) with F, (10) with MB, (11) with EB, (12)-(13) with D. As before, thick-bordered red nodes correspond to the set S_i at the i-th iteration.

In case there are series-connected heat flow or mass flow components in the system, as in Figure 3, their variables and equations will form strong components in the E-V graph, which will end up in sets S_1 (heat flows) and S_2 (mass flows), respectively.

4.3 Outlook

First of all, it is worth noting how the number of sets of equations S_i, that need to be solved in sequence, remains very low (2 for thermal networks and 4 for thermo-hydraulic networks), regardless of the size of the system. Therefore, as the number of components increases, the possibility of exploiting a large number of parallel CPU also increases, since there will be an increasing number of tasks in each S_i that can be performed in parallel before synchronizing for the transition to the next set S_{i+1}. For instance, if there are 1000 storage components in a thermo-hydraulic network, it is possible to distribute the computation of the corresponding fluid properties over up to 1000 parallel cores.

It is also worth noting that in the case of thermo-hydraulic systems such as steam power plant models, the computation of the fluid properties in each storage component, equation (8), often takes up the lion's share (90% or more) of the CPU time required to solve the DAEs for the state derivatives, and also a large share of the total CPU time required for the entire system simulation, as the time required to compute $h(x,t)$ dominates the time spent by the integration algorithm to find the value of the next state vector. A simple strategy as the one proposed here will be thus very effective in this case, since those computations will all end up in set S_1, and thus will be performed in parallel on all the available CPU cores. In these cases, a speed-up ratio close to the number of cores can be expected.

5. Scheduling Policies

The parallel tasks determined by the algorithm discussed in the previous section need to be run several times at each simulation time step, depending on the chosen integration algorithm, so they will be run hundreds or thousands of times in a typical simulation run. A trivial scheduling policy for the parallel solution of the system equations is to first set up a thread for each (macro) nodes in the graph; subsequently, for every required computation of $h(x,t)$, the threads corresponding to the set S_1 are activated, so they run on the first available core until there are no more threads running, then those corresponding to set S_2, and so on and so forth until S_N is completed. All threads read and write from and to a shared memory; since every node only computes the variables it is matched to, and only reads variables that were computed in previous parallel sequences, it is guaranteed that read/write conflicts cannot take place, thus avoiding the need of mechanisms such as semaphores.

In many cases (e.g., when cumbersome fluid properties computations are involved, as noted in the previous section), such a simple policy could already be highly advan-

Figure 5. Iterations of the parallelization algorithm: thermo-hydraulic network

tageous, compared to a purely sequential solution of the DAEs. However, there are two major potential problems that could arise.

The first problem is the impact of the overhead required to activate a thread for each (macro) node in the equations directed graph. Consider for example the case described in Section 4.1: every instance of equation (7), which only requires a subtraction and a multiplication to be solved, will end up in a separate thread. If the thread activation time is comparable or higher than the time required for the two floating point operations, then the end result of this parallelization strategy could be a code that actually runs slower than its sequential counterpart. This problem can be solved by roughly estimating the order of magnitude of the execution time associated to each (macro) node, and then aggregate many of them until the thread set-up time is negligible compared to the total execution time.

The second problem is how to guarantee that all cores are used as much as possible, and none stays idle for a long time. If a few tasks in S_i take a much longer time than all the other ones, and there is a large number of parallel cores available, activating them as the last ones might result in a waste of time, because most of the cores will eventually stay idle, waiting for those longer tasks to end. A possible solution to this problem is to estimate the execution time of each task, then start the longer-running ones first. Again, a very rough estimate of the order of magnitude of the execution time is enough for this purpose. If the number of tasks in S_i, which corresponds to the number of nodes in the physical networks, is much larger than the number of processing units, the impact of the idle time spent at the end of each parallel section of the algorithm will on average be small, compared to the total execution time. This is the case, for example, if a 16-cores CPU is used to simulate a network of a few hundred nodes (e.g., a thermal power plant model).

6. Conclusions and Future Work

In this paper, a simple algorithm has been presented that allows to distribute over parallel CPU cores the solution of DAEs stemming from object-oriented models. It has been shown how this algorithm can be very effective in partitioning the solution of the system DAEs over many parallel CPU cores, when applied to large models of thermal and thermo-hydraulic networks, which can easily involve hundreds or thousands of storage and flow models.

In the near future, it is planned to implement the algorithm in the OpenModelica compiler, which already offers support for parallel simulation of systems having a decoupled structure (e.g., thanks to the TLM methodology), using the OpenMP framework. This will allow to experiment the proposed strategy on generalized physical network models, but also on different kinds of models, such as mechanical systems. Another interesting perspective could be to couple the strategy presented here for parallel solving of DAEs with parallel ODE solvers for sparse systems, in order to fully exploit parallelism in all the tasks required to simulate an object-oriented declarative model.

References

[1] Niclas Andersson and Peter Fritzson. Generating parallel code from object oriented mathematical models. In *Proceedings 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, USA, Jul 19–21 1995.

[2] P. Aronsson. *Automatic Parallelization of Equation-Based Simulation Programs*. PhD thesis, Linköping University, Department of Computer and Information Science, 2006.

[3] Willi Braun, Stephanie Gallardo Yances, Kilian Link, and Bernhard Bachmann. Fast simulation of fluid models with colored jacobians. In *Proceedings of the 9th International Modelica Conference*, pages 247–252, Munich, Germany, Sep. 3–5 2012. Modelica Association.

[4] Willi Braun, Lennart Ochel, and Bernhard Bachmann. Symbolically derived jacobians using automatic differentiation - Enhancement of the OpenModelica compiler. In *Proceedings 8th International Modelica Conference*, pages 495–501, Dresden, Germany, Mar 20-22 2010. Modelica Association.

[5] F. E. Cellier and E. Kofman. *Continuous System Simulation*. Springer-Verlag, 2006.

[6] I. S. Duff and J. K. Reid. An implementation of Tarjan's algorithm for the block triangularization of a matrix. *ACM Transactions on Mathematical Software*, 4(2):137–147, 1978.

[7] Jens Frenkel, Gunter Künze, and Peter Fritzson. Survey of appropriate matching algorithms for large scale systems of differential algebraic equations. In *Proceedings 9th International Modelica Conference*, pages 433–442, Munich, Germany, Sep. 2012. Modelica Association.

[8] H. Lundvall. Automatic parallelization using pipelining for equation-based simulation languages, 2008. Lic. Thesis.

[9] H. Lundvall, K. Stavåker, P. Fritzson, and C. Kessler. Automatic parallelization of simulation code for equation-based models with software pipelining and measurements on three platforms. *Computer architecture news, Special issue MCC08 - Multicore computing 2008*, 36(5), 2008.

[10] S. E. Mattsson and G. Söderlind. Index reduction in differential-algebraic equations using dummy derivatives. *SIAM Journal on Scientific Computing*, 14(3):677–692, 1993.

[11] Kaj Nyström and Peter Fritzson. Parallel simulation with transmission lines in Modelica. In *Proceedings 5th Modelica Conference*, pages 325–331, Vienna, Austria, Sep 6–8 2006. The Modelica Association.

[12] Constantinos C. Pantelides. The consistent initialization of differential-algebraic systems. *SIAM Journal on Scientific and Statistical Computing*, 9(2):213–231, 1988.

[13] Martin Sjölund, Robert Braun, Peter Fritzson, and Petter Krus. Towards efficient distributed simulation in modelica using transmission line modeling. In *Proceedings 3rd International Workshop on Equation-Based Object-Oriented Languages and Tools*, pages 71–77, Oslo, Norway, Oct 3 2010.