Parallelizing Equation-Based Models for Simulation on Multi-Core Platforms by Utilizing Model Structure.

Martin Sjölund Mahder Gebremedhin Peter Fritzson
Department of Computer and Information Science
Linköping University, Sweden
{martin.sjolund, mahder.gebremedhin, peter.fritzson}@liu.se

***Abstract*—In today's world of high tech manufacturing and computer-aided design simulations of models is at the heart of the whole manufacturing process. Trying to represent and study the variables of real world models using simulation computer programs can turn out to be a very expensive and time consuming task. On the other hand advancements in modern multi-core CPUs promise remarkable computational power. Modern modeling environments provide different optimization and parallelization options to take advantage of the available computational power. Some of these parallelization approaches are based on automatically extracting parallelism with the help of the model compiler or translator. Another approach is to provide the model programmers with the necessary language constructs to express any potential parallelism in their models.**

In this paper we present an automatic parallelization approach for Modelica models using Transmission Line Modeling (TLM). TLM is suitable for parallel simulations because larger models can be partitioned into smaller independent sub-models. TLM introduces parallelism into the system by decoupling subsystems using delays greater than the step size of the numerical solver. A prototype has been implemented in the OpenModelica Compiler (OMC) framework. Our approach re-uses the dependency analysis from the sequential translation step of OMC. With the help of the dependency analysis information the set of equations for a model is partitioned into a number of sub-systems. The resulting independent sub-systems are scheduled and executed in parallel. The run-time system for OMC has been improved to provide thread safety and handle parallelism while keeping the introduced overhead to minimum for normal sequential operation and maintaining portability.

Keywords— Transmission Line Modeling; Parallel Computing; Simulation; Modelica; Compiler; Multi-Core

I. INTRODUCTION

With the advent of multi-core computers an important way of creating efficient simulations and computations is to use parallel computing where the computational work is divided between the processors of a multi-core system. Since multi-core processors are becoming more mainstream than single-core processors, it is very important to utilize the resulting parallel computing power. This requires some kind of support in compilers and development environments.

This work has been supported by Serc, by Elliit, by the Swedish Strategic Research Foundation in the EDOp and HIPo projects and by Vinnova in the RTSIM and ITEA2 MODRIO projects. The Open Source Modelica Consortium supports the OpenModelica work.

The process of compiling and simulating Modelica models to sequential code is described in detail in [*1*] and [*2*]. The handling of equations is rather complex and involves symbolic index reduction, topological sorting according to the causal dependencies between the equations, conversion into assignment statement form, etc. Simulation corresponds to "solving" the compiled equation system with respect to time using a numerical integration method.

The rest of the paper is organized as follows. Section II provides some background information on the language and tools used in this work. Section III describes parallelization approaches and opportunities to parallelize a Modelica model. In section IV we describe Transmission Line Modeling which is at the core of this work. Section V describes how partitioning to sub-systems is handled. Section VI explains modifications done to the OpenModelica run-time system to handle parallel simulations and thread safety. Finally, in Section VII we present some measurements and results and in Section VIII a general discussion and conclusions are presented.

II. BACKGROUND

A. Modelica

Modelica [*3*] is a non-proprietary, object-oriented, equation based, multi-domain modeling language for component-oriented modeling of complex physical systems, e.g., containing mechanical, electrical, electronic, hydraulic, thermal, control, electric power, and/or process-oriented subcomponents.

The development and standardization of the Modelica language is overseen and supported by the non-profit Modelica Association [*4*]. The Modelica Association also develops the open source Modelica Standard Library.

Modelica is an equation-based language. Equations, in Modelica, represent *equality* rather than *assignment* relations and have no predefined *causality*. Unlike assignment statements equations can contain expressions on both right-hand and left-hand sides of the equation. Such equations are manipulated symbolically and sorted in data dependency order by a Modelica compiler to determine their relative order of execution in the solution process.

Modelica compilation results in an Ordinary Differential Equation system or a Hybrid Differential Algebraic Equation system, depending on the specific Modelica model. The Modelica compiler typically performs symbolic optimizations on this system of equations to reduce its size and make it more stable for numerical computation. The optimized code mostly consists of simple arithmetic operations, assignments, function calls, and function definitions.

B. The OpenModelica Compiler (OMC)

OpenModelica [*5*] is an open-source Modelica-based modeling and simulation environment intended for industrial and academic usage. Its long-term development is supported by a nonprofit organization – the Open Source Modelica Consortium (OSMC) [*6*].

The Programming Environments Laboratory (PELAB) [*7*] at Linköping University, together with OSMC, is developing the OpenModelica modeling and simulation environment including the OpenModelica Compiler (OMC) for the Modelica language (including the MetaModelica extensions). There is also an Eclipse plug-in called Modelica Development Tooling (MDT) which includes a debugger. A Template Code Generation language called Susan [*8*] [*9*] is also used to simplify code generation and further developed.

III. Parallel Simulation of Modelica Models on Multi-Core Computers

Compiling Modelica models for efficient parallel simulation on multi-core architectures requires additional methods compared to the typical approaches described in [*1*] and [*2*]. The parallel methods can be roughly divided into the following groups:

- Explicit parallelism in the language: With this approach the language can be extended to provide additional constructs for explicitly stating potential parallelism in the model code. This approach has been explored in [*10*] and [*11*].
- Automatic parallelization: In this approach the compiler itself is responsible for analyzing the program or model, extracting potential parallelism, partitioning the computational work and automatically produce parallel code. This approach has been explored in [*12*] [*13*] and [*14*].

Automatic parallelization is a preferred way of parallelization from the users' perspective since users and programmers do not need to be familiar with parallel programming which is usually time consuming and error prone. This is even more advantageous in areas of equation-based modeling languages where modelers are often application field experts rather than programming experts.

There can be different approaches to automatic parallelization of equation-based languages like Modelica.

In this paper we present two automatic parallelization approaches. The first is based on Transmission Line Modeling (TLM) in which the modeler can introduce additional parallelism into the system by inserting delay elements into the model. Such subsystems can be de-coupled using delays greater than the step size of the numerical solver. The second is based on a set of recursive decompositions of model equations into independent subsystems of equations and computing the subsystems in parallel. The prototypes have been implemented in the OpenModelica Compiler (OMC). In this context automatic means that the extraction and parallelization is done by OMC rather than the modeler or programmer.

OMC takes an object-oriented Modelica representation of a model, translates the model descriptions and provides a set of flat equations representing the model and then solves these sets of equation. Our approach re-uses the dependency analysis from the sequential translation step. Scheduling the computations of these sub-systems ensures that the data-dependencies are obeyed and computational load is balanced between subsystems. With the help of the dependency analysis information the set of equations for a model is partitioned into a number of sub-systems. The resulting independent sub-systems are scheduled and executed in parallel. Finally, parallel C source code is generated for these subsystems instead of the normal sequential C source code.

The current implementation uses OpenMP [*15*] as parallel runtime platform. The implementation is done partially in the code generator of OMC and partially in the runtime system.

IV. Transmission Line Modeling (TLM)

A computer simulation model is basically a representation of a system of equations that model some physical phenomena.

The goal of simulation software is to solve this system of equations in an efficient, accurate and robust way. In order to achieve such a goal the by far most common approach is to use a centralized solver algorithm which puts all equations together into a DAE or an ODE system of equations. The system is then solved using matrix operations and numeric integration methods.

One disadvantage of this approach is that it often introduces data dependencies between the central solver and the equation system, making it difficult to parallelize the equations for simulation on multi-core platforms. Another problem is that the stability of the numerical solver depends on the simulation step size.

The fundamental idea behind the TLM method is to model a system in a way such that components can be somewhat numerically isolated from each other. This allows each component to solve its own equations independently of the rest of the system. This is achieved by replacing capacitive components (for example volumes in hydraulic systems) with transmission line elements of a length for which the physical propagation time corresponds to at least one simulation time step. In this way a time delay is introduced between the resistive components (for example orifices in hydraulic systems). The result is a physically accurate description of wave propagation in the system [*16*].

One noteworthy property with this method is that the time delay represents a physically correct separation in time between components of the model. Since the wave propagation speed (speed of sound) in a certain liquid can be calculated, the conclusion is that the physical length of the line is directly

proportional to the delay time used when simulating the component, see Equation 1. Note that this delay time is a parameter in the component and can very well differ from the time step used by the simulation engine. Keeping the delay in the transmission line larger than the simulation solver time step is important, to avoid extrapolation of delayed values. This means that a minimum time delay of the same size as the maximum time step is required, introducing a modeling error for very short transmission lines.

$$l = ha = \sqrt{\frac{\beta}{\rho}} \tag{1}$$

TLM isolates/decouples model components, making them largely independent from other subsystems. This property makes it very suitable to prepare models for parallel simulations .

The Modelica delay-operator will break any equation-variable direct dependency and thus also makes the adjacency matrix contain independent subsystems. Finding independent subsystems (i.e., strongly connect components of a graph) in an adjacency matrix is an operation that can be performed using fast algorithms from graph theory given that the adjacency matrix is sparse.

V. PARTITIONING

The main reason for using TLM is that a coarse-grained parallelization of the system is implicitly gained. We present a general approach to partitioning a system of equations that utilizes the time-delay introduced by TLM [*17*].

Each partition of the equation system will be independent from any other within the current time step. This means they can be parallelized by synchronizing between time steps.

The common data-structure used for sorting and matching equations uses the dependences between variables and equations, usually stored as a sparse adjacency matrix (See [*1*] chapter 17). We can assume that any Modelica compiler will have this structure readily available.

The equation system is transformed into block lower triangle (BLT) form, where each BLT block corresponds to either a single equation from the original system of equations or a strongly connected component (several equations) in the strongly connected components dependency graph of the system of equations. This means that the system can be solved sequentially. What happens when TLM is used to model the system is that some entries in the adjacency matrix disappear since delay expressions are allowed to decouple the system if they only access data in former time steps.

The basic data structure needed to perform the partitioning analysis is the adjacency matrix. Neither the BLT matrix nor the sorted system is needed since it is possible to use the adjacency matrix alone to determine if two equations are totally connected in the graph. The benefit of only looking at the adjacency matrix is that the equation system can be partitioned before optimizations are performed, some of which are costly to perform on large systems since they do not have linear time complexity.

Since not all nodes are connected to each other, the graph is not a tree, but a forest. The goal is to find all trees in the forest since these are possible to run in parallel. There are many ways to do this operation fast. Which one to choose depends mostly on the data representation that is used. Cormen [*18*] contains an interesting algorithm using disjoint sets. Our approach instead uses a depth-first search, marking all reachable nodes then choosing the next unmarked vertex and repeating the algorithm until all trees have been found. The algorithm has a complexity of O(|V| + |E|), as any connected components algorithm should.

VI. RUNTIME SYSTEM AND THREAD SAFTY

The OpenModelica runtime system is quite complex and provides a lot of functionality. It has to provide support for simulations which can use different solvers, interactive simulations, external features like FMI etc. It has to support different mathematical operations, optimization features and so on. It has to provide different interfaces for specific purposes, for e.g. Java, FORTRAN interfaces. Support for MetaModelica compilation and execution is also part of the runtime system.

To provide efficient performance and implementations of this functionality the runtime system needs to have a flexible and efficient memory management system.

Among many things this should include support for smart arrays. Smart, in the sense that, the array representations should be aware of the number of dimensions as well as the sizes of each dimension. Since the runtime system is mainly implemented using C this is not available out of the box. This means that there needs to be a custom array container implementation that can work hand in hand with the memory management system. The OpenModelica runtime achieved this by representing Modelica arrays as C structures with additional information about the array.

The actual data of each array is located in a global memory pool. The global memory pool operates in a very similar way to a traditional stack implementation of last in first out. Each array structure has a pointer pointing to its own specific data. There are two main reasons for using a global memory pool instead of raw allocations (*malloc()*) per need. The first reason is that using a global memory pool the runtime system can avoid potential memory leaks which can be very troublesome for simulations which take relatively longer time.

Simulations involve computations over many simulation time steps. This means that a small memory leak can build up over time to a considerable amount. By using a custom memory pool the runtime achieves a better and safe memory model. The second reason is more related to the nature of physical models, specifically those which are usually modeled with Modelica. Most Modelica models (at least so far) do not have large arrays. This means that the amount of memory allocated per array is relatively small, in other words the memory allocated per *malloc* call is relatively small. Now keeping in mind that a certain array allocation will most probably be done at every time step, it would be expensive to call *malloc* at each time-step compared to the actual operations performed on the array elements.

Having a global memory pool as explained above is suitable for normal sequential operations of the run-time system. However it adds certain issues for multi-threaded execution. Having all threads share the common memory pool and update it properly will require an extensive amount of tracking of allocations and de-allocations to ensure that memory is not left locked. In addition this will require extra measures to ensure that proper synchronization is achieved to avoid data races and conflicts.

A rather relaxed approach is to let threads have their own representations of the memory model which they keep updated between themselves. Memory pools are created based on the number of threads specified for the simulation. Each thread uses its own memory pool to perform any allocations and de-allocations as needed. These memory pools act as private data/memory spaces for each thread. Anything that needs to be shared between threads is allocated directly (not using the memory pools) and is shared directly.

Of course not the whole run-time system needs multiple threads for its execution. More specifically we are using multiple threads right now for computing the different partitions or subsystems generated by the compiler backend. Currently this includes algebraic equations and ODE equations. These equations are solved on every time step.

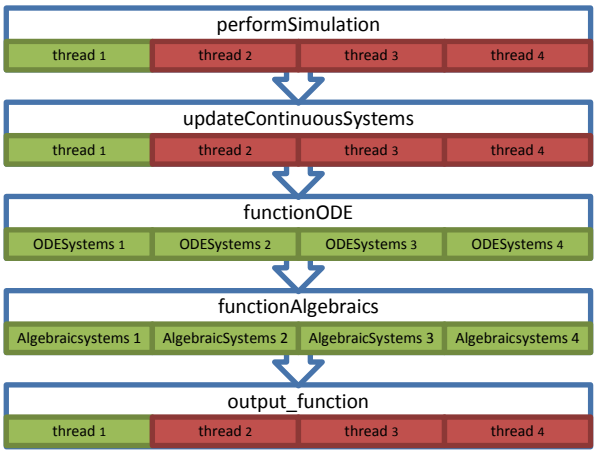


Fig. 1. Simplified thread guidance example through run-time system.

This means that the implementation can launch threads when needed, for example when it needs to solve a set of algebraic independent subsystems. It will synchronize them as needed and continue with the sequential execution when it is done.

However launching threads at every time-step will incur a quite unnecessary overhead. For example if a simulation involves 1000 time steps then this means the run-time system will have to launch and join threads 1000 times. To work around this issue we have decided to create a pool of threads at the start of simulation and guide them through the execution environment to the desired location. All threads except one will be available but idle until the exact point where they are needed. Then each thread will operate on an individual subsystem until there are no subsystems left as shown in Fig. 1.

Once the computations on independent subsystems is finished the master thread continues with the rest of the execution with the rest of the threads going back to idle state. This process is repeated whenever there are independent computations to be performed.

VII. Measurements and Results

To be able to evaluate the relative performance gains of the implementation we have used a simple hydraulic system model consisting of a volume with a pressure relief valve as shown in Fig. 2.

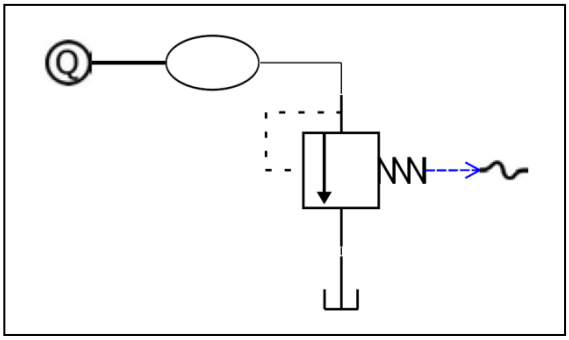


Fig. 2. A volume with a pressure relief valve.

A pressure relief valve is a safety component. It has a spring at one end of the spool and the upstream pressure, that is, the pressure at the side of the component where the flow is into the component, is acting on the other end.

The preload of the spring will make sure that the valve is closed until the upstream pressure reaches a certain level, when the force from the pressure exceeds that of the spring. The valve then opens, reducing the pressure to protect the system.

The results of performance measurements with the volume split into different number of segments using an RK4 integrator and a step size of $5*10^{-6}$ are shown in Fig. 3 and Fig. 4. The measurement does not include the model instantiation, flattening and back-end specific operations. Only the computation time for the simulation executable to complete is measured and compared.

The simulation is performed on an Intel Xeon W3565 quad-core CPU with clock frequency of 3.2GHz.

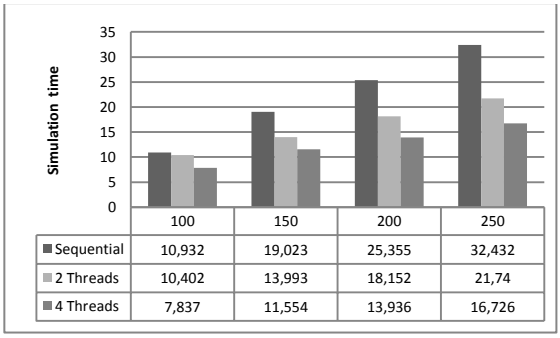
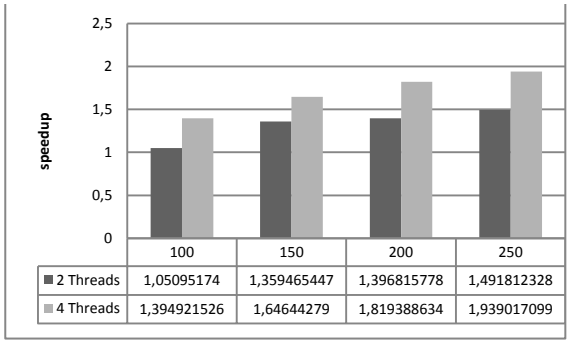


Fig. 3. Simulation time vs. number of segments.



	100	150	200	250
2 Threads	1,05095174	1,359465447	1,396815778	1,491812328
4 Threads	1,394921526	1,64644279	1,819388634	1,939017099

Fig. 4. Gained Speedup for different number of segments.

VIII. Discussion and Conclusion

From the results it can be seen that doubling the number of cores does not necessarily reduce the simulation time by half. Of course parallelism is rarely close to the ideal expectation. If a computation is complex and involves a number of algorithms, the parallelization efficiency can be degraded to some extent due to the inherently sequential parts of the computation. This applies to most if not all simulations of physical systems.

The reason that the performance gains for this specific model are not proportional to the number of threads or cores used is that a typical simulation involves a considerable amount of setup, sanity checks and other miscellaneous operations which have to be performed within each time step. Some of these operations cannot be or are not yet parallelized.

Performance gains for simulations can be improved further as more and more of the static or dynamic (model dependent) run-time gets parallelized. Since the OpenModelica run-time system is now thread safe for most parts, it is relatively easy to add parallelism according to need. The current thread and memory management implementation can be used to direct computations in parallel when possible and needed in other parts of the run-time system.

References

[1] Peter Fritzson, *Principles of Object-Oriented Modelling and Simulation with Modelica 2.1*, 1st ed.: Wiley-IEEE Press, 2004.

[2] François E., Kofman, Ernesto Cellier, *Continuous System Modeling.*, 2006.

[3] Modelica. [Online]. [Last accessed: 2013-06-09]. Available from: https://www.modelica.org/.

[4] Modelica Association. [Online]. [Last accessed: 2013-06-09]. Available from: https://www.modelica.org/association.

[5] OpenModelica. [Online]. [Last accessed: 2013-06-09]. Available from: http://www.openmodelica.org/.

[6] Open Source Modelica Consortium (OSMC). [Online]. [Last accessed: 2011-06-09]. Available from: http://www.ida.liu.se/labs/pelab/modelica/OpenSourceModelicaConsortium.html.

[7] PELAB. [Online]. [Last accessed: 2013-06-09]. Available from: http://www.ida.liu.se/~pelab/.

[8] Rickard Lindberg, "A TemplateBased Code Generator for the OpenModelica Compiler," Linköping University, LIU-IDA/LITH-EX-A--10/006--SE, 2010.

[9] Peter Fritzson. Modelica Text Template Language Susan. [Online]. [Last accessed: 2013-06-09]. Available from: https://openmodelica.org/svn/OpenModelica/tags/OPENMODELICA_1_9_0_BETA_4/doc/OpenModelicaTemplateProgramming.pdf.

[10] Christoph Kessler, Peter Fritzson, and Mattias Eriksson, "NestStepModelica – Mathematical Modeling and Bulk-Synchronous Parallel Simulation," in *PARA'06 Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing.*, Linköping, Sweden, 2006, pp. 1006-1015.

[11] Mahder Gebremedhin, Afshin Hemmati Moghadam, Peter Fritzson, and Kristian Stavåker, "A Data-Parallel Algorithmic Modelica Extension for Efficient Execution on Multi-Core Platforms," in *Proceedings of the 9th International Modelica Conference*, Munich, Germany, Sept 3-5, 2012.

[12] Peter Aronsson, "Automatic Parallelization of Equation-Based Simulation Programs," Linköping University, Dissertation No. 1022, 2006.

[13] Håkan Lundvall, "Automatic Parallelization using Pipelining for Equation-Based Simulation Languages," Linköping University, Linköping, Sweden, Licentiate Thesis 1381, 2008.

[14] Per Östlund, "Simulation of Modelica Models on the CUDA Architecture.," Linköping Univeristy, Linköping, Sweden, Master Thesis LIU-IDA/LITH-EX-A--09/062--SE, 2009.

[15] openmp.org. [Online]. [Last accessed: 2013-06-09]. Available from: http://openmp.org.

[16] Petter Krus, "Robust System Modelling Using Bi-lateral Delay Lines," in *Proceedings of the 2nd Conference on Modeling and Simulation for Safety and Security (SimSafe)*, Linköping, Sweden, 2005.

[17] Martin Sjölund, Robert Braun, Peter Fritzson, and Petter Krus, "Towards Efficient Distributed Simulation in Modelica using Transmission Line Modeling," in *Proceedings of the 3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, (EOOLT'2010)*, Oslo, Norway, Oct 3, 2010.

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L Rivest, and Clifford Stein, *Introduction to Algorithms*, 3rd ed.: The MIT Press, 2009.