AUTOMATIC PARALLELIZATION OF OBJECT ORIENTED MODELS ACROSS METHOD AND SYSTEM

Håkan Lundvall and Peter Fritzson

PELAB – Programming Environment Lab, Dept. Computer Science
Linköping University, S-581 83 Linköping, Sweden

haklu@ida.liu.se (Håkan Lundvall)

Abstract

In this work we report preliminary results of automatically generating parallel code from equation-based models together at two levels: Performing inline expansion of a Runge-Kutta solver combined with fine-grained automatic parallelization of the resulting RHS opens up new possibilities for generating high performance code, which is becoming increasingly relevant when multi-core computers are becoming common-place.
We have introduced a new way of scheduling the task graph generated from the simulation problem which utilizes knowledge about locality of the simulation problem. The scheduling is also done in a way that limits communication, to the greatest extent possible, to neighboring processors and expensive global synchronization is avoided. Preliminary tests on a PC-cluster show speedup that is better than what was achieved in previous work were parallelization was done on the system only.

Keywords: Modelica, automatic parallelization.

Presenting Author's biography

Håkan Lundvall got his masters degree in Computer Science and Engineering from Linköping university in 1999, has since been working in the software industry as a consultant. Since 2003 he combines his work as a consultant with part time studies towards a phd at the Programming Environments Laboratory at the computer science department of Linköping University . Håkan's interests lies within modeling and simulation, high performance computations and parallelization.

1 Background – Introduction to Mathematical Modeling and Modelica

Modelica is a rather new language for equation-based object-oriented mathematical modeling which is being developed through an international effort [5][4]. The language unifies and generalizes previous object-oriented modeling languages. Modelica is intended to become a *de facto* standard. It allows defining simulation models in a declarative manner, modularly and hierarchically and combining various formalisms expressible in the more general Modelica formalism. The multidomain capability of Modelica gives the user the possibility to combine electrical, mechanical, hydraulic, thermodynamic, etc., model components within the same application model.

In the context of Modelica *class libraries* software components are Modelica classes. However, when building particular models, components are *instances* of those Modelica classes. Classes should have well-defined communication interfaces, sometimes called ports, in Modelica called *connectors*, for communication between a component and the outside world. A component class should be defined *independently of the environment* where it is used, which is essential for its *reusability*. This means that in the definition of the component including its equations, only local variables and connector variables can be used. No means of communication between a component and the rest of the system, apart from going via a connector, is allowed. A component may internally consist of other connected components, i.e. *hierarchical* modeling.

To grasp this complexity a pictorial representation of components and connections is quite important. Such graphic representation is available as *connection diagrams*, of which a schematic example is shown in Figure 1 where a complex car simulation model is built in a graphical model editor.
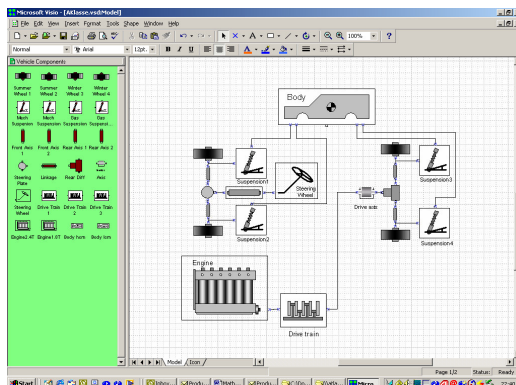

Figure 1. Complex simulation models can be built by combining readily available components from domain libraries.

To summarize, Modelica has improvements in several important areas:

- *Object-oriented mathematical modeling*. This technique makes it possible to create physically relevant and easy-to-use model components, which are employed to support hierarchical structuring, reuse, and evolution of large and complex models covering multiple technology domains.

- *Acausal modeling*. Modeling is based on *equations* instead of assignment statements as in traditional input/output block abstractions. Direct use of equations significantly increases re-usability of model components, since components adapt to the data flow context in which they are used. This generalization enables both simpler models and more efficient simulation. However, for interfacing with traditional software, algorithm sections with assignments as well as external functions/procedures are also available in Modelica.

- *Physical modeling of multiple application domains*. Model components can correspond to physical objects in the real world, in contrast to established techniques that require conversion to "signal" blocks with fixed input/output causality. In Modelica the structure of the model becomes more natural in contrast to block-oriented modeling tools. For application engineers, such "physical" components are particularly easy to combine into simulation models using a *graphical* editor (Figure 1).

2 The OpenModelica Open Source Implementation

In this work the OpenModelica software is used which is the major Modelica open-source tool effort.

The OpenModelica environment is the major Modelica open-source tool effort [3] consists of several interconnected subsystems, as depicted in Fig. 2. Arrows denote data and control flow. Several subsystems provide different forms of browsing and textual editing of Modelica code. The debugger currently provides debugging of an extended algorithmic subset of Modelica. The graphical model editor is not really part of OpenModelica but integrated into the system and available from MathCore without cost for academic usage. In this research project two parts of the OpenModelica subsystem is used.

- *A Modelica compiler subsystem*, translating Modelica to C code, with a symbol table containing definitions of classes, functions, and variables. Such definitions can be predefined, user-defined, or obtained from libraries.

- *An execution and run-time module.* This module currently executes compiled binary code from translated expressions and functions, as well as simulation code from equation based models, linked with numerical solvers.
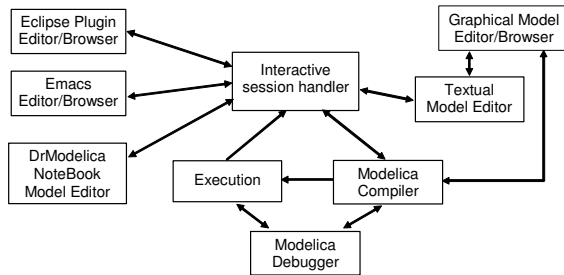

Fig. 2. OpenModelica architecture.

3   Approaches to Integrate Parallelism and Mathematical Models

There are several approaches to exploit parallelism in mathematical models. In this section we briefly review some approaches that are being investigated in the context of parallel simulation of Modelica models.

3.1   Automatic Parallelization of Mathematical Models

One obstacle to parallelization of traditional computational codes is the prevalence of low-level implementation details in such codes, that also makes automatic parallelization hard.

Instead, it would be attractive to directly extract parallelism from the high-level mathematical model, or from the numerical method(s) used for solving the problem. Such parallelism from mathematical models can be categorized into three groups:

- *Parallelism over the method.* One approach is to adapt the numerical solver for parallel computation, i.e., to exploit parallelism over the method. For example, by using a parallel ordinary differential equation (ODE) solver for that allows computation of several time steps simultaneously. However, at least for ODE solvers, limited parallelism is available. Also, the numerical stability can decrease by such parallelization.

- *Parallelism over time.* A second alternative is to parallelize the simulation over the simulated time. This is however best suited for discrete event simulations, since solutions to continuous time dependent equation systems develop sequentially over time, where each new solution step depends on the immediately preceding steps.

- *Parallelism of the system.* This means that the modeled system (the model equations) are parallelized. For an ODE or DAE equation system, this means parallelization of the right-hand sides of such equation systems which are available in explicit form; moreover, in many cases implicit equations can automatically be symbolically transformed into explicit form.

A thorough investigation of the third approach, automatic parallelization over the system, has been done in our recent work on automatic parallelization (fine-grained task-scheduling) of a mathematical model [1] [11], Fig 3.
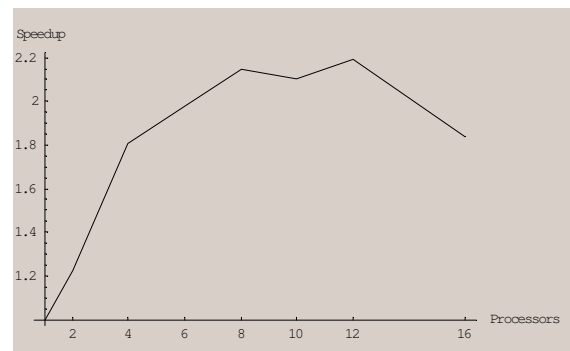

Fig. 3. Speedup on Linux cluster with SCI interconnect.

In this work we aim at extending our previous approach to inlined solvers, integrated in a framework exploiting several levels of parallelism.

3.2   Coarse-Grained Explicit Parallelization Using Computational Components

Automatic parallelization methods have their limits. A natural idea for improved performance is to structure the application into computational components using strongly-typed communication interfaces.

This involves generalization of the architectural language properties of Modelica, currently supporting components and strongly typed connectors, to distributed components and connectors. This will enable flexible configuration and connection of software components on multiprocessors or on the GRID. This only involves a structured system of distributed solvers/ or solver components.

3.3   Explicit Parallel Programming

The third approach is providing general easy-to-use explicit parallel programming constructs within the algorithmic part of the modeling language. We have previously explored this approach with the NestStep-Modelica language [7, 12]. NestStep is a parallel programming langue based on the BSP (Bulk-Synchronous Parallal) model which is an abstraction of a restricted message passing architecture and charges cost for communication. It is defined as a set of language extensions which in the case of

NestStepModelica is added to the algorithmic part of Modelica. The added constructs provide shared variables and process coordination. NestStepModelica processes run, in general, on different machines that are coupled by the NestStepModelica language extensions and runtime system to a virtual parallel computer.

4 Combining Parallelization at Several Levels

Models described in object oriented equations based languages like Modelica render large differential algebraic equation systems that can be solved using numerical ODE-solvers. Many scientific and engineering problems require a lot of computational resources, particularly if the system is large or if the right hand side is complicated and expensive to evaluate. Obviously, the ability to parallelize such models is important, if such problems are to be solved in a reasonable amount of time.

As mentioned in Section 3, parallelization of object oriented equation based simulation code can be done at several different levels. In this paper we explore the combination of the following two parallelization approaches

- Parallelization across the method, e.g., where the stage vectors of a Runge-Kutta solver can be evaluated in parallel within a single time step
- Fine grained parallelization across the system where the evaluation of the right hand side of the system equations is parallelized.

The nature of the model dictates to a high degree what parallelization techniques that can be successfully exploited.

We suggest that often it is desirable to apply parallelization on more than one level simultaneously. In a model where two parts are loosely coupled it can, e.g., be beneficial to split the model using transmission line modeling and use automatic equation parallelization across each sub model. In this paper however we investigate the possibility of doing automatic parallelization across the equations and the solver simultaneously. In previous work [1] automatic parallelization across the system has been done by building a task graph containing all the operations involved in evaluating the equations of the system DAE. In order to make the cost of evaluating each task large enough compared to the communication cost between the parallel processors he uses a graph rewriting system that merges tasks together in such a way that the total cost of computing and communicating is minimized. In his approach the solver is centralized and runs on one processor. Each time the right hand side is to be evaluated, data needed by tasks on other processors is send and the result of all tasks is collected in the first process before returning to the solver. As a continuation of this work we now inline an entire Runge-Kutta solver in the task graph before scheduling of the tasks.

Many simulation problems have DAE:s consisting of a very large set of equations but were each equation only depends on a relatively small set of other equations.

Let $f = (f_1,...,f_n)$ be the right hand side of such a simulation problem and let f_i contain equations only depending on equations of components of indices in a range near i. This makes it possible to pipeline the computations of the resulting task graph, since
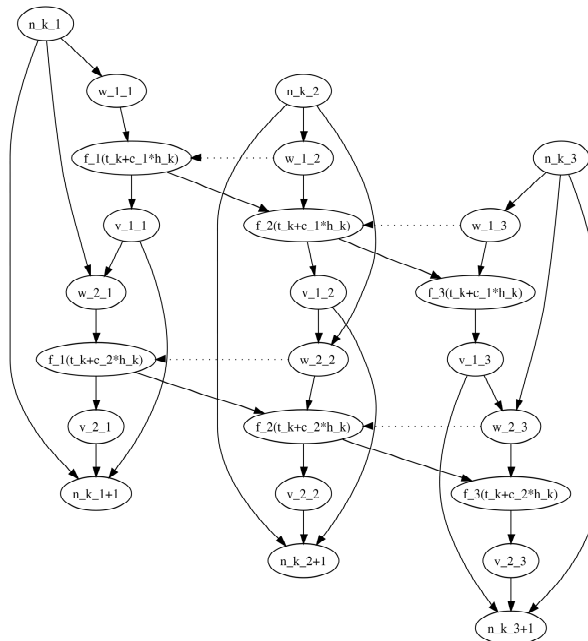

Fig. 4. Task graph of two stage inlined Runge-Kutta solver.

evaluating f_i for stage s of the Runge-Kutta solver depend only on f_j of stage s for j close to i and on f_i of stage s-1.

A task graph of a system where the right hand side can be divided into three parts, denoted by the functions f_1, f_2 and f_3 where f_i only depend on f_{i-1}, inlined in a two stage Runge-Kutta solver is shown in figure 4. In the figure n_k represent the state after the previous time step. We call the function f_i the blocks of the system. If we schedule each block to a different processor, let us say f_i is scheduled to p_i, then p_1 can continue calculating the second stage of the solver as p_2 starts calculating the first stage of f_2. The communication between p_1 and p_2 can be non-blocking so that if many stages are used communication can be carried out simultaneous to the calculations. In a shared memory system we only have to set a flag that the data is ready for the next process and no data transfer must take pace.

The pipelining technique is described in [10]. Here we aim to automatically detect pipelining possibilities in the total task graph containing both the solver stages and the right hand side of the system, and automatically generate parallelized code optimized for the specific latency and bandwidth parameters of the target machine.

It the earlier approach with task merging including task duplication the resulting task graph usually ends up with one task per processor and communication takes place at two points in each simulation step; initially when distributing the previous step result from the processor running the solver to all other processors and at the end collecting the results back to the solver.

When inlining a multi-stage solver in the task graph each processor only needs to communicate whit its neighbor. In this approach however we cannot merge tasks as much since the neighbors of a processor depends on initial results to be able to start their tasks. So, instead of communicating a lot in the beginning and in the end smaller portions are communicated throughout the calculation of the simulation step.

If the task graph of a system mostly has the property of having a narrow access distance, which is required for the pipelining, but only on a small number of places access components in more distant parts of the graph. The rewriting system could also make suggestions to the user of places in the model that would benefit from a decoupling using transmission line modeling if it can be done without loosing the physical correctness of the model.

The task rewriting system is build into the OpenModelica compiler previously mentioned.

5 Pipelining the task graph

Since communication between processors is going to be more frequent with this approach we want to make sure the communication interfere as little as possible with computation. Therefore, we schedule the tasks in such a way that communication taking place inside the simulation step is always directed from a processor with lower rank to a higher ranked processor. In this way the lower ranked processor is always able to carry on with calculations even if the receiving processor temporarily falls behind. At the end of the simulation step there is a face were values required for the next simulation step is transferred back to lower ranked processors, but this is only needed once per simulations step instead of once for each evaluation of the right hand side. Further more this communication takes place between neighbors and not to a single master process which otherwise can get overloaded with communication as the number of processors becomes large.

6 Sorting Equations for Short Access Distance

One part of translating an acausal equation-based model into simulation code involves sorting the equations into data dependency order. This is done using Tarjan's algorithm which also finds any strongly connected components in the system graph, i.e., a group of equations that must be solved simultaneously. We assign a sequence number to each variable, or set of variables in case of a strongly connected component, and use this to help the scheduler assign tasks that communicate much within the same processor. When the task graph is generated each task is marked with sequence number of the variable it calculates. When a system with n variables is to be scheduled onto p processors, tasks marked 1 through n/p is assigned to the first processor and so on.

Even though Tarjan's algorithm assures that the equations are evaluated in a correct order we cannot be sure that there is not a different ordering where the access distance is smaller. If for example two parts of the system is largely independent they can become interleaved in the sequence of equations making the access distance unnecessarily large. Therefore we apply an extra sorting step after Tarjan's algorithm which moves equations with direct dependencies closer together. This reduces the risk of two tasks with a direct dependency getting assigned to different processors.

As input to the extra sorting step we have a list of components and a matching defining which variable is solved by which equation. On component represent a set of equations that must be solved simultaneously. A component often includes only one equation. The extra sorting step works by popping a component from the head of the component list and placing them in the resulting sorted list as near the head of the sorted list as possible without placing it before a component on which it depends. See pseudo-code for the algorithm below.

The following data structures are used in the code:

`matching`	A map from variables to the equations that solve them.

componentList	The initial list of components.
sortedList	The resulting sorted list of components.

```
Set sortedList To an empty set of components
While componentList not empty
    Set comp To  componentList.PopHead()
    Set varSet To the set of all variables
        accessed in any equation of comp
    Set eqSet To an empty set of equations
    For each variable v in varSet
        eqSet.insert(matching[v])
    end for

    Devide sortedList into left and right so
    that right is the largest suffix of
    sortedList where Intersection(right,eqSet)
    is empty.

    Set sortedList To the concatenation of
        left, comp and right
End while
```

7   Scheduling

In this section we describe the scheduling process. We want all communication occurring inside the simulation step to be one-way only, from processors with lower rank to processors with higher rank. To achieve this we make use of information stored with each task telling us from which equation it originates and thus which variable it is a part of evaluating. We do this by assigning the tasks to the processors in the order obtained after the sorting step described in section 6.

Task with variable number 0 through n_1 is scheduled to the first processor, n_1+1 through n_2 to the second and so on. The values of n_i are chosen so that they are always the variable number representing a state variable.

If we generate code for a single stage solver, e.g., Euler, this would be enough to ensure backward communication only takes place between simulation steps, since the tasks are sorted to ensure no backward dependencies. This is not, however, the case when we generate code for multi-stage solvers. When sorting the equations in data-dependency order, variables considered known, like the state of the previous step are not considered, but in a later stage of the solver those values might have been calculated by an equation that comes later in the data-dependency sorting. This kind of dependency is represented by the dotted lines in figure 4. Luckily such references tend to have a short access distance as well and we solve this by adding a second step to the scheduling process. For each processor p starting with the lowest ranked, find each task reachable from any leaf task scheduled to p by traversing the task graph with the edges reversed. Any task visited that was not already assigned to processor p is then moved to processor p. Tests show that the moved tasks do not influence the load balance of the schedule much.

When generating code for the individual processors there might be internal dependencies that dictate the order in which the tasks are laid out, which do not correspond to the order in which the results are needed by dependent tasks on other processors.

Assume t_1 and t_2 are assigned to processor p_1 and t_3 and t_4 are assigned to processor p_2. Assume further that there are dependencies between the tasks as shown in figure 5.
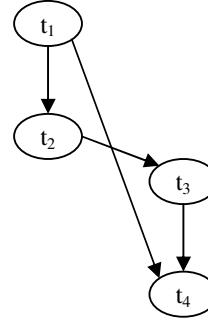

.

Fig. 5. Data dependency graph.

In this case there is no sense in scheduling a send operation from t_1 until t_2 is also done since no other processor can proceed with the result of t_1 alone. Therefore all send operations are postponed until there is one that another processor actually might be waiting for. Then all queued up send operations are merged into a single message which reduces the communication overhead.

8   Measurements

In order to evaluate the gained speedup we have used a model of a flexible shaft using a one-dimensional discretization scheme. The shaft is modeled using a series of n rotational spring-damper components connected in a sequence. In order to make the simulation task computationally expensive enough, to make parallelization worth while, we use a non linear spring-damper model.
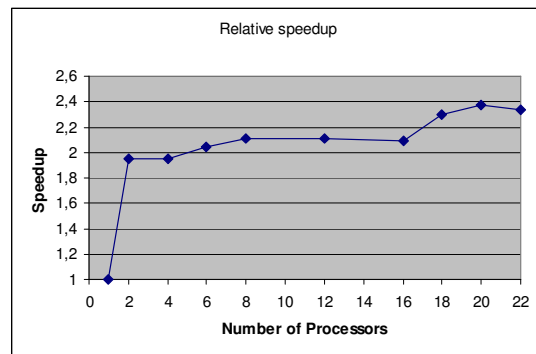

Fig. 6. Relative speedup on PC-cluster

In these tests we use a shaft consisting of 100 spring-damper elements connected together. The same model has bee used when the task merging approach was evaluated in [1], which makes it possible to compare

the results of this work to what was previously achieved.

The measurements were carried out on a 30-node PC cluster where each computation node is equipped with two 1.8 GHz AMD Athlon MP 2200+ and 2GB of RAM. Gigabit Ethernet is used for communication.

Figure 6 shows the results of the tests carried out so far. As can be seen the speedup for two processors is almost linear, but when the number of processors increase the speedup does not follow.

9 Conclusion

To conclude we can se that for two processors the tests were very promising, but those promises were not fulfilled when the number of processors increased. If we compare to the previous results obtained with task merging in [1], though, we do not suffer from slowdown in the same way (see figure 3). Most likely this has to do with the fact that the communication cost for the master process running the solver increases linearly with the number of processors whereas in our new approach this communication is distributed more evenly among all processors.

We have yet to do an in depth analysis of why the speedup for two processors is so good and way it does not scale as great.

10 Future work

In the nearest future we will profile the generated code to see were the bottlenecks are when ran on more than two processors and see if the scheduling algorithm can be tuned to avoid them. Also, tests must be carried out on different simulation problems to see if the results are general or if it differs much depending on the problem.

We also intend to port the runtime to run on threads in a shared memory setup. Since the trend is for CPU manufacturers to add more and more cores to the CPUs, it is becoming more and more relevant to explore parallelism in such an environment.

A runtime for the Cell BE processor is also planed. This processor has eight, so called, Synergistic Processing Elements (SPE) which do not actually share memory. Instead each SPE has it's own local memory. Transfers to and from those local memories can be carried out using DMA without using any computation resources, so it should be possible to hide the communication latency during computation.

11 Acknowledgements

12 References

[1] Peter Aronsson. *Automatic Parallelization of Equation-Based Simulation Programs*. PhD thesis, Dissertation No. 1022, Dept. Computer and Information Science, Linköping University, Linköping, Sweden.

[2] Olaf Bonorden, Ben Juurlink, Ingo von Otte, and Ingo Rieping. The Paderborn University BSP (PUB) Library. *Parallel Computing*, 29:187–207, 2003.

[3] Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. The OpenModelica Modeling, Simulation, and Software Development Environment. In *Simulation News Europe*, 44/45, December 2005. See also: http://www.ida.liu.se/projects/OpenModelica.

[4] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, 940 pp., ISBN 0-471-471631, Wiley-IEEE Press, 2004. See also book web page: www.mathcore.com/drModelica

[5] The Modelica Association. *The Modelica Language Specification Version 2.2*, March 2005. http://www.modelica.org.

[6] OpenMP Architecture Review Board. OpenMP: a Proposed Industry Standard API for Shared Memory Programming. White Paper, http://www.openmp.org/, October 1997.

[7] Joar Sohl. *A Scalable Run-time System for NestStep on Cluster Supercomputers*. Master thesis LITH-IDA-EX-06/011-SE, IDA, Linköpings universitet, 58183 Linköping, Sweden, March 2006.

[8] Kaj Nyström and Peter Fritzson. Parallel Simulation with Transmission Lines in Modelica. *In Proceedings of the 5th International Modelica Conference (Modelica'2006)*, Vienna, Austria, Sept. 4-5, 2006.

[9] Alexander Siemers, Dag Fritzson, and Peter Fritzson. Meta-Modeling for Multi-Physics Co-Simulations applied for OpenModelica. *In Proceedings of International Congress on Methodologies for Emerging Technologies in Automation (ANIPLA2006)*, Rome, Italy, November 13-15, 2006.

[10] Matthias Korch and Thomas Rauber. Optimizing Locality and Scalability of Embedded Runge-Kutta Solvers Using Block-Based Pipelining. *Journal of Parallel and Distributed Computing,* Volume 66 , Issue 3 (March 2006), Pages: 444 – 468.

[11] Peter Aronsson and Peter Fritzson. Automatic Parallelization in OpenModelica. In *Proceedings of 5th EUROSIM Congress on Modeling and Simulation*, Paris, France. ISBN (CD-ROM) 3-901608-28-1, Sept 2004.

[12] Christoph Kessler, Peter Fritzson and Mattias Eriksson. NestStepModelica: Mathematical Modeling and Bulk-Synchronous Parallel Simulation. *PARA-06 Workshop on state-of-the-art in scientific and parallel computing*, Umeå, Sweden, June 18-21, 2006.