Equation based parallelization of Modelica models

Marcus Walther Volker Waurich Christian Schubert Dr.-Ing. Ines Gubsch
Dresden University of Technology
{marcus.walther, volker.waurich, christian.schubert, ines.gubsch}@tu-dresden.de

Abstract

In order to enhance the performance of modern computers, the current development is towards placing multiple cores on one chip instead of inreasing the clock rates. To gain a speed-up from this architecture, software programs have to be partitioned into several independent parts. A common representation of these parts is called a task graph or data dependency graph. The authors of this article have developed a module for the OpenModelica Compiler (OMC), which creates, simplifies and schedules such task graphs. The tasks are created based on the BLT (block lower triangular)-structure, which is derived from the right hand side of the model equations. A noticeable speed-up for fluid models on modern six-core CPUs can be achieved.

Keywords: modelica; openmodelica; parallelization; BLT, task graph

1 Introduction

Modelica has become a widely used standard to describe physical simulation models. Compiling such a model into binary code can be performed by applications like Dymola, SimulationX or OpenModelica. However, all these tools only create a single thread simulation code out of standardized Modelica models, which does not allow for a speed-up with modern multi-core CPUs. This is due to the dependencies among the model equations which have to be considered in order to distribute the tasks amongst several threads.

The approaches to parallelize Modelica models can be divided into manual and automatic parallelization. Manual approaches comprise the *parModelica* language extension [1] or the TLM technique [2]. In this paper, manual parallelization shall not be pursued further as it is not suitable for parallelizing existing models. A lot of effort has been spent on automatic parallelization methods. Peter Aronsson [3] presented a method based on fine grained task graphs which were derived from the expressions of the model equations. Later, this approach was adapted by other authors (see for example [4], [2] and [5]), to perform simulations on Cell- and GPU-Architectures. Handling fine grained task graphs is a complicated and time consuming topic. Therefore additional work was required to reduce the graph complexity, for example with the help of a graph rewriting system [3]. This paper follows the ideas of Casella [6] who suggested to build a task graph parallelization based on the BLT representation of a model. He then also showed that, in case of fluid applications, this approach will lead to task graphs which can be well parallelized. Therefore, this paper explores the implementation of the ideas of [6] into the OpenModelica compiler. To evaluate the efficiency of the implementation, the idea of the *Maximum Theoretical Speedup* is introduced. Afterwards, different scheduling algorithms are presented which are required to assign each task to a thread. It is followed by a number of benchmarks which compare the effectiveness of the different scheduling algorithms and reveal further properties of different domains with respect to parallelization.

2 Parallelization of model equations

The equations of a simulation model are typically described as a set of Differential Algebraic Equations (DAEs). Equation 1 shows the basic definition of such a DAE.

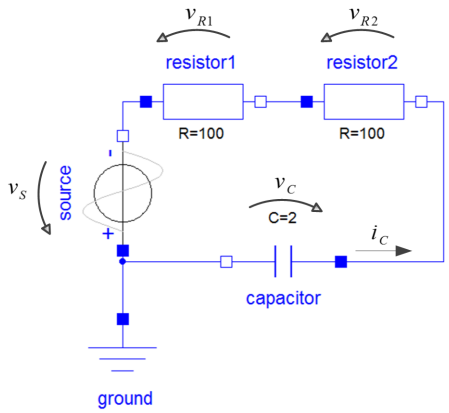
$$F(\boldsymbol{x}, \dot{\boldsymbol{x}}, \boldsymbol{v}, t) = 0 \tag{1}$$

The value t represents the time. The vector $\boldsymbol{x}$ holds all variables of the system whose derivatives with respect to time appear inside the equations. The derivatives itself are stored in $\dot{\boldsymbol{x}}$. In addition, $\boldsymbol{v}$ contains all other algebraic variables. By applying index reduction, (1) is converted into a DAE with index one or zero. The Underlying Ordinary Differential Equation (UODE) contains all equations and variables necessary to calculate the reduced state set $\dot{\boldsymbol{y}} \subseteq \dot{\boldsymbol{x}}$ of the model (see equation

2) and other equations (see equation 3) [6].

$$\dot{\boldsymbol{y}} = f(\boldsymbol{y},t) \quad (2)$$
$$\boldsymbol{v} = g(\boldsymbol{y},t) \quad (3)$$

The equations and variables of the ODE system can be organized as an incidence matrix, with each matrix-row representing one equation and each column one variable [7]. If a variable is part of an equation, the matrix entry is filled with a value. A simple electric circuit, containing a power supply and two resistors as well as a capacitor, is displayed in figure 1. It can be described by the equations below. The rows and



$$f_1 : v_s = offset$$
$$f_2 : v_{R1} = R_1 \cdot i_c$$
$$f_3 : P_{R1} = v_{R1} \cdot i_c$$
$$f_4 : v_{R1} = v_s - v_{R1n}$$
$$f_5 : i_c = C \cdot \dot{v}_c$$
$$f_6 : v_{R2} = R_2 \cdot i_c$$
$$f_7 : P_{R2} = v_{R2} \cdot i_c$$
$$f_8 : v_{R2} = v_{R1n} - v_c$$

Figure 1: *Simple model of an electrical circuit*

columns of the incidence matrix can be arranged in a way that the matrix forms a block lower trianguler matrix (BLT). Thus, the blocks of equations can be solved from the top to the bottom via forward substitution. First, the power supply voltage v_S is calculated from equation f_1. After that, the equations f_4, f_8, f_6 and f_2 of the circuit cannot be calculated as single equations, as they have two unknown variables. More precisely they are forming a circular dependency, because the variable u_{R1} is solved in equation f_2, which requires variable i_c, solved by equation f_6, for calculation. Furthermore, the calculation of f_6 depends on the equation f_8 which depends on equation f_4. And

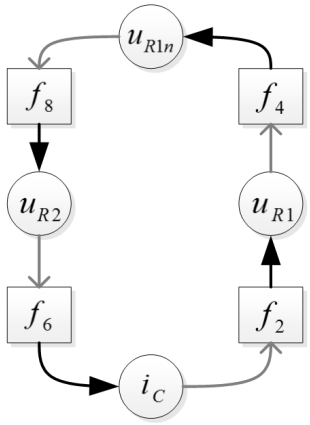


Figure 2: A bipartite graph representing the circual dependency between the equations f_2, f_4, f_6 and f_8

finally the equation f_4, solving variable u_{R1n}, requires u_{R1}, still solved by f_2. This fact is shown in Figure 2. That is why they have to be handled in an equation system which combines all equations into one block (see figure 3, gray coloured box). As shown in the example, blocks can be very simple, containing just one single equation or they can be really complex, containing hundreds of equation stored in an equation system. In order to solve the system efficiently, the block size

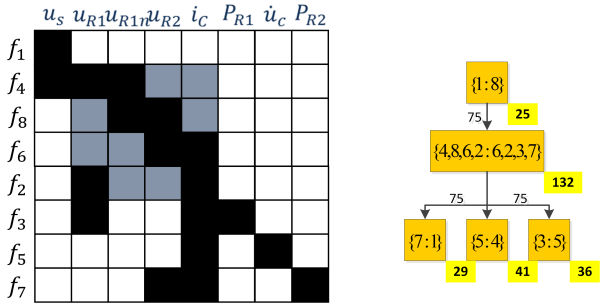


Figure 3: BLT-Matrix of the example on the left, derived task graph on the right side

should be as small as possible. To find the smallest blocks of the system, Tarjan's algorithm [8] can be used. A complete algorithm to get the blocks from the DAE-System is presented in [6].

3 Task Graph representation

A task graph or data dependency graph is a widely used technique to describe the different parts of a program and their relationships among each other. The graph contains nodes representing the tasks and directed edges. If an edge goes from node n_1 to n_2, the task n_1 has to be executed before the task n_2 can start.

DOI 10.3384/ECP140961213

Parallel branches of such a graph can be calculated in parallel, because there are no direct dependencies between them and thus they could be handled by different threads.
To get a task graph out of the BLT-structure, all blocks of the matrix are converted into a node of the graph. After that, the calculation dependencies between the blocks have to be inserted. An edge between the nodes of the blocks n_i and n_j is is added to the graph, if the BLT-matrix has an entry at position (i, j), with $i > j$ and (i, j) representing the row and column index of the blockmatrix, respectively.
The task graph of the given example circuit is displayed in figure 3. The notation inside the nodes is *{equation index : variable index}*. For the given example, the last three tasks could be handled in parallel by three different threads. In order to evaluate the simulation speed, both execution and communication costs for the tasks and processors have to be known. The execution cost of a task is the number of cycles or the time span required to calculate it. Communication costs are the time to transfer all required variables from one thread to another. To measure these values, two benchmark programs were developed. To estimate the communication costs, a standalone benchmark has been created which copies different sized data arrays of 64 bit long floating point value from one thread to another. By analyzing the task equations, the number of variables, which are transferred by each edge in the task graph, can be obtained. Thus a communication cost estimate can be assigned to each edge. The execution costs are being measured with the help of the OpenModelica *measure time* functionality for a serial calculation run, which creates a xml-file containing execution times for each block. This approach has still some drawbacks. First, it cannot be exact, as it is not possible to predict the occurence of cache misses or context switches between different processes. Second, one serial execution of the model is required, which may cause a severe overhead, before the estimates are available.
Execution costs are displayed on the bottom right corner of the nodes on a yellow background while communication costs are displayed near the edges, see Figure 3.

4 Graph simplification

Complex Modelica models may lead to complex task graphs with thousands of nodes and edges. Scheduling these graphs (see section 5) is a time consuming process, which can, depending on the scheduling algorithm, scale superlinear with the number of nodes and edges. Therefore the authors have implemented two rules to simplify complex task graphs, based on the ideas of [3]. The first one is a simple rule to merge chains of nodes with a maximum of one successor and one predecessor into one, as there is no point in calculating these nodes by different threads. The rule is called ”mergeSimpleNodes” and an example is displayed in figure 4. The second rule, which is more

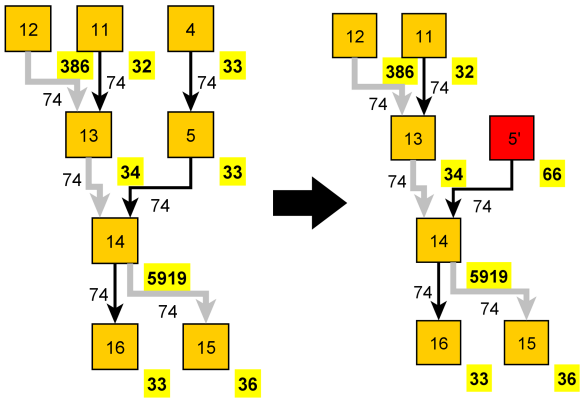


Figure 4: Example of the ”mergeSimpleNodes” rule. Node four and five are merged into one.

complex, is called ”mergeParentNodes”. It consolidates a node with its parents, if this leads to an decreasing execution time. See figure 5 for an example. If the nodes 11 and 13 are handled by different threads than task 12, the execution time increases compared to the serial execution. To prevent this, the nodes are merged into task 13′.

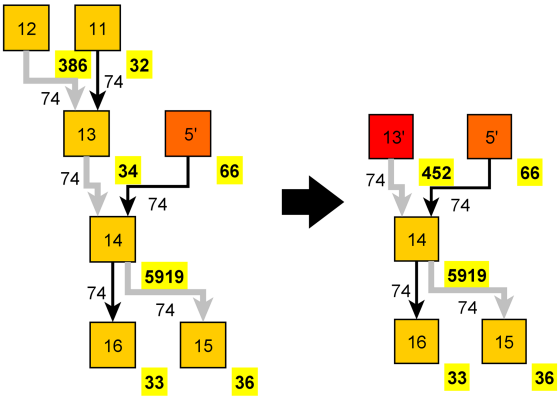


Figure 5: Example of the ”mergeParentNodes” rule. Node 11, 12 and 13 are merged into one.

5 Task Graph Scheduling

Based on the derived task graph, the different tasks can be dispersed among different threads, which can later be distributed on different processors or processor-cores. This is called Scheduling. In the presented work the Scheduling is performed during compile time (static scheduling). If the mapping between tasks and threads is set during run time, it is called a dynamic scheduling.
In order to achieve a proper mapping, most of the static scheduling algorithms use the information about the execution and communication costs to load the threads evenly and with shortest idle time. Finding the ideal schedule is a NP-hard problem [9]. Thus, different heuristics are used which have been implemented into the OpenModelica compiler module. To analyse the scheduling of a task graph several evaluation parameters can be obtained. First, some basic definitions shall be given. The *serial time* t_S of a model is the sum of all execution costs of all tasks T

$$t_S = \sum_{i \in T} t_i \tag{4}$$

The *minimum parallel time* t_{Pmin} is equal to the sum of the execution costs along the critical path, denoted as $crit'$

$$t_{Pmin} = \sum_{j \in crit'} t_j \tag{5}$$

This definition neglects all communication costs and corresponds to the case where all tasks of the critical path are assigned to the same thread. Further, it is assumed that all other tasks are handled in parallel by other threads not causing any delays. Clearly, this would require a sufficiently large number of computing cores. The *parallel time* t_P accounts for a limited number of computing cores and denotes the time required to calculate all tasks of a task graph given a schedule (assignment for each task to a thread or computing core) considering both execution as well as communication costs. The *Maximum Theoretical Speed-up* n_{max} can be obtained by dividing the serial time by the minimum parallel time

$$n_{max} = \frac{t_S}{t_{Pmin}} \tag{6}$$

assuming that an infinite number of computing cores is available. The *Theoretical Speed-up* n_t provides the expected speed-up for a given schedule. It is defined as the serial time divided by the parallel time

$$n_t = \frac{t_S}{t_P} \tag{7}$$

This quantity shall be used to compare different scheduling algorithms and to evaluate the implementation of the parallel code. In the following, different scheduling algorithms which will be compared are presented.

5.1 Level Scheduling

The simplest implemented scheduling algorithm is the level scheduling, which divides the graph into several layers. All tasks of one layer have just dependencies to tasks of previous layers. Thus, no direct dependencies between tasks of the same layer are allowed. The tasks of each layer are calculated in parallel until the algorithm proceeds with the next layer. Figure 6 shows a small graph example. Each layer is implemented as one OpenMP-Sections-Region and each task is handled in one OpenMP-Section. An example is displayed in listing 1.

Listing 1: Level scheduling code for graph in figure 6

```
static void solveODE(data) {
  //Level 1
  #pragma omc parallel sections {
    #pragma omc section {
      eqFunction_12(data);
    }
    #pragma omp section {
      eqFunction_11(data);
    }
    #pragma omp section {
      eqFunction_4(data);
    }
  }
  //Level 2
  #pragma omp parallel sections {
    #pragma omp section {
      eqFunction_13(data);
    }
    #pragam omp section {
      eqFunction_5(data);
    }
  }
  //Level 3
  #pragma omp parallel sections {
    #pragma omp section {
      eqFunction_14(data);
    }
  }
}
```

Therefore, OpenMP takes care of the concrete mapping between tasks and processors. This approach is following the ideas of a breadth-first-scheduling [10]. An eminent advantage of this scheduling method is that no information about execution and communication costs is needed. The scheduling is only based on the task dependencies. Thus, this algorithm is not fully static, but a hybrid of static and dynamic scheduling.

5.2 List Scheduling

The authors have also implemented a simple list scheduling algorithm [11], which can handle the tasks from root-nodes to leaf-nodes or vice versa. The list scheduling algorithm performs in the following way.

DOI 10.3384/ECP140961213

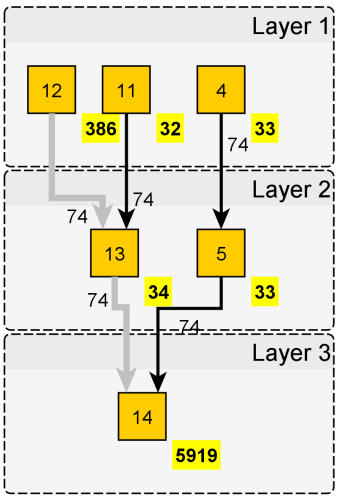


Figure 6: Example of level scheduling.

The root-nodes of the task graph are collected in a so-called 'ready list'. These tasks are distributed to all available threads. Before the first task assignment, every thread has a ready time of zero which means it is considered to be idle. If a task is assigned to an idle thread, the execution costs of the task will be added to its ready-time. The assignment of tasks enables the scheduling of their successor nodes which will be appended to the ready-list. The tasks from the ready list will be distributed successively to the thread with the earliest ready time. If the predecessor of an assignable task is scheduled to the same thread, the communication costs between these tasks are not taken into account. Otherwise, they have to be added to the ready time of the thread. The algorithm terminates when the ready list is empty.

5.3 Modified Critical Path Scheduling

Static scheduling algorithms are reviewed thoroughly. There is a multiplicity of approved scheduling heuristics and each performance depends on the structure of the graph, the dispersal of the costs etc. As an exemplary method, the 'Modified Critical-Path Scheduler' (MCP) by Wu and Gajski [12] has been implemented since this one is well-established as a reference heuristic[13]. The MCP distributes the tasks successively like the list scheduling to the thread that allows its earliest execution. The prioritisation of the assignable tasks is based on their ALAP-binding. The ALAP-binding stands for the as-late-as-possible start time and is computed as the longest path from the task to the finishing time of the last executed task.

5.4 External Scheduling

In order to understand and test the effect of different schedulings, the authors implemented a manual graph scheduler. The tasks can be assigend to the threads by hand on a graphical interface. For instance, the graph can be divided into vertical stripes, each handled by one thread like performed by libraries like metis [14].

5.5 Deadlock Detection

To check if a schedule is free of deadlocks, a transformation into a state / transition - petri net was developed. Every task of the graph is transformed into two states and one transition and every edge to one transition. An example for such a transformation is given in figure 7, where the task graph is shown on top and the petri net on bottom. The tasks of each thread are displayed one below the other. This kind of view gives more detailed information about the required locks between the different threads. If the final states of all threads are connected to a transition that is connected to the first states (not displayed in the figure), a petri net tool can check if the graph is free of deadlocks.

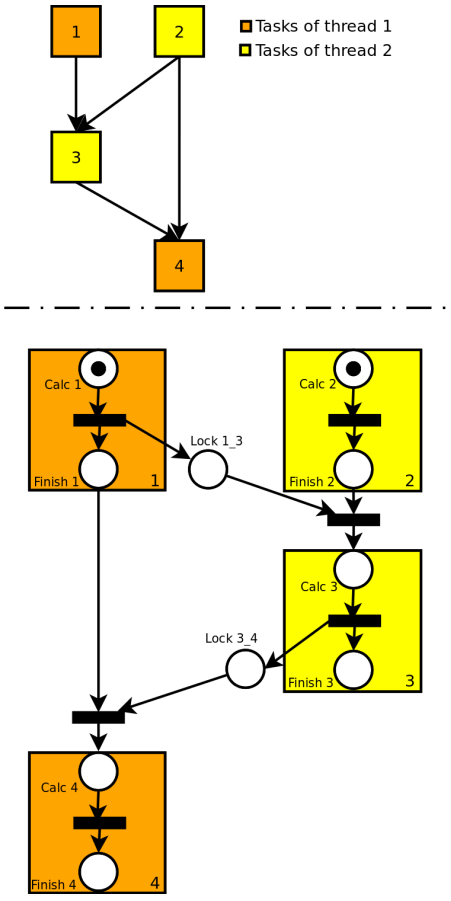


Figure 7: Example of state / transition net transformation

6 Benchmarks

To compare the different scheduling algorithms with the serial code, the simulation time of various models was measured. Three representative models from the domains Mechanics, Fluid and Electrics were selected out of the modelica standard library "MSL32". The first one is the engine V6 mechanic model. The second is the branching dynamic pipes example of the fluid domain. And the last model is the electrical cauer low pass sc model.
The test system was a computer with an Intel Core i7-3930K with six cores @ 3.20GHz and 32 gigabyte RAM running Windows 7 professional. All models were simulated from 0s to 1s using the dassl-solver.
In order to evaluate if it is possible to achieve shorter simulation times, the theoretical maximum speed-ups are displayed in table 1 for the three models.
The generated code of all scheduling algorithms, except the level-scheduling, is realized with pThreads using spin locks. Level-scheduling is based on an OpenMP implementation, as described in the previous section. Unfortunately, the results of level-scheduling were considerably slower than the other algorithms. Hence they were omitted from the diagrams for the sake of clarity.

Table 1: *Maximum Theoretical Speed-up* n_{max}

Modell	n_{max}
Engine V6	1.11
BranchingDynamicPipes	13.09
CauerLowPass	5.97

For the mechanical model it is not possible to achieve a speed-up at the moment. This is due to the graph's structure. Every multibody system requires the solution of the following linear system [15]

$$\boldsymbol{M}(\boldsymbol{q})\ddot{\boldsymbol{q}} = \boldsymbol{h}(\boldsymbol{q},\dot{\boldsymbol{q}}) + \boldsymbol{f}^a + \boldsymbol{G}^T(\boldsymbol{q})\boldsymbol{f}^c \quad (8)$$

$$\boldsymbol{0} = \boldsymbol{g}(\boldsymbol{q}) \quad (9)$$

The mass matrix $\boldsymbol{M}$ is in general densely populated. Its number of rows and columns is equal to the degrees of freedom of all the *tree-joints*, see [15]. The authors have noted that a large portion of the calculation time is spent on solving this linear system which corresponds to a single node in the task graph. In case of the EngineV6 model this can be up to 95% of the entire execution time. To take advantage of the BLT approach for such a model, one would have to either find a way to split up this task into smaller ones or to solve the task itself in parallel. This will be a topic of further research. For the fluid and electrical models, a speed-up is theoretically possible.
The benchmark results are displayed in the figures 8, 9 and 10. As already assumed, no speed-up with the mechanic model was found. The results of the fluid

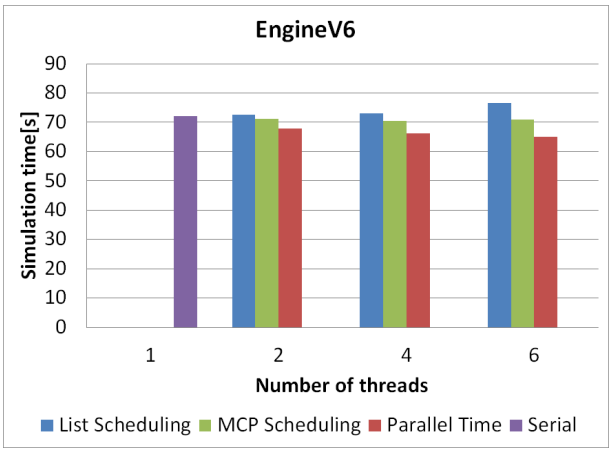


Figure 8: Benchmark of the engineV6 example

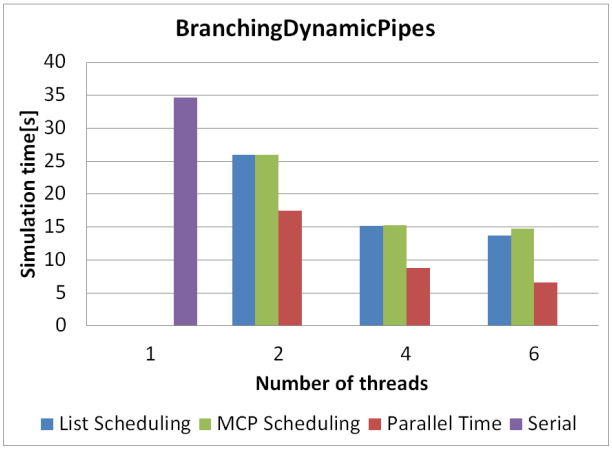


Figure 9: *Benchmark of the dynamic pipes example*

benchmarks indicates significant enhancement. The simulation showed a large step size, resulting in a lower number of calculations of the ODE functions. Carrying out such an ODE calculation takes a long time compared to the other examples. Moreover, the task graph has a lot of tasks which can be calculated in parallel. The trend of the Parallel Time indicates further potential. Additional research is needed in order to close the gap between predicted and measured speed-up. The low pass example shows currently no significant speed-up for the BLT parallelization, although the task graph has a lot of tasks in parallel, as can be seen in the behaviour of the Parallel Time. The issue appears to be the short execution time of the entire ODE system. The calculation of this sys-

DOI 10.3384/ECP140961213

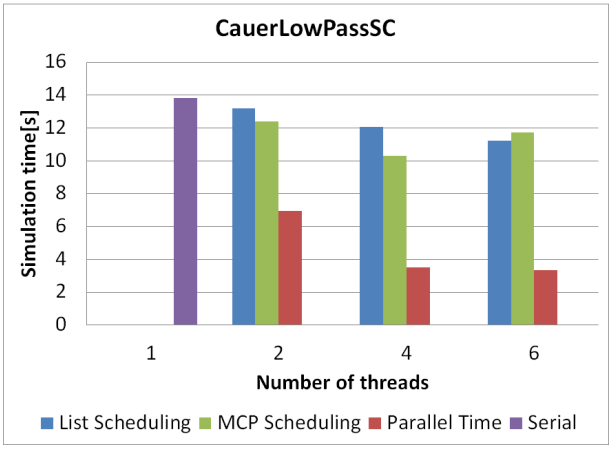


Figure 10: *Benchmark of the cauer low pass example*

tem is about 30 times faster than the calculation of the branching dynamic pipes ODE system. The overhead of the parallel code seems to be too big to cope with such short calculation cycles. At the beginning of the presented work, the parallel code was solely implemented with OpenMP. In consequence of the bad measurements of the cauer low pass example, the OpenMP code was exchanged with pThreads code using spin locks. The execution time of the parallel code could be halved for the cauer low pass, but is still too large to achieve a significant speed-up for the model.

7 Conclusion

The implementation has shown that the BLT parallelization approach is able to reduce the simulation time for some models, especially if they are part of the fluid domain. To achieve speed-ups for various models, further research is required, especially to reduce the overhead of the parallel code and to handle one big task using multiple cores. The problem with the big tasks can be solved by applying parallel solvers or by splitting up the complex task into simpler ones.
Furthermore, some memory analysis needs to be performed to reduce the number of cache misses and invalidations between the threads. At the moment, the variables are stored regarding their types (real, int or boolean) in different arrays. To get some improvements, the variables have to be organized regarding their task affinity.
All in all the presented BLT approach looks promising for a parallellization speed-up on ordinary shared memory systems.

References

[1] M. Gebremedhin, "Parmodelica: Extending the algorithmic subset ofmodelica with explicit parallel languageconstructs for multi-core simulation," *Linköping University Electronic Press, Linköpings universitet*, 2011.

[2] K. Stavåker, "Contributions to parallel simulation of equation-based models on graphics processing units," *Linköping University, Sweden*, 2011.

[3] P. Aronsson, *Automatic Parallelization of Equation-Based Simulation Programs*. Institutionen för datavetenskap, 2006.

[4] 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," *SIGARCH Comput. Archit. News*, vol. 36, pp. 46–55, June 2009.

[5] P. Östlund, "Simulation of modelica models on the cuda parallel architecture," *Linköping University Electronic Press, Linköpings universitet*, 2010.

[6] F. Casella, "A strategy for parallel simulation of declarative object-oriented models of generalized physical networks," *Linköping University Electronic Press, Linköpings universitet*, 2013.

[7] F. Cellier and E. Kofman, *Continuous System Simulation*. Springer, 2006.

[8] R. E. Tarjan, "Depth-first search and linear graph algorithms.," *SIAM J. Comput.*, vol. 1, no. 2, pp. 146–160, 1972.

[9] M. R. Garey, D. S. Johnson, and R. Sethi, "The complexity of flowshop and jobshop scheduling," *Math. of Op. Res.*, vol. 2, pp. 117–129, May 1976. ctr127.

[10] A. Duran, J. Corbalán, and E. Ayguadé, "Evaluation of openmp task scheduling strategies.," in *IWOMP* (R. Eigenmann and B. R. de Supinski, eds.), vol. 5004 of *Lecture Notes in Computer Science*, pp. 100–110, Springer, 2008.

[11] U. Banerjee, "Parallelization, basic block," in *Encyclopedia of Parallel Computing* (D. A. Padua, ed.), pp. 1450–1458, Springer, 2011.

[12] M.-Y. Wu and D. D. Gajski, "Hypertool: A programming aid for message-passing systems," *IEEE TRANS. ON PARALLEL AND DISTRIBUTED SYSTEMS*, vol. 1, pp. 330–343, 1990.

[13] A. Radulescu and A. J. C. van Gemund, "Flb: Fast load balancing for distributed-memory machines.," in *ICPP*, pp. 534–541, 1999.

[14] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, pp. 359–392, Dec. 1998.

[15] H. Elmqvist and M. Otter, "Methods for tearing systems of equations in object-oriented modeling," in *ESM*, vol. 94, p. 1–3, 1994.

DOI 10.3384/ECP140961213