D5.1	Develop efficient multicore simulation
Access[1]:	**PU**
Type[2]:	**SW**
Version:	**1.0**
Due Dates[3]:	**M24**

Open Cyber-Physical System Model-Driven Certified Development

Executive summary[4]:

This deliverable reports on the support for large scale and multicore simulation in OpenModelica.

For speeding up simulations using multicore computers, automatic parallelization can be employed but speedup depends a lot on the model structure.

For speeding up compilation and simulation of large models, alternative solutions for solving the systems of equations can be employed, using sparse solvers and DAE instead of ODE solvers.

Both solutions have been implemented as prototypes in OpenModelica.

[1] Access classification as per definitions in PCA; PU = Public, CO = Confidential. Access classification per deliverable stated in FPP.
[2] Deliverable type according to FPP, note that all non-report deliverables must be accompanied by a deliverable report.
[3] Due month(s) according to FPP.
[4] It is mandatory to provide an executive summary for each deliverable.

Deliverable Contributors:

	Name	Organisation	Primary role in project	Main Author(s)[5]
Deliverable Leader[6]	Adrian Pop	LIU	WP5 Leader	x
Contributing Author(s)[7]				
Internal Reviewer(s)[8]	Bernhard Thiele	LIU	WP3 Leader	

Document History:

Version	Date	Reason for Change	Status[9]
0.1	23/11/2017	First Draft Version	Draft
1.0	24/11/2017	First Issue	Released

[5] Indicate Main Author(s) with an “X” in this column.
[6] Deliverable leader according to FPP, role definition in PCA.
[7] Person(s) from contributing partners for the deliverable, expected contributing partners stated in FPP.
[8] Typically person(s) with appropriate expertise to assess deliverable structure and quality.
[9] Status = “Draft”, “In Review”, “Released”.

CONTENTS

ABBREVIATIONS

List of abbreviations/acronyms used in document:

Abbreviation	**Definition**
FMI	Functional Mock-up Interface
FMU	Functional Mock-up Unit
M&S	Modelling and Simulation
N/A	Not Applicable

1 EFFICIENT COMPILATION AND SIMULATION

Compiling and running simulations as efficient as possible is always desirable to minimize cost. In this deliverables we briefly present two strategies for speeding the compilation and simulation of models.

1.1 Automatic parallelization on multicore

The automatic parallelization on multicore is presented in detail in [1]. The approach is based on a task system representation and a parallelization C++ library called ParModAuto presented in [6], extended with continuous runtime profiling and scheduling. The execution of ODE during simulation varies with time and a static schedule defined before or at the start of the simulation will not stay efficient during the entire simulation (see Figure 1. BranchingDynamicPipes ODE evaluation cost). Continuous profiling and scheduling can improve the efficiency of parallel execution during simulation.

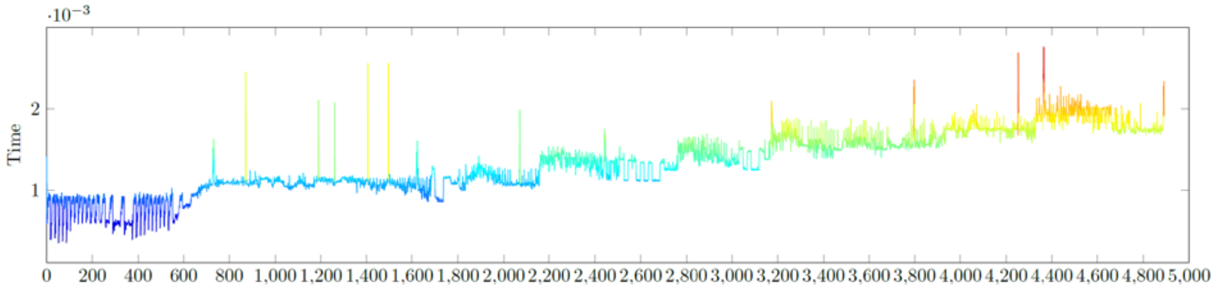


Figure 1. BranchingDynamicPipes ODE evaluation cost

The parallelization approach focuses on Strongly Connected Components supported by the ParModAuto task parallelization library. ParModAuto was designed to be a complete runtime library. It has runtime profiling and adaptive rescheduling capabilities. This means the library can be used to perform more responsive adaptive parallel simulations which is not possible with static compile-time approaches. ParModAuto is sufficiently high level to be used without requiring advanced knowledge in parallel programming. It is also designed with portability in mind, using Intel Intel®Threading Building Blocks Library (TBB) and Boost Graph Library, so that it can be used by multiple simulation tools. It can save time and effort for developers and researchers that intend to achieve either a complete task parallelization implementation or try out different scheduling approaches.

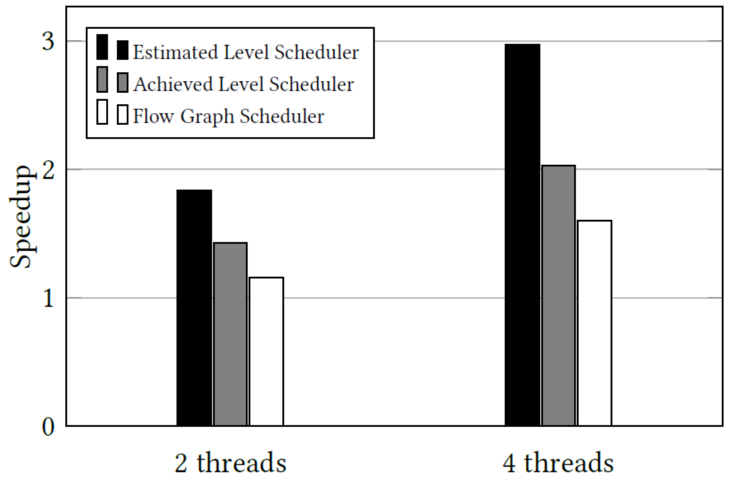


Figure 2. Speed-up for CauerLowPassSC model

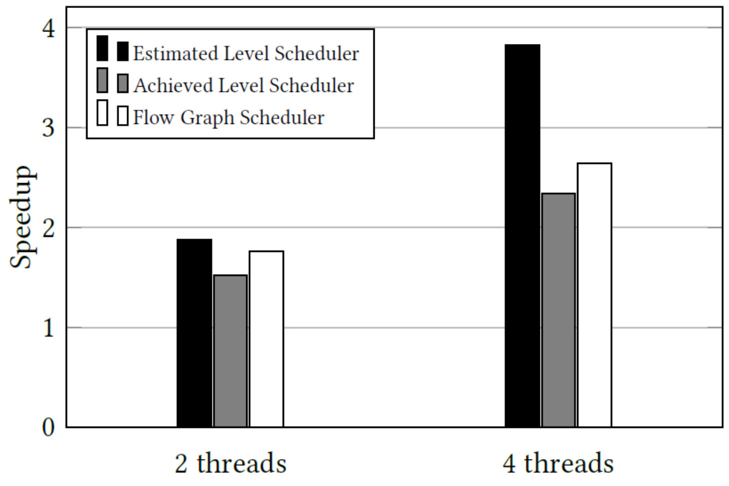


Figure 3. Speed-up for BranchingDynamicPipes model

The performance has been investigated with a prototype implementation in the OpenModelica Modeling and Simulation Environment and shows promising results. Tests have been performed, using models from the Modelica Standard Library (MSL), to evaluate the performance of the prototype implementation with the OpenModelica compiler. All tests have been performed on a 64-bit Intel® Core™ i7-3930K CPU@3.20GHz machine running Ubuntu 16.04.3 LTS. The time results presented do not include model compilation time. Actual simulation times are used in the analysis. However, it should be noted that all parallelization related execution times are included. These include all the extra overhead from task system creation, profiling, clustering and scheduling. Therefore, the timing results here are what users should expect when running simulations normally. Simulations are performed from time 0 to 1 second. All simulations are done using the default solver of OpenModelica environment, DASSL. For each model we present the estimated speed-up of the Level Scheduler, the achieved speed-up of the Level Scheduler, and the speedup for the Flow Graph based scheduler. The estimated speed-up for the Level Scheduler is the ratio of the sequential cost to the ideal parallel cost. Sequential cost is obtained by adding up the costs of all individual tasks in the system while the ideal cost is obtained by adding up the costs of the largest tasks at each level of task graph. Timing results for two models from the Modelica Standard Library are presented here. The first model is the fifth order lowpass-filter CauerLowPassSC. Measurements for this model are presented in Figure 2. A speed-up of factor 2.0262 is achieved using the Level Scheduler execution with 4 threads. Results for the second test model, BranchingDynamicPipes, are presented in Figure 3. This model achived a speed-up of factor 2.6445 using the Flow Graph Scheduler with 4 threads.

To compile a model with automatic parallelization using the OpenModelica compiler one just needs to use a special flag: `-d=parmodauto` when building the simulation code. See also the OpenModelica documentation:
https://openmodelica.org/doc/OpenModelicaUsersGuide/latest/omchelptext.html#omcflag-debug-parmodauto

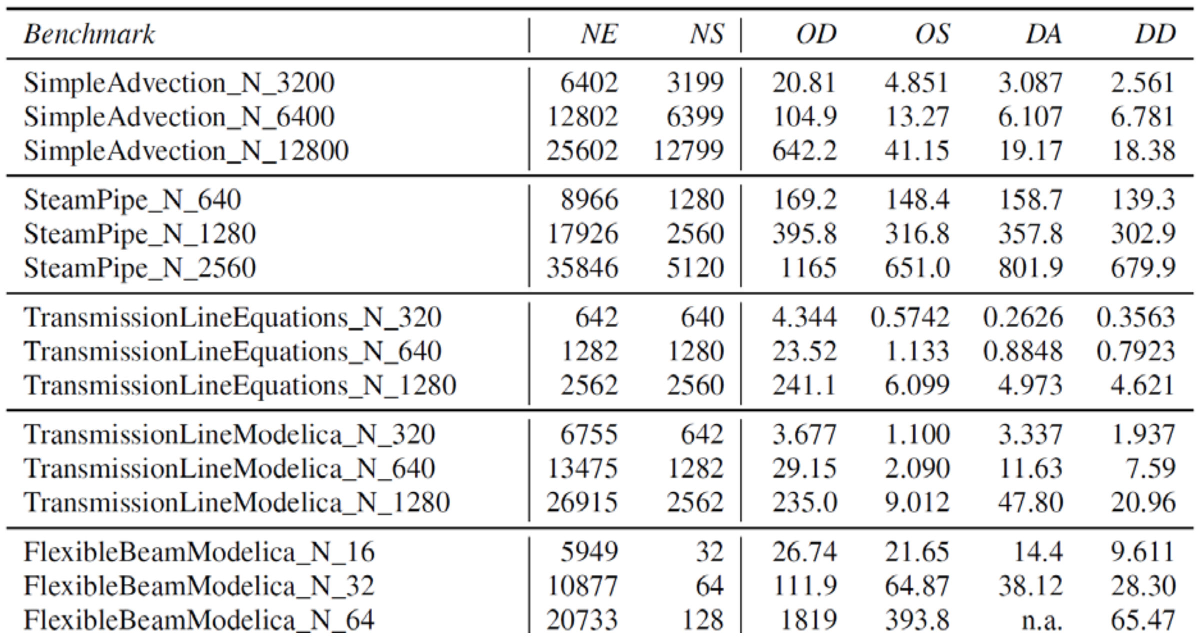
During year 3 of OpenCPS project we will focus on testing the automatic parallelization on the provided benchmarks and demonstrators and improve it further.

1.2 Alternative strategies for solving large-scale models

The proposed handling of large-scale models using DAE mode and sparse solvers is presented in detail in [2].
The ease of model composition in Modelica makes it easy to develop very large systems. The system size of interest in a Modelica-based simulation is continuously increasing and the traditional way of generating simulation code, e.g. involving symbolic transformations like matching, sorting, and tearing, must be adapted to this situation.
This section describes recently implemented sparse solver techniques in OpenModelica in order to efficiently compile and simulate large-scale Modelica models. Performance is evaluated on selected models from ScalableTestSuite. The ScalableTestSuite contains a number of different benchmark models, whose size can be easily chosen by setting one or more Integer parameters. The benchmarks are designed to stress some aspect of the code generation and execution, e.g. by possessing large implicit systems of algebraic equations, large number of states, large number of event-generating functions, etc.

The performance is presented in Figure 4 for a selection benchmark models, each one coming in three different sizes. The results obtained with four different numerical solution strategies are presented and compared. The table shows the number of equations *NE*, number of states *NS* and the running times of the simulations in seconds, including the time spent for initialization.

The first solution strategy, labelled *OD* in the result table, is the default approach to solving Modelica models implemented in the OpenModelica tool. The DAEs are turned into ODEs by solving them for the derivatives, using the BLT transformation to do so efficiently, applying symbolic index reduction if the system has index greater than one. The implicit equations in the BLT corresponding to strong components in the dependency graph are solved with dense linear and nonlinear equation solvers, using tearing to reduce the size of the implicit part of the problem and thus somehow exploiting sparsity. The ODEs are then solved by the DASSL BDF integrator, using a dense linear solver for its internal operations.



Benchmark	*NE*	*NS*	*OD*	*OS*	*DA*	*DD*
SimpleAdvection_N_3200	6402	3199	20.81	4.851	3.087	2.561
SimpleAdvection_N_6400	12802	6399	104.9	13.27	6.107	6.781
SimpleAdvection_N_12800	25602	12799	642.2	41.15	19.17	18.38
SteamPipe_N_640	8966	1280	169.2	148.4	158.7	139.3
SteamPipe_N_1280	17926	2560	395.8	316.8	357.8	302.9
SteamPipe_N_2560	35846	5120	1165	651.0	801.9	679.9
TransmissionLineEquations_N_320	642	640	4.344	0.5742	0.2626	0.3563
TransmissionLineEquations_N_640	1282	1280	23.52	1.133	0.8848	0.7923
TransmissionLineEquations_N_1280	2562	2560	241.1	6.099	4.973	4.621
TransmissionLineModelica_N_320	6755	642	3.677	1.100	3.337	1.937
TransmissionLineModelica_N_640	13475	1282	29.15	2.090	11.63	7.59
TransmissionLineModelica_N_1280	26915	2562	235.0	9.012	47.80	20.96
FlexibleBeamModelica_N_16	5949	32	26.74	21.65	14.4	9.611
FlexibleBeamModelica_N_32	10877	64	111.9	64.87	38.12	28.30
FlexibleBeamModelica_N_64	20733	128	1819	393.8	n.a.	65.47

Figure 4. Simulation times of ScalableTestSuite benchmarks in seconds

The second strategy, labelled *OS*, still resorts to causalization; however, the implicit equations corresponding to the strong components in the BLT are solved by the Kinsol/KLU sparse solvers, while the ODEs are solved by the IDA BDF integrator, relying on the KLU sparse linear solver internally. In this case, tearing is not applied to solve the implicit equations corresponding to strong components in the BLT. The rationale behind this decision is that on the one hand, the sparse solver already vastly reduces the computational complexity, if the system is highly sparse. On the other hand, tearing very large systems might take a disproportionately large amount of time by the compiler back-end, so that the time savings at run time are likely to be more than offset by the much longer code generation time.

The third strategy, labelled *DA*, is to only apply symbolic index reduction (if needed) to the DAEs, and then use the sparse IDA solver directly to solve them over time.

The fourth strategy, labelled *DD*, is a variant of the former one, in which the subset of the DAEs that is strictly required to be solved in order to compute the state variables at the next time step is identified and passed to the sparse IDA solver. Once the new time step has been computed and accepted as valid by the error estimation routine, the remaining equations are solved for the remaining variables by OpenModelica-generated code, exploiting the usual BLT decomposition to solve them efficiently. This strategy can be advantageous because it avoids computing unnecessary variables during the internal solver iterations, particularly when tentatively computing a step that may then be rejected by the error estimation routine. Also, it is often the case that the dependencies in the systems are such that, once the variables required to advance the states have been computed, the remaining ones can be computed by explicit assignments. Furthermore, these are only computed once instead of getting unnecessarily involved many times in the iterative solution of the implicit sparse nonlinear DAEs.
As to the initialization problem, with the first strategy the standard dense linear and nonlinear solvers with tearing are used; with the other three, the sparse solvers Kinsol/KLU without tearing are used instead.

All tests were carried out on the Open Source Modelica Consortium continuous testing infrastructure, using the development version 1.12.0 of OpenModelica. The computer used to run the tests is a 16-core Intel i7-6900K CPU @ 3.20 GHz, with 132 GB RAM.

The speedup achieved for the DD strategy varies from 2 to 60 times faster than the *OD* strategy, which is currently the default in OpenModelica.

The main result of this study is that the use of sparse solvers is almost always beneficial, sometimes very substantially, over the traditional use of dense solvers supported by symbolic manipulation. The comparison between sparse DAE solvers and sparse ODE solvers has many different outcomes, depending on the specific problems at hand.
Another interesting result is that we have demonstrated the feasibility of using such sparse solvers to successfully simulate Modelica models of industrially relevant systems with size up to over half a million DAEs.

To compile a model with DAE mode the OpenModelica compiler one just needs to use special flags: `--daeMode=dynamic` when building the simulation code. Additional flags are needed when running the built executable code: `-mei=4000 -daeMode -s=ida -idaLS=klu`. Please refer to OpenModelica documentation for more information about these flags:
https://openmodelica.org/doc/OpenModelicaUsersGuide/latest/omchelptext.html#omcflag-daemode
https://openmodelica.org/doc/OpenModelicaUsersGuide/latest/simulationflags.html#simflag-mei

https://openmodelica.org/doc/OpenModelicaUsersGuide/latest/simulationflags.html#simflag-daemode

During year 3 of the OpenCPS project we will focus testing the DAE mode on the benchmarks and demonstrators and improve it further.

REFERENCES

[1] Mahder Gebremedhin, Peter Fritzson “Parallelizing Simulations with Runtime Profiling and Scheduling”, 8th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, Munich, Germany, 2017.
[2] Willi Braun, Francesco Casella, Bernhard Bachmann “Solving Large-scale Modelica Models: New Approaches and Experimental Results using OpenModelica”, 12th International Modelica Conference, Prague, Czech Republic, 2017.
[3] Hilding Elmqvist, Sven Erik Mattsson, Hans Olsson “Parallel Model Execution on Many Cores”. 10th International Modelica Conference, Lund, 2014
[4] Martin Sjölund, Mahder Gebremedhin, Peter Fritzson “Paralellizing equation-based models for simulation on multi-core platforms by utilizing model structure”. 17th Workshop on Compilers for Parallel Computing.
[5] Martin Sjölund, Robert Braun, Peter Fritzson, Peter Krus “Towards Efficient Distributed Simulation in Modelica using Transmission Line Modeling”. Proceedings of the 3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT'2010). Oslo.
[6] Mahder Gebremedhin, Peter Fritzson: “Automatic task based analysis and parallelization in the context of equation based languages”. EOOLT 2014