Parallelization approaches for the simulation of large-scale multibody systems

M. Klöppel[1,*], M. Flehmig[2], A. Naumann[1], M. Walther[2], V. Waurich[3], J. Wensch[1]

Abstract— Virtual prototyping plays an important role in the engineering disciplines. The possibility to model and simulate prototypes on a computer instead of building real-world ones saves time and money. Nowadays, engineers can rely on special tools like object-oriented modeling languages, e.g., Modelica, to describe their models. These models can be automatically processed and simulated using standard Differential Algebraic Equation (DAE) solvers. The advantage of this approach is that the practitioners can concentrate themselves on modeling, whereas the numerical intricacies of the simulation are handled by the software. The disadvantage is that such simulations are usually slower than implementations which are parallelized and optimized by hand. In this contribution, we concentrate on the widely used simulation software OpenModelica, which is open source and thus appropriate to evaluate several parallelization approaches.The implemented methods are demonstrated on engineering examples.

Index Terms— Modeling, Multibody Systems Simulation, Parallelization, Task Graph Parallelism

1 Introduction

THE advent of virtual prototyping had a decisive influence on the engineering sciences. The possibility to simulate complex and large-scale models instead of building real-world prototypes allows to save costs and shorten development cycles. Nowadays, a plethora of commercial and open-source simulation software exists. While the current release of these programs is quite powerful, simulating large-scale and/or multi-domain models is still computationally challenging, i.e., even the simulation of short time periods might take a considerable amount of computation time. The parallel execution of the computational workload (under consideration of data dependencies) using modern multi-processor hardware is one way to reduce the computation time.

In this paper we evaluate several parallelization approaches. Our main goal here is to give suggestions to the developers of simulation codes, i.e., we are not looking for the best way to parallelize a specific problem but consider methods, which are applicable to a wide range of systems. This approach results in some special requirements on the investigated methods. Besides the obvious requirement of speeding up the computation, we focused on methods that can be applied without the interaction of the user. Here, we concentrate on the special case of (rigid) multibody system simulation and the open-source software OpenModelica (http://openmodelica.org), but the suggested algorithms will be useful in other simulation codes as well.

We use the suggestions in [1] as inspiration for choosing parallel methods. In addition we consider general linear methods in the form of peer methods [2]. There have been other attempts of parallelization in the Modelica field, namely ParModelica [3] and Transmission Line Modelling (TLM) [4]. These approaches are not considered here for the following reasons. ParModelica extends the Modelica language standard by parallel constructs, i.e., the user has to specify tasks, which can be carried out in parallel. This contradicts our second requirement. TLM decouples the model using so called transmission line elements and solves the resulting system using co-simulation approaches. This again requires user interaction for defining the elements, which requires a significant amount of in-depth knowledge. Additionally, the TLM system is not algebraically equivalent to the original problem, i.e., the TLM approach might result in a completely different solution.

Here, we consider general multibody systems with the following equations of motion

$$M(q)\ddot{q} = f(t,q,\dot{q}) - \left(\frac{\partial g(q)}{\partial q}\right)^T \lambda, \tag{1}$$

$$0 = g(q), \tag{2}$$

where $q(t) \in \mathbb{R}^n$ are position coordinates, $M(q(t)) \in \mathbb{R}^{n\times n}$ the positive definite mass matrix, $t \in [t_0, t_e] \subset \mathbb{R}$ the time, and $f: \mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$ describes the applied forces. If the described system contains kinematic loops, the model equations contain additional constraints of the form (2), where $g: \mathbb{R}^n \to \mathbb{R}^m$, and Langrange multipliers $\lambda(t) \in \mathbb{R}^m$. Within the OpenModelica framework a system of form (1), (2) is transformed into an explicit Ordinary Differential Equation (ODE) of the form

$$\dot{x} = \tilde{M}(x)^{-1}\tilde{f}(t,x) \tag{3}$$

where $x \in \mathbb{R}^{\tilde{n}}$, $\tilde{M}(x) \in \mathbb{R}^{\tilde{n}\times\tilde{n}}$ and $\tilde{f}: \mathbb{R} \times \mathbb{R}^{\tilde{n}} \to \mathbb{R}^{\tilde{n}}$, using index reduction techniques. A more concise description of this process is given in the next section. Please note, that usually $n \neq \tilde{n}$ can be taken for granted.

The remainder of this paper is structured as follows. Section 2 contains a general overview of the OpenModelica software and describes how the model equations are derived and the index reduction to the ODE-case is carried out. Section 3 introduces the benchmarks used throughout the remainder of this

1 Institute of Scientic Computing, TU Dresden, 01062 Dresden, Germany
2 Center for Information Services and High Performance Computing, TU Dresden, 01062 Dresden, Germany
3 Chair of Construction Machines and Conveying Technology, TU Dresden, 01062 Dresden, Germany
* Corresponding author: michael.kloeppel@tu-dresden.de

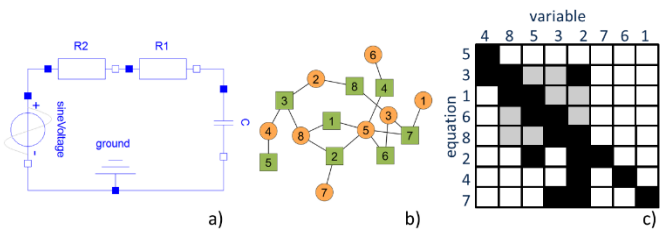

Fig. 1 Different stages of a Modelica model in the compilation process a) graphical model representation b) bipartite graph of flat Modelica c) BLT matrix of causalized system.

work. Several parallelization approaches for multibody system simulation are discussed in Section 4. Section 5 summarizes the presented parallelization techniques and draws conclusions.

2 The OpenModelica environment - from model description to ODE

OpenModelica is an open-source Modelica-based modeling and simulation environment. The core of OpenModelica is a capable Modelica compiler, which transforms a textual model description into an executable simulation program. Modelica is a model description language which allows modeling of differential algebraic equations both in the continuous and discrete time domain. The Modelica modeling language features object-orientation and acausal modeling (AM). AM means that models are described with equations that can be rearranged and derived. Causal modeling approaches, e.g., in block-diagram-modeling have to consist of algorithms which already have a direction of computation and, therefore, cannot be rearranged. To give an example for the power of AM, consider the equations of a simple electrical engine. Depending on the given variables, the same equations can also be used to describe a generator. The compiler figures out how the equations have to be rearranged in order to derive a solvable model. The object-oriented approach makes modeling very convenient on a graphical user interface. There are various libraries which can be used to model complex systems in a clear, hierarchical structure as can be seen in Fig. 1 a) for the example of a simple electrical circuit. The particular elements, as for example the capacitor, represent a textual model description containing variables as v or i and equations as $i = C * der(v)$. Connections of elements lead to equations, which sum up the fluxes to zero and equalizes the potential variables of the connect ports.

The compilation process of a Modelica Compiler is as follows: The textual Modelica model is parsed and the objects are instantiated. The objectoriented, hierarchical structure is destroyed and the result is a list of all equations and variables, i.e., the flat Modelica model. This model can be represented as a bipartite graph. Fig. 1 b) shows the bipartite graph of the circuit model, where the square nodes represent the equations and the circular nodes represent the variables. If a variable exists in an equation, there is an edge between the corresponding nodes. The model is still acausal and the next step is to determine a computation sequence. Therefore, each variable has to be assigned to an equation, which is able to solve it. In graph theoretical terms, a perfect matching has to be identified. If the system is a non-singular ODE or DAE system, every equation-node can be assigned to an adjacent variable node. In case of higher index systems, an index reduction method has to be applied, in order to convert the system into an ODE or index 1 system. Typically the Panthelides index reduction extended with the dummy derivatives method is applied [5]. In order to determine the computation sequence of the system, the adjacency matrix of the system is rearranged. The adjacency matrix contains the same information as the bipartite graph, where the rows correspond to the equations and the columns to the variables. If the matrix can be transformed to a lower triangular matrix, the main diagonal contains the matching information and the equations have to be computed from the first to the last row in order to solve the system. A strictly lower triangular form cannot be established if there are algebraic loops, which is in general the case. Thus, only a block-lower-triangular form (BLT form) can be determined as depicted in Fig. 1 c). The matrix entries inside the upper triangular part form blocks, which correspond to strongly connected components (SCCs) in the bipartite graph. By applying Tarjans algorithm, these strongly connected components can be determined [6]. If the BLT form is derived, the computation sequence is represented by the entries and blocks on the main diagonal. Blocks that belong to just one entry belong to single equations and blocks that contain multiple entries belong to equation systems, which have to be solved as a whole. In order to solve the DAE system, the derivatives of the state variables have to be computed. If all state derivatives are solved, a numerical time integration method can be applied to compute the state variables of the next time step.

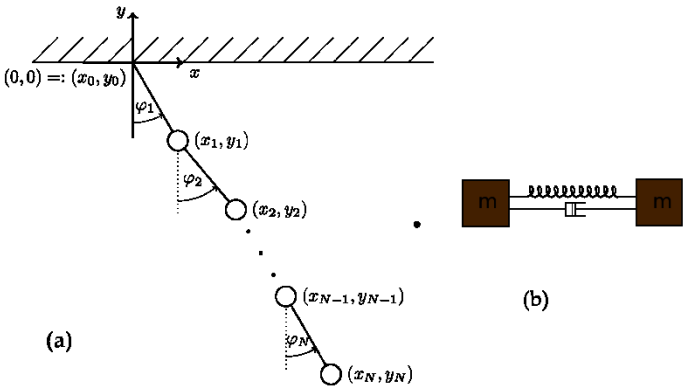

Fig. 2 The N-Pendulum (a) and a part of the Spring-Mass-Damper network (b).

3 Benchmarks

The parallelization approaches are tested on the following two benchmark examples, which are extremes in a certain sense. In the N-pendulum example, the mass matrix M is fully populated whereas in the Spring-Mass-Damper network it is just a diagonal matrix and, therefore, easily invertible. Furthermore, the Jacobian is dense in the N-pendulum case while it is sparse in Spring-Mass-Damper network case. Real world examples usually fall between these two extremes. Both problems are depicted in Fig. 2. All computational experiments were conducted

on the Taurus high performance computing system at TU Dresden, which, at the time of writing, is listed on rank 77 of the top 500 supercomputers (http://www.top500.org/list/2015/11/, accessed on December 1st, 2015.). The newest extension of Taurus consists of Xeon E5-2680v3 processors with 12 cores per processor and a clock speed of 2.5 GHz. Furthermore, each processor core has access to a maximum of 1.7 GB of RAM. One of these processors was used througout the measurements. The experiments ran exclusively on this processor with a maximum of eight CPU cores. All experiments used the C++ runtime within OpenModelica, unless otherwise mentioned. The models were translated using OpenModelica version 1.9.3 and the generated code was compiled using g++ version 4.9. Only the optimization flag "-O0" was used. Higher optimizations levels are not feasible (at least at the present state of the C++ runtime), because they significantly increase the compilation time. Since CVode [7] is the standard ODE-solver within the runtime, we take CVode computation times as reference.

3.1 N-pendulum

The first problem is a mathematical N-pendulum, which is given by

$$\begin{pmatrix}\dot{\varphi}\\ \dot{\psi}\end{pmatrix} = \begin{pmatrix}\psi\\ M(\varphi)^{-1}F(\varphi,\psi)\end{pmatrix},$$

where $M(\varphi) \in \mathbb{R}^{N\times N}, F(\varphi,\psi) \in \mathbb{R}^N$ with

$$M_{jk}(\varphi) = (N - \max(j,k) + 1)\cos(\varphi_k - \varphi_j),$$

$$F_j(\varphi,\psi) = \frac{g}{l}(N - j + 1)\sin\varphi_j + (N - j + 1)\sum_{k=1}^{j-1}\sin(\varphi_k - \varphi_j)\,\dot{\varphi}_k^2 + \sum_{k=j+1}^{N}(N - k + 1)\sin(\varphi_k - \varphi_j)\,\dot{\varphi}_k^2,$$

Here, φ is the angle as shown in Fig. 2 and ψ the angular velocity. The parameters g and l are the standard gravity and the length of a segment of the N-pendulum, respectively.

3.2 Spring-mass-damper network

The second problem is a spring-mass-damper network containing N masses. The equations of motion are given by

$$\begin{pmatrix}\dot{\eta}\\ \dot{\xi}\end{pmatrix} = \begin{pmatrix}\xi\\ M^{-1}F(\eta,\xi)\end{pmatrix},$$

where $M \in \mathbb{R}^{N\times N}$, $F(\eta,\xi) \in \mathbb{R}^N$ with $M = diag(m_1,\dots,m_N)$,

$$F_j(\eta,\xi) = \sum_{i\in N(j)}\left[-k_{ij}\frac{\eta_i-\eta_j}{\|\eta_i-\eta_j\|}\left(\|\eta_i-\eta_j\| - l_{ij}\right) - \mu_{ij}(\xi_i-\xi_j)\right]$$

Here, η are the positions whereas ξ are the velocities of the masses. The parameters k_{ij} and μ_{ij} describe the spring constant and damping factor of the spring-damper connecting the masses i and j, respectively, whereas l_{ij} describes its length. The function $N(j)$ returns all neighboring masses of mass j, i.e.,all masses i, which are directly connected to mass j via a spring-damper.

4 Parallelization approaches

As mentioned in the introduction, we are following the suggestions of Burrage [1] and consider the following approaches to parallelization:

1. parallel linear algebra,
2. parallelism across the method (inherently parallel methods),
3. parallelism across the steps (parallel computation of several time steps),
4. parallelism across the system (parallelism in the evaluation of the right hand side of (3)).

These approaches are not mutually exclusive, e.g., parallel linear algebra can be combined with a parallelism across the system and a parallelism across the method approach. Such a setup could benefit from several sources of parallelism, which could be implemented on different architectures, e.g., parallel linear algebra on a shared memory basis using OpenMP [21] combined with an inherently parallel method based on distributed memory with MPI [26]. In addition to the methods given in [1] we also consider parallel peer methods [8].

4.1 Parallel linear algebra

Parallel linear algebra is probably the easiest way to parallelize a given application. Modern implementations of the LAPACK and BLAS routines, like OpenBLAS [9], automatically scan for possible parallelism and choose their number of threads accordingly. On modern processors, parallelism shows advantages for linear systems involving dense matrices with dimension exceeding 1000×1000 [10], [11]. Such matrices usually do not occur in the code generated by OpenModelica. For instance, one would need to simulate a 1000-pendulum to obtain problems of such size, but such large problems do not even compile on present day hardware, at least when using the OpenModelica compiler. Therefore, parallel linear algebra is not a feasible choice in our setting. Furthermore it might be more feasible to employ a sparse linear solver, if the problem permits. An example for a suitable method is our spring-mass-damper example, where a speed up of 2.9 can be achieved when using a sparse solver (see Table 2).

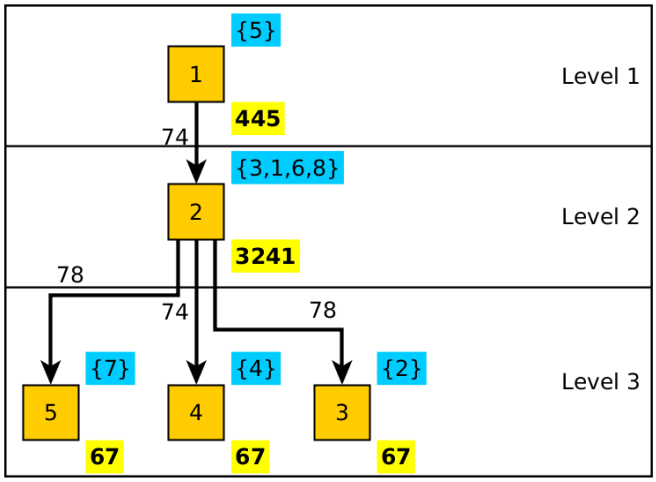

Fig. 3 The task graph of the model shown in Fig. 1.

4.2 Parallelism across the method

Several methods with inherent parallelism exist. Parallel Runge-Kutta methods were investigated in depth in [12]. In general, this class of methods does not offer much potential for parallelism, since no parallel Runge-Kutta method of order higher than four exists. More promising are the parallel iterated Runge-Kutta methods (PIRK) and their diagonally implicit variants, which implement the Picard iteration based on Gaussian quadrature rules. The advantage of these methods is that they can have arbitrary order, depending only on the underlying quadrature rule. Another parallelization approach is the use of general linear methods (GLMs). Here, we consider a peer method. In addition, we consider a way to add parallelism to any method, which relies on finite differencing to determine the Jacobian.

4.2.1 General linear methods: A peer method

As one type of GLM we considered the multi-implicit parallel two-step W method given in [2]. We used a fixed step implementation with five stages having order five (and order four for varying step sizes). Table 1 shows that our simple implementation of the parallel peer method is not much slower than the sophisticated codes CVode and DASSL. Implementing a variable step size strategy based on local error control with an embedded method of order two could actually lead to a competitive method. We did not investigate this further due to difficulties of implementing the method in OpenModelica.

4.2.2 Parallel evaluation of Jacobi matrices

Parallel evaluation of the Jacobian is a rather straight forward approach whenever a method uses finite differencing for evaluating the Jacobian. The advantage of this approach is that speed ups are obtained whenever the evaluation time of the right hand side of (3) has a significant influence on the overall computation time. In order to use the parallel evaluation we implemented a C++ version of the DASKR method [13] and used parallelization based on OpenMP. Simulation results show a speed up of 1.1 - 1.2 on two processors (see Tables 1, 2).

Table 1 Computation times for the N-pendulum example with different solvers, $N = 50$, $t_e = 1$ and absolute and relative error tolerance 10^{-8}.

Method	Computation Time	Speed Up
CVode	14.05 s	-
CppDASSL	11.80 s	1.20
CppDASSL (parallel Jac.)	10.70 s	1.31
Peer	18.95 s	0.74

Table 2 Computation times for the spring-mass-damper network example with N = 1000 elements, different solvers, $t_e = 100$ and absolute and relative error tolerance 10^{-6}.

Method	Computation Time	Speed Up
CVode	12.32 s	-
CppDASSL (sparse)	4.24 s	2.91
CppDASSL (sparse, parallel Jac.)	3.56 s	3.46

4.3 Parallelism across the steps

Parallelism across the steps is relatively new area of research, with the first algorithms being proposed in the 1960s. During the course of our investigations we considered several methods of this class: ParaReal [15], P(D)IRKAS [1] and PFASST [16]. Our experiences were twofold. While we obtained speed ups in the N-pendulum example for the ParaReal method, this required 32 processors to get a speed up of about 1.7. In addition, this solution required a manual and tedious adaption of the method parameters, which violates our second requirement. With PFASST we had a similar experience, with the sole difference that we could not even produce speed ups. In theory, speed ups should be possible for coarse grained systems, but not for general purpose models. With P(D)IRKAS we actually obtained speed ups on a reasonbale number of processors, but only for very small tolerances ($10^{-12} - 10^{-10}$), which are, in most cases, not of interest in engineering.

4.4 Parallelism across the system

Here, we present a method, which automatically parallelizes the evaluation of the right hand side of arbitrary problems. The method is based on the task graph, which is introduced in the following.

4.4.1 Task graph of a model

A task graph is a well known representation for parallelization problems [17], [18]. Such a graph G is directed, contains a set of tasks T and a set of edges E as well as information about the calculation and communication time.

$$G := (T, E, c, \tau)$$

$$E \subset T \times T$$

$$c : E \to R$$

$$\tau : T \to R$$

A task is an arbitrary calculation problem, which has dependencies to other tasks. These dependencies are descriped by edges $e \in E = (t_1, t_2)$ with the constraint $t_1, t_2 \in T$.

Table 3 Measured speed ups for the BranchingDynamicPipes example from the Modelica Standard library using different scheduling algorithms and eight cores.

Method	Speed Up
Modified Critical Path (MCP)	3.48
List	3.34
Metis	3.67
Level	4.45
Intel TBB	2.95

The meaning of such an edge is that the task t_1 has to be evaluated completely before the task t_2 can be calculated. Additionally, the graph holds information about the time that is required to calculate a task and to communicate the results to another task if there is an edge between them. These calculation costs are marked with τ in the given representation and the communication costs are descriped by c. The equations of an arbitrary simulation model can be transformed into such a graph, by using the SCC representation. First of all, a task is generated for each SCC of the model. If a variable calculated in SCC S_1 is required by SCC S_2 for calculation, an edge is added from the task of S_1 to the task of S_2. If a task has no incoming edges, it is called a root task or root node. If a task has no outgoing edges, it i s called a leaf task or leaf node. The level of a task t is defined as the number of tasks that are along the longest path from t to a root task.

As a real world example, the electrical model shown in Fig. 1 is analyzed further. The corresponding task graph, which is automatically generated by OpenModelica, is shown in Fig. 3, the text inside each node represents its unique identifier, which is just an ongoing number. The solved equations are displayed with a blue background in the right uper corner. The yellow shaded numbers, displayed on the right bottom edge of each task, represent the required execution time. The numbers

Table 4 Obtained speed ups for different models from the Modelica Standard Library using eight cores.

Model	Speed Up
BranchingDynamicPipes (fluid)	4.45
CauerLowPass (electric)	1.9
N-pendulum	1.08

along the edges represent the communication time that is required to transmit the results of a task from one processing unit to another. If two tasks are handled by the same processing unit (e.g. CPU-core), the communication time is zero. The graph shows that equation 5 has to be solved first. After the calculation of equation 5, the equation system consisting of equations 3, 1, 6, and 8 can be calculated. Finally, the equations 7, 4, and 2 can be solved in parallel

Since the task graphs can become quite large, even for simple models, the graph should by simplified as much as possible. Therefore different graph rewriting rules have been implemented. For the given example, the first two tasks would be merged into one, because it makes no sense to calculate them by different processing units.

4.4.2 Scheduling of a task graph

After the task graph creation, a scheduling has to be performed. Scheduling means that a mapping from the tasks of the task graph to the available processing-units has to be found. Finding the best solution is a known NP-hard problem [19], so algorithms use heuristics to create a good solution. Different scheduling algorithms were implemented and verified in the context of OpenModelica. They are divided into static and dynamic algorithms. Static algorithms create a fixed mapping during compile time, dynamic algorithms perform a load balancing at runtime, to achive a better workload.

The implemented dynamic schedulers were the flow graph

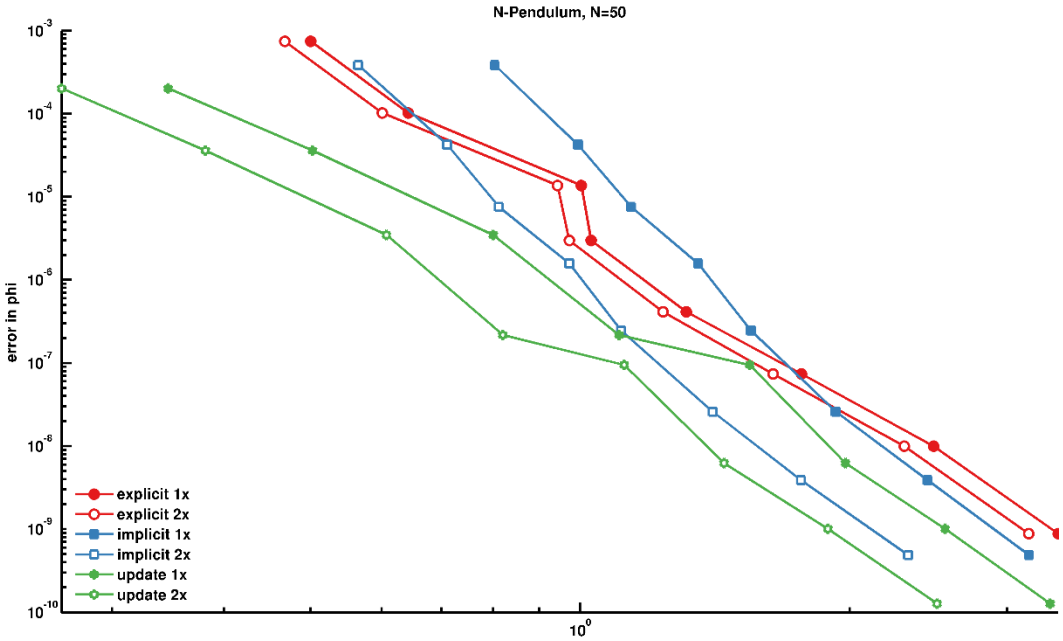

Fig. 4 Error over computation time for different formulations of the N-Pendulum (N = 50) problem combined with a parallelization across the system approach.

framework of Intel TBB [20] and the task dependency constructs of OpenMP 4.0 [21]. Both are simple, because they just take the graph $G(T, E)$ and perform a scheduling and work-balancing completely automatic. Additionally, a simple semi-dynamic scheduling algorithm was implemented. It is called level-scheduler and it calculates all nodes of the same level in parallel until it starts with the next level. The nodes of the same level can be dynamically scheduled with the help of OpenMP-sections.

For the evaluation of static scheduling, the following algorithms were taken into account: list scheduler (breadth first) [22], breadth first scheduler based on metis partitions [23] and the modified critical path scheduler [24]. A major drawback of these static algorithms is that the values of c and τ have to be accurate and constant over time. Otherwise the scheduling will not reach the intended performance. By profiling all tasks in a serial run, good values can be created for simple equations without trigonometric parts and for linear equation systems. Especially the computation time of nonlinear equation systems varies significantly dependent on the structure of the Jacobian.

4.4.3 Results and further analysis

Table 3 shows the speed ups measured for the BranchingDynamicPipes example from the Modelica standard library using the scheduling algorithms from the previous subsection. It is clear to see that the semi-dynamic level scheduler offers the best speed up. Therefore, we chose this scheduling algorithm for our further experiments. Table 4 shows the speed up obtained for models from different domains within the Modelica library. The results clearly show that models from the fluid domain benefit especially well from parallelism across the system. This is due to the many line elements in the system, which can all be simulated in parallel. The electric domain also shows a decent speed up. Similar to the hydraulics example several parts of a larger network can be handled in parallel. Unfortunately, nearly no speed ups are obtained for the N-pendulum example. In the following we will further investigate this behavior.

Looking at the N-pendulum equations, one can see that the single entries $F_i(\varphi, \psi), i = 1, \dots, N$ can be evaluated in parallel, since they do not depend on each other. Nonetheless, nearly no speed up was obtained. This is due to the fact that in order to evaluate the $M(\varphi)^{-1}F(\varphi, \psi)$ part, the solution of a linear system is required. This linear system can only be solved sequentially and dominates the computation time of the right hand side.

Since we are using either CVode or DASSL as solver it is a straight forward idea to change the problem formulation (4) into a residual formulation of the form

$$\begin{pmatrix} I & 0 \\ 0 & M(\varphi) \end{pmatrix} \begin{pmatrix} \dot{\varphi} \\ \dot{\psi} \end{pmatrix} = \begin{pmatrix} \psi \\ F(\varphi, \psi) \end{pmatrix}.$$

Fig. 4 shows errors over computation time for implementations on one and two processors. Since, at the time of writing, OpenModelica does not support the residual formulation, the measurements were taken on a manual implementation. Comparing the results for the original implementation (termed explicit in the figure) and residual formulation (termed implicit) it is clear that especially for higher errors the residual formulation is actually slower on one processor than the explicit formulation. On the other hand, the residual formulation actually benefits from parallelization, resulting in a speed up of about 1.5 over the one processor implementation.

The fact that the residual formulation actually takes longer to solve than the explicit formulation was already observed in [25]. The reason for this is that the solver needs significantly more Jacobian evaluations than in the explicit case. The author of [25] proposed a change in the Jacobian update strategy to overcome this problem. Given the Jacobian of the residual formulation

$$J_{res} = \alpha \begin{pmatrix} I & 0 \\ 0 & M(\varphi) \end{pmatrix} - \left(\frac{\partial F(\varphi, \psi)}{\partial \varphi} + \frac{\partial M(\varphi)}{\partial \varphi} \dot{\psi} \quad \frac{\partial F(\varphi, \psi)}{\partial \psi} \right),$$

where α is a method parameter, which is proportional to $1/h$ with h the actual step size, the idea is to update only the first summand of J_{res} whenever the stepsize and, therefore, α changes. The effects of this change are twofold. First, the evaluation costs for the Jacobian are decreased, since no finite differencing is necessary. Second, because the Jacobian matrix is updated proactively, actually fewer convergence failures occur and, therefore, fewer Jacobian re-evaluations are required. The effect of this improved update strategy can be seen in Fig. 4 (results denoted update). The calculations are faster than in the explicit case even on one processor while the good parallel performance is pertained. Compared with the explicit formulation on one processor, a speed up of about 2.4 can be reached on two processors. We later found out that Arnold et al. [14] proposed the same solution.

5 Conclusions

In this work we examined several approaches for parallelization with the intend of using them in a general multibody system simulation code. From our experiments we can offer the following suggestions:

- If your code permits the residual formulation use it together with the presented parallelism across the system approach.
- Use sparse linear algebra, whenever your problem permits it.
- If you have further processors available, using the parallel evaluation of the Jacobian results in a further speed up of 1.1 - 1.2.
- A sophisticated implementation of the parallel peer method might also result in reasonable speed ups. Such a method could further be enhanced by parallelism across the system and the parallel evaluation of the Jacobian.

Acknowledgment

This work was funded by the German Federal Ministry of Education and Research (BMBF). A. Naumann is grateful for funding by the German Research Foundation (DFG) within CRC/TR 96.

REFERENCES

[1] K. Burrage, *Parallel and Sequential Methods for Ordinary Differential Equations*. New York, NY, USA: Clarendon Press, 1995.

[2] B. Schmitt, R. Weiner, and H. Podhaisky, "Multi-Implicit Peer Two-Step W-Methods for Parallel Time Integration," *BIT Numerical Mathematics*, vol. 45, no. 1, pp. 197-217, 2005.

[3] M. Gebremedhin, "Parmodelica: Extending the Algorithmic Subset of Modelica with Explicit Parallel Language Constructs for Multi-Core Simulation," Master thesis, University of Linköping, Sweden, 2011.

[4] P. B. Johns and M. O'Brien, "Use of the Transmission-Line Modelling (t.l.m.) Method to Solve Non-Linear Lumped Networks," *Radio and Electronic Engineer*, vol. 50, pp. 59-70, 1980.

[5] S. E. Mattsson and G. Söderlind, "Index Reduction in Differential-Algebraic Equations Using Dummy Derivatives," *SIAM Journal on Scientic Computing*, vol. 14, no. 3, pp. 677-692, 1993.

[6] R. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146-160, 1972.

[7] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward, "Sundials: Suite of Nonlinear and Differential Algebraic Equation Solvers," *ACM Trans. Math. Softw.*, vol. 31, pp. 363-396, 2005.

[8] B. A. Schmitt and R. Weiner, "Parallel Two-Step W-methods with Peer Variables," *SIAM Journal on Numerical Analysis*, vol. 42, no. 1, pp. 265-282, 2004.

[9] Q.Wang, X. Zhang, Y. Zhang, and Q. Yi, "Augem: Automatically Generate High Performance Dense Linear Algebra Kernels on x86 CPUs," in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13, (New York, NY, USA), pp. 1-12, ACM, 2013.

[10] D. Hackenberg, R. Schöne, W. Nagel, and S. Pflüger, "Optimizing OpenMP Parallelized dgemm Calls on SGI ALTIX 3700," in Euro-Par 2006 Parallel Processing (W. Nagel, W. Walter, and W. Lehner, eds.), vol. 4128 of Lecture Notes in Computer Science, pp. 145-154, Springer Berlin Heidelberg, 2006.

[11] E. Agullo, B. Hadri, H. Ltaief, and J. Dongarrra, "Comparative Study of One-Sided Factorizations with Multiple Software Packages on Multi-Core Hardware," in High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on, pp. 1-12, Nov 2009.

[12] K. R. Jackson and S. P. Nørsett, "The Potential for Parallelism in Runge-Kutta Methods. Part 1: RK Formulas in Standard Form," *SIAM Journal on Numerical Analysis*, vol. 32, no. 1, pp. 49-82, 1995.

[13] K. Brenan, S. Campbell, and L. Petzold, *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*. Society for Industrialand Applied Mathematics, 1995.

[14] M. Arnold, A. Fuchs, and C. Führer, "Efficient Corrector Iteration for DAE Time Integration in Multibody Dynamics," *Computer Methods in Applied Mechanics and Engineering*, vol. 195, no. 50-51, pp. 6958-6973, 2006.

[15] J.-L. Lions, Y. Maday, and G. Turinici, "Résolution d'EDP ar un Schéma en Temps < Pararéel >," *Comptes Rendus de l'Academie des Sciences. Serie I, Mathematique*, vol. 332, no. 7, pp. 661-668, 2001.

[16] M. L. Minion and S. A. Williams, "Parareal and Spectral Deferred Corrections," in American Institute of Physics Conference Series (T. E. Simos and C. Tsitouras, eds.), vol. 1048 of American Institute of Physics Conference Series, pp. 388-391, Sept. 2008.

[17] Y. Robert, "Task Graph Scheduling," in Encyclopedia of Parallel Computing (D. A. Padua, ed.), pp. 2013-2025, Springer, 2011.

[18] P. Aronsson, "Automatic Parallelization of Equation-Based Simulation Programs." Institutionen for datavetenskap, 2006.

[19] P. Chretienne, "Tree Scheduling with Communication Delays," *Discrete Applied Mathematics*, vol. 49, no. 1-3, pp. 129-141, 1994.

[20] Intel Corporation, "Threading building blocks (intel tbb)." https://www.threadingbuildingblocks.org/, Oct. 2015.

[21] O. A. R. Board, "Openmp (open multi-processing) standard." http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, 2015.

[22] J. Chase, "On the Near-optimality of List Scheduling Heuristics for Local and Global Instruction Scheduling." Master thesis, University of Waterloo, Canada, 2007.

[23] 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, 1998.

[24] A. Radulescu and A. J. C. van Gemund, "FLB: Fast Load Balancing for Distributed-Memory Machines.," in ICPP, pp. 534-541, IEEE Computer Society, 1999.

[25] P. Schulz, "Vergleich und Analyse von Zeitintegratoren und Indexreduktionstechniken für DAEs," Master thesis, TU Dresden, Germany, 2014.

[26] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, S. Huss-Lederman, *MPI: The Complete Reference*. MIT Press Cambridge, MA, USA, 1995.