Contributions to the Efficient and Parallel Jacobian Evaluation and its Application in OpenModelica

Willi Braun[1] Martin Schroschk[2] Vitalij Ruge[3] Andreas Heuermann[1] Bernhard Bachmann[1]

[1]University of Applied Sciences Bielefeld, Germany, w-braun@posteo.de,
{bernhard.bachmann,andreas.heuermann}@fh-bielefeld.de
[2]Center for Information Services and High Performance Computing, TU Dresden, Germany
martin.schroschk@tu-dresden.de
[3]Siemens AG, Energy Sector, Erlangen, Germany vitalij.ruge@siemens.com

Abstract

Many algorithms related to Modelica-based simulations heavily rely on the efficient provision of Jacobian matrices. Besides the accuracy of the derivative information, the performance of the derivative evaluation is also of great interest, since it can have a large share in the total simulation time. In this paper, we propose two complementary approaches basing on identification of constant parts and parallelization to accelerate Jacobian evaluation. Furthermore, the implementations of these techniques in the open-source Modelica tool OpenModelica are discussed. The gained speedup in Jacobian evaluation is demonstrated on benchmark models of the ScalableTestSuite.

Keywords: Jacobian Evaluation, Symbolic Differentiation, Derivatives Computation, Coloring, Sparsity, Parallelization, Modelica, OpenModelica

1 Introduction

Solving computational problems numerically often rely on the access of derivative information, e.g. Jacobian matrices. In the Modelica context this includes algorithms used for solving algebraic loops, implicit integration methods as well as optimization algorithms. Both the speed of the derivatives generation and their accuracy can have a significant impact on such algorithms w.r.t. runtime and robustness. From a mathematical point of view there are several techniques to provide derivatives. A very common numerical method are finite differences, which approximate derivatives by difference quotients. For implicit integration methods with stepsize control, like DASSL or Sundials/IDA, the accuracy of the Jacobian is subordinated, since it is dominated by the current stepsize. The numerical approach using finite differences is very reasonable for this class of algorithms and therefore default in OpenModelica. So accuracy is not an argument for the contrary symbolic differentiation (cf. (Braun, Gallardo Yances, Link, and Bachmann 2012)) in case of implicit integration. But performance might be. The computation time of the Jacobian matrices can have a major share in the total simulation time of a model. To emphasize this statement we consider the model DistributionSystemLinear from the ScalableTestSuite (Casella 2015): In Table 1 the execution time for the simulation and Jacobian evaluation using the symbolical derivative module of OpenModelica, as well as the total number of Jacobian evaluations over the simulation time is stated for three different model sizes N. The timings were obtained on an Intel Xeon E5-2680 v3 processor using OpenModelica in version 1.12. Since these models contain one large linear system, which makes the compression futile, the Jacobian evaluation takes a significant amount of the total simulation runtime. Therefore, the right hand side for the Jacobian has to be evaluated significantly more often. Accelerating the evaluation of Jacobians would lead to a sizeable reduction of the total simulation runtime. Furthermore, the Jacobian evaluation does not make use of the available parallel hardware resources today's multi-core processors, like the used Intel Haswell processor, offer. This example serves as a representative for models having evaluation expensive right hand sides which are dominating the large algebraic loop.

Table 1. Timings for overall simulation t_{total} and Jacobian evaluation t_{jac} for sequential execution and the total number of Jacobian evaluations over a simulation run for three different model sizes N.

N	Dim. of J	Evals. of J	t_{total} [s]	t_{jac} [s]
14	196×196	15	3.8	2.7
20	400×400	14	12.7	10.3
28	784×784	15	47.3	41.5

In previous work, it was shown how directional derivatives are computed in OpenModelica within the symbolic derivative module (Braun, Ochel, and Bachmann 2011). These techniques were combined with coloring approaches in (Braun, Gallardo Yances, Link, and Bachmann 2012) and become extensively used by other algorithms inside of the OpenModelica Compiler as for algebraic loops, optimization and FMI (cf. (Braun and Bachmann 2014), (Ruge and Bachmann 2014), (Åkesson, Braun, Lindholm, and Bachmann 2012), (Franke, Walther, Worschech, Braun, and Bachmann

2015)). Algorithms for optimization and solving algebraic loops strongly rely on fast calculation of accurate derivatives. Therefore, the focus of this paper is on efficient evaluation of directional derivatives necessary for evaluating symbolical Jacobians. We present two complementary techniques to improve the performance of Jacobian evaluation and apply them to the OpenModelica Compiler. The first technique identifies recurring calculations with constant parts in the directional derivatives. These constant parts need to be evaluated only once and the results will be reused in the construction of the Jacobian matrix. The second approach introduces a parallelization approach for the Jacobian evaluation and sketches the implementation in the C simulation runtime of OpenModelica.

The paper is organized as follows: First, the context, generation and evaluation of Jacobian matrices in Modelica compilers is described in Section 2. Based on that two complementary techniques to accelerate the Jacobian evaluation are presented in Section 3. In Section 4 the proposed algorithms and improvements are evaluated using Modelica models. Finally, the presented work is summarized and an outlook for further improvements is given.

2 Jacobians in Modelica Models

A Modelica model is typically translated to a basic mathematical representation of differential and algebraic equations (DAEs) and transformed to ordinary differential equations (ODEs) before being able to simulate the model. The result of the so-called flattening process is the equation system

$$\begin{aligned} F(x(t),\dot{x}(t),y(t),u(t),p,t) = 0, \quad t \in [t_0,t_f] \\ x(t_0) = x_0, \end{aligned} \tag{1}$$

where $x(t) \in \mathbb{R}^{n_x}$ are the potential states, $\dot{x}(t) \in \mathbb{R}^{n_x}$ are the potential state derivatives, $y(t) \in \mathbb{R}^{n_y}$ are the algebraic variables and $u(t) \in \mathbb{R}^{n_u}$ are the inputs. The simulation time is given by $t \in [t_0,t_f]$. For simplicity, the initial conditions of the DAE states at start time t_0 are given by x_0. Introducing $z = (\dot{x}\ y)$, denoting the unknown variables, and $v = (x\ u\ p)$, denoting the known variables, the DAE can be re-written as

$$F(z,v,t) = 0. \tag{2}$$

Next, the causalization process to get an ordering of the unknown variables $z(t)$ is applied, which enables to solve them sequentially

$$z = G(v,t) \in \mathbb{R}^{n_x+n_y}. \tag{3}$$

The general form of the causalized system consists of a sequence of k assignment statements including implicit systems of equations. These assignments and so-called algebraic loops can be stated as

$$0 = g_i(z_i, z_1, z_2, \ldots, z_{i-1}, v, t) \in \mathbb{R}^{n_i}, \quad i = 1,\ldots,k, \tag{4}$$

with

$$(z_1,\ldots,z_k) := z, \quad z_i \in \mathbb{R}^{n_i} \quad \text{and} \quad \sum_{i=1}^{k} n_i = n_x + n_y.$$

Each function g_i assigns values to z_i by utilizing previously computed values for $z_1,\ldots,z_{i-1}$.

The Jacobian of a function of vector-valued function $f : \mathbb{R}^n \to \mathbb{R}^m, x \mapsto f(x)$ is defined as

$$\frac{\partial f}{\partial x} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix}. \tag{5}$$

An important tool when computing Jacobians are directional derivatives. The directional derivative of a vector valued function $f(x)$ is defined by

$$df = \frac{\partial f}{\partial x} \cdot dx, \tag{6}$$

where $dx \in \mathbb{R}^n$ represents the direction in which the directional derivative $df \in \mathbb{R}^m$ is evaluated. The vector dx is also referred to as a *seed vector*. In the following, directional derivatives will be used extensively to construct Jacobians. A straight forward, although naive, approach to construct a Jacobian from directional derivative evaluations is as follows: Using the identity matrix $I \in \mathbb{R}^{n\times n}$, and the unit vectors $e_1,\ldots,e_n \in \mathbb{R}^n$ it holds

$$\begin{aligned} \frac{\partial f}{\partial x} &= \frac{\partial f}{\partial x} \cdot I \\ &= \frac{\partial f}{\partial x} \cdot \begin{pmatrix} e_1 & \ldots & e_n \end{pmatrix} \\ &= \begin{pmatrix} \frac{\partial f}{\partial x} \cdot e_1 & \ldots & \frac{\partial f}{\partial x} \cdot e_n \end{pmatrix}. \end{aligned} \tag{7}$$

Thus, a Jacobian with n columns may be constructed from n evaluations of directional derivatives along the n unit vectors. Applying the concept of directional derivatives on the DAE (2) yields the relation

$$\frac{\partial F}{\partial z} dz + \frac{\partial F}{\partial v} dv = 0, \tag{8}$$

where dv is the input seed vector and dz works as the directional derivative of the relation (3) with respect to the direction dv. By solving the system of equations (8) for a particular seed vector dv, the directional derivative of the DAE is obtained. Technically this system of equations is represented in OpenModelica as a symbolic equation system, like the original equation system (3). This system contains the desired partial derivatives dz as unknowns, the seed vector dv and all other variables from the original system are considered as known and needs to be transformed like the original system into an explicit form. This can be achieved by applying so-called block lower triangular algorithms resulting in

$$dz = -\left[\frac{\partial F}{\partial z}\right]^{-1} \cdot \frac{\partial F}{\partial v} \cdot dv =: H(z,v,t) \cdot dv. \tag{9}$$

 10.3384/ECP20169159 DOI

This form can be further passed to the code generation to enable the evaluation of the directional derivative at simulation time. It is important to note that the system of equations (9) is linear in the unknown variables dz and the application of linear solvers is sufficient.

3 Accelerate Jacobian Evaluation

In this section we present two different but not incompatible approaches to accelerate the Jacobian evaluation to motivate the use of the symbolic differentiation module of OpenModelica for simulation and optimization.

3.1 Reuse of Constant Parts

Analyzing system (9), it is obvious, that the calculation of $H(z,v,t)$ is independent from the seed vector. If no algebraic loop is involved, the calculation is symbolically realized sequentially as

$$A = H(z,v,t) \tag{10}$$
$$dz = A \cdot dv. \tag{11}$$

Therefore, the calculation of equation (10) can be evaluated once for each Jacobian matrix construction. The implementation within OpenModelica is realised on expression level and basing on the `wrapFunctionCall` module to identify time-consuming function calls as well as common subexpressions and separate them inside the code generation phase of the OpenModelica Compiler backend. In order to emphasize the approach Example 3.1.1 is outlined in the following.

Example 3.1.1. A simple Modelica model with some sort of expensive function foo is considered:

```
model reuseConstantParts
  Real x_1(start=1,fixed=true);
  Real x_2(start=0,fixed=true);
  Real y_1, y_2;
equation
  y_1 = sin(x_1)*foo(x_2);
  y_2 = sin(x_1)*cos(y_1);
  der(x_1) = y_1*y_2;
  der(x_2) = y_1 + y_2;
end reuseConstantParts;
```

Let bar be the derivative of foo and dv the seed vector with respect to $v := (x_1, x_2)$. Then, differentiation will lead to

$$\frac{\partial y_1}{\partial v_i} = \cos(x_1)\cdot dv_1 \cdot foo(x_2) + \sin(x_1)\cdot bar(x_2)\cdot dv_2$$
$$\frac{\partial y_2}{\partial v_i} = \cos(x_1)\cdot dv_1 \cdot \cos(y_1) - \sin(x_1)\cdot \sin(y_1)\cdot \frac{\partial y_1}{\partial v_i}$$
$$\frac{\partial der(x_1)}{\partial v_i} = \frac{\partial y_1}{\partial v_i}\cdot y_2 + \frac{\partial y_2}{\partial v_i}\cdot y_1$$
$$\frac{\partial der(x_2)}{\partial v_i} = \frac{\partial y_1}{\partial v_i} + \frac{\partial y_2}{\partial v_i}.$$

OpenModelica will generate the additional common subexpressions `cse4=cos(x_1)`, `cse5=bar(x_2)`, and `cse6=sin(y_1)` independent of the seed vectors to be reused. Those are needed for the Jacobian evaluation but not the ODE-evaluation and only computed once for each integrator step and reused for each seed vector. The subexpressions `cse1` and `cse2` are determined during the ODE-evaluation, and will be reused for the Jacobian evalution. Finally, OpenModelica will generate C source code for the Jacobian evaluation similar to:

```
/* Constant equations */
cse4 = cos(x_1);
cse5 = bar(x_2);
cse6 = sin(y_1);
/* Dynamic equations */
y1.pDER = cse4*dv_1*cse2 + cse1*cse5*dv_2;
y2.pDER = cse4*dv_1*cse3 - cse1*cse6*y1.pDER;
der(x1).pDER = y1.pDER*y2 + y1*y2.pDER;
der(x2).pDER = y1.pDER + y2.pDER;
```

Furthermore, if the equation system (3) contains (non-) linear algebraic loops, an additional optimization of the Jacobian matrix generation can be achieved as follows: Implicit equation systems of the form

$$g_i(z_i, z_1, \ldots, z_{i-1}, v, t) = 0 \tag{12}$$

are differentiated straightforwardly equation by equation in order to compute the directional derivative dz_i. This yields into

$$\frac{\partial g_i}{\partial z_i}dz_i + \sum_{k=1}^{i-1}\frac{\partial g_i}{\partial z_k}dz_k + \frac{\partial g_i}{\partial v}dv = 0 \tag{13}$$

and can be solved as a linear system

$$dz_i = -\left[\frac{\partial g_i}{\partial z_i}\right]^{-1}\left(\frac{\partial g_i}{\partial v}dv - \sum_{k=1}^{i-1}\frac{\partial g_i}{\partial z_k}dz_k\right), \tag{14}$$

for dz_i. A LU factorization of the matrix

$$\frac{\partial g_i}{\partial z_i} = L\cdot U$$

is performed allowing to obtain the solution directly by forward and backward substitution. At this, the determination of the lower triangular matrix U and the upper triangular matrix L is the most computational expensive step. Since the matrix is independent of the seed variables dv, the LU factorization is constant for every Jacobian matrix construction. Up to now, the LU factorization is performed for each Jacobian matrix evaluation in OpenModelica. We implemented the reuse of the LU factorization so that the LU factorization is computed only for the first evaluation of equation (11) and than reused for the subsequent evaluations. The performance benefits of this improvement is quite huge for the OpenModelica implementation as depicted in Section 4.

3.2 Parallelization of Jacobian Evaluation

From a computer architecture view, the overall performance of a system is driven nowadays by increasing core count due to the fact that further scaling of single core frequency has reached physical limits w.r.t. overheating and power consumption (Kirk and Wen-Mei 2016). Thus, algorithms and software need to be parallelized to benefit from today's multi-core chips (Sutter and Larus 2005), (Olukotun and Hammond 2005). In this section, we deduce a parallelization using multithreading for evaluation of Jacobians basing on representation (7).

For simplicity and without limiting the generality the sparsity and coloring is omitted in the following. The algorithms and results presented in this paper are directly applicable for colored Jacobians with compressed columns.

The basic idea is to enable concurrent evaluations of the independent columns (i.e. the directional derivatives) in the Jacobian matrix. In Listing 1 the main Jacobian construction loop of OpenModelica is depicted: The Jacobian columns are elemtwise evaluated within two nested `for` loops by calls to the function `directional_der_JacA`. The later corresponds to the directional derivative function in equation (9) and the main work here is to achieve a thread-safe version of it. This means that all writable variables and structures, as linear algebraic loops, need thread local data to enable simultaneous execution.

Listing 1. Main Jacobian construction

```
for(int i = 0; i < jac->columns; i++) {
  jac->seedVars[i] = 1.0;
  callbacks->directional_der_JacA(data, jac);

  for(int j = 0; j < jac->rows; j++)
    matrixA[i][j] = jac->resultVars[j];

  jac->seedVars[i] = 0.0;
}
```

Since the Jacobian can contain linear algebraic systems it is necessary to deal with non-thread-safe solvers in parallel regions. In general each thread has to solve the same linear system for different seed vectors simultaneously. Figure 1 provides a schematic overview. We decided for our first implementation to provide a duplication of the structure of all linear algebraic systems to every thread. Each thread works (read and write) exclusively on its own copy and is thus thread-safe. This approach is not the most memory economical and will be changed in the final implementation in OpenModelica. To switch from a global to a thread local data structure is a major change in the OpenModelica Compiler backend and its C simulation runtime. The thread local Jacobian data structure is depicted in Listing 2.

Listing 2. Thread local data structure for Jacobians.

```
typedef struct LINEAR_SYSTEM_THREAD_DATA {
  void *solverData[2]; /* Private date for
      external linear solvers */
```

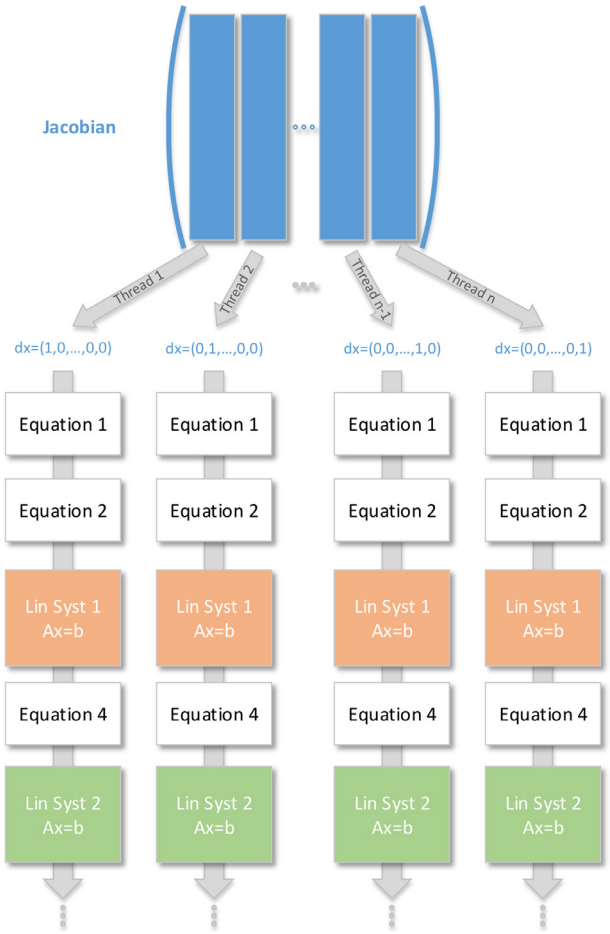

Figure 1. Evaluate Jacobian columns with linear systems in parallel

```
  modelica_real *x;          /* Solution x */
  modelica_real *A;          /* Matrix A */
  modelica_real *b;          /* Vector b */
  ANALYTIC_JACOBIAN *jacobian; /* Jacobian */
  ...
} LINEAR_SYSTEM_THREAD_DATA;
```

The OpenMP API (*OpenMP Application Programming Interface* 2018) for shared-memory parallelization is used to implement the presented parallel Jacobian evaluation in the C runtime of OpenModelica. We choose OpenMP over possible alternatives, like Pthreads and Intel TBB, for two significant reasons:

Availability: The OpenMP specifications are corporately developed by hardware, software and compiler manufactures since 1997 and it has become a de-facto standard for shared-memory parallelization of C/C++ and Fortran applications. Thus, it is widely supported by all major compiler collections as well as available and applicable on most systems.

Suitability: Since OpenMP heavily supports the fork-join model by providing work sharing constructs, like parallel `for` loops and parallel sections. This fits the need

 10.3384/ECP20169159 DOI

for alternating single- and multithreaded computing stages in our presented approach. Parallelization is introduced through so-called pragmas, which are ignored if the compiler does not support OpenMP or no OpenMP support is requested. This leads to the huge advantage, that existing sequential code can be parallelized, but still remains sequentially executable if OpenMP is not supported.

The developed parallelization approach makes use of the fork-join model in the sense that the columns of the Jacobian are computed in parallel while the remaining simulation is sequentially executed. The n columns of the Jacobian matrix are evaluated in the existing sequential implementation within n iterations of a `for` loop. For the parallel implementation, this `for` loop is preceded by a work sharing `#pragma omp for` clause, which distributes the several loop iterations to the available threads at runtime. The implementation is free of explicit barriers and critical sections, i.e., the spawned threads run fully concurrently in the parallel region. The maximal number of independent chunks is barely the total number of columns of the Jacobian. Consequently, the approach can be scaled up to a maximum of n threads for a fixed model. In practice, the speedup gained from using an increasing number of threads will saturate because of the ever increasing overhead due to thread management will predominate the computations. The actual mapping of the columns to the threads is performed by a scheduling algorithm. The OpenMP API offers several scheduling algorithms like static, guided and dynamic scheduling. Which scheduler fits best depends among other things on the number of columns, the system's architecture and the containing equations. The Section 4.3 holds studies on using different schedulers.

4 Benchmarks

In the following, we study the characteristics and possible accelerations of the proposed enhancements w.r.t. Jacobian evaluation within OpenModelica using models from the ScalableTestSuite. We choose the ScalableTestSuite over other Modelica libraries for benchmarking for the following reasons:

Scalability: All contained models are scalable by setting one or more integer parameters. This is essential for the analysis of the scaling abilities of the approaches and their implementations, since the size of the Jacobian scales with the model size.

Significance: There are several models with a quite time consuming Jacobian evaluation.

Open-source: The ScalableTestSuite is freely available under the BSD 3-Clause License at GitHub[1]. Thus, the models can be inspected, and the results can be retraced and reproduced, respectively.

[1] `https://github.com/casella/ScalableTestSuite`

4.1 Hardware and Software Stack

In order to provide transparent, comparable and reproducible results, we describe the hardware and the relevant software stack on which the benchmarks are evaluated. The software configuration is as follows: The GNU Compiler Collection (GCC, 7.0.1) was used as C and C++ compilers with the optimization flags `-O2 -march=native`. The open-source Lapack implementation OpenBlas in version 0.2.20 is linked for basic linear algebra routines. We implemented the proposed algorithms and the parallelization of the Jacobian evaluation within a branch basing on OpenModelica 1.13-dev. The OpenModelica compiler needs to be invoked with `--generateSymbolicJacobian` to generate symbolical Jacobians for a model and the simulation flag `-jacobian=symbolical` must be passed for execution.

All benchmarks are evaluated on the High Performance Computing system (HPC) Taurus at TU Dresden (*Online Documentation for HPC System Taurus* 2018). In particular, we choose an Intel Haswell node comprising of two Intel Xeon E5-2680 v3 CPUs with 12 cores each, and 64 GB RAM in total. In order to obtain reproducible benchmark results the Hyper-threading technology as well as the turbo mode is disabled and the benchmarks run with the CPU's base frequency of 2.5 GHz. The resources are exclusively allocated, i.e., no noise from other users has to be considered.

Unless otherwise stated, all benchmark configurations are repeated five times and the average time over the runs is determined and used for the analysis. Using the average is reasonable due to the fact that the minimal and maximal timing values for all benchmarks are very close to the corresponding mean value.

4.2 Reuse of Constant Parts

The thermal models HeatingSystem_N ($N = 20, 40, 80$) and electric models PowerSystemStepLoad_N_64_M ($M = 4, 8, 16$) from ScalableTestSuite are used to analyze the proposed reuse of constant parts.

The HeatingSystem model represents a district heating system with N heated units, supplied by a heat distribution system. We noticed that for all N between 14 and 15 % of the total number of equations for the Jacobian evaluation is recognized as constant equations regarding the seed vector and therefore computed only once per time step.

The PowerSystemStepLoad model assembles a power system with a linear topology, obtained by connecting N power generators with each M finite volumes in a linear network with equal transmission lines, and with a load connected to each generator. Similar to the model above OpenModelica with reuse of constant parts is able to recognize between 22 and 23 % of the Jacobian equations to be reusable for different seed vectors.

The constant equations in these models consists entirely of trigonometric functions, depending only on state variables. While they are not especially expensive, there

are a lot of them. For example the PowerSystemStepLoad_N_64_M models contain each 4032 constant equations and up to 14210 equations dependent of the seed vector, while the Jacobian is of dimension up to 1537×1537.

The Tables 2 and 3 present the achieved timings for the HeatingSystem and PowerSystemStepLoad models using colored symbolical Jacobians. The columns with heading *No RcP* refer to the previous implementation without any reusage of constant parts in Jacobian evaluation. In contrast, the timings provided by the improved implementation, which considers constant parts, are stated within the columns *RcP*. For completeness and comparison reasons, the timings obtained from the default numerical technique for Jacobian evaluation are also given. The best timings for the Jacobian evaluation are highlighted in bold face.

As can be seen from Table 2, the reusage of constant parts significantly accelerates the Jacobian evaluation for symbolical Jacobians by a factor of 1.8 up to 2.0.

For this model, it would be advantageous to use dense symbolical Jacobians, since the number of colors found by the used heuristic is equal to the dimension of the Jacobian. Overall, the best timings are obtained for all model sizes using symbolic Jacobians in conjunction with reusage of constant parts. Furthermore, the symbolical Jacobian evaluation is considerably faster than the default numerical method. For example, the time spent to evaluate the Jacobian matrices drops from 316.61 s to 62.96 s for model size $N = 80$.

The benchmark results for the model PowerSystemStepLoad_N_64 are depicted in Table 3. For this model, the gained speed-up from the reusage of constant parts while Jacobian evaluation is about 1.7. The presented approach speeds up the Jacobian evaluation to such an extent, that it now outperforms the numerical evaluation.

The effect of the reusage of constant parts technique depends on the ratio between function calls that can be gathered as common sub expressions and then extracted with `RemoveSimpleEquations` and the total number of equation for the Jacobian evaluation.

4.3 Parallel Jacobian Evaluation

The model DistributionSystemLinear_N_M ($N = 14, 20, 28$, $M = N$) is used to demonstrate the application of the parallel Jacobian evaluation. This model represents an AC current distribution system, where the amount of segments can be scaled by the two parameters N for the primary distribution line and M for each secondary distribution line. The considered model sizes and the corresponding attributes are depicted in Table 4. Since the columns of the Jacobian matrix can be evaluated completely independently from each other and the implementation is free of explicit synchronization constructs (e.g., barriers and critical clauses), we expect near linear scalability for the parallelization approach. Full linear scalability cannot be expected, because the columns evaluated by the various threads need to be written to the global Jacobian matrix which is in shared memory. And on the other hand, the OpenMP loop scheduling will introduce some parallelization overhead as well. Figure 2 shows the strong scaling of the parallel Jacobian evaluation for models of size $N = 14$, $N = 20$ and $N = 28$, whereas the speedups refer to execution time with one thread. The black doted curve refers to the ideal (i.e, linear) speedup. As can be seen, the achieved speedup curves of all three models are nearly identical and the scalability is very close to linear speedup. This is a very good result and shows that the Jacobian evaluation can be significantly accelerated by parallelization. We also like to stress that the parallel Jacobian evaluation scales up to almost the full system size for all model sizes. Because the synchronization on OS level using the full system with 24 threads dominates the small amount of calculations w.r.t. thread-local Jacobian evaluations, the scaling slumps for model sizes $N = 14$ and $N = 20$ at this point. For model size $N = 28$ the computational portions overlay the overhead due to OS synchronization.

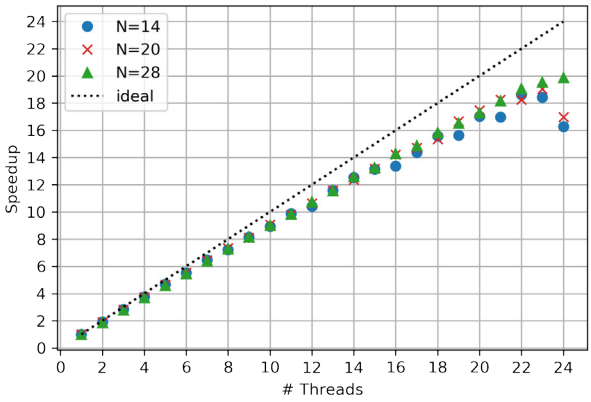

Figure 2. Strong scaling of parallel Jacobian evaluation for models with size $N = 14$, $N = 20$ and $N = 28$.

Next, we study the impacts of the four loop scheduling algorithms that the OpenMP API offers. Loop scheduling can have an serious impact on the scaling behavior and execution time of a parallel loop. This is especially true if not all iterations of a loop are equal in terms of computational effort. Such disequilibrium is called load imbalance. With respect to the considered parallelization approach for the Jacobian evaluation we do no expect load imbalances. But, eventually the good scaling behavior can be continued to full system size even for small models. The available scheduling algorithms are an in short:

Static: Without specification of the chunk size, the loop is divided into approximately equal chunks. They are assigned to the available threads.

Dynamic: The iterations are distributed to threads in the team in chunks. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed. The chunk size defaults to 1.

Table 2. Comparison of timings (in seconds) for total simulation and Jacobian evaluation for the HeatingSystem models achieved by using numerical and symbolical Jacobian evaluation without and with reuse of constant parts.

	Numerical			Symbolical colored				
				No RcP		*RcP*		
N	t_total	t_jac	#Jac Eval	t_total	t_jac	t_total	t_jac	#Jac Eval
20	11.70	6.20	54780	8.87	2.91	7.49	**1.51**	54647
40	67.82	43.64	102521	52.71	21.21	43.76	**11.59**	102548
80	451.65	316.61	189576	404.59	153.70	324.56	**75.98**	190012

Table 3. Comparison of timings (in seconds) for total simulation and Jacobian evaluation for the PowerSystemStepLoad_N_64_M models achieved by using numerical and symbolical Jacobian evaluation without and with reuse of constant parts.

	Numerical			Symbolical colored				
				No RcP		RcP		
M	t_total	t_jac	#Jac Eval	t_total	t_jac	t_total	t_jac	#Jac Eval
4	4.29	0.62	17	4.46	0.84	4.11	**0.49**	21
8	4.76	0.78	20	4.92	0.85	4.57	**0.50**	21
16	5.25	0.83	19	5.26	0.84	4.89	**0.49**	19

Table 4. Model size N and the corresponding values for number of variables, states and algebraic loop size and columns in Jacobian matrix.

N	Vars	States	Loops	Cols. in J	Evals. of J
14	12364	196	1621	196	15
20	25096	400	3277	400	14
28	49016	784	6381	784	15

Guided: The iterations are assigned to threads in the team in chunks. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be assigned. For a chunk size of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads in the team, decreasing to 1. It defaults to 1.

Auto: The decision regarding scheduling is delegated to the compiler and/or runtime system. The programmer gives the implementation the freedom to choose any possible mapping of iterations to threads in the team.

The OpenMP API provides the *runtime* clause, via which the schedule type can be specified at runtime. Thus, recompilation of the entire project is not necessary. In Figures 3 and 4 the speedups of the four different scheduling types for model size $N = 14$ and $N = 28$, respectively, are diplayed. The speedup is calculated with respect to the sequential execution time of the default scheduling type, which is dynamic for GNU libgomp (*Online Documentation for GNU libgomp: OMP_SCHEDULE* 2018). The speedups for the different schedulers are very tightly bunched together for 1 to 23 threads. There is a meaningful difference in performance using 24 threads for the smaller model. At this point, the computational parts become to small compared to the overhead introduced by the thread managment. The scalibility of the approach is not limited to 23 threads, but depends on the number of Jacobian columns. The larger model scales up to the full system using dynamic and guided scheduling. Overall, the default scheduler (i.e., dynamic with chunk size of 1) provides the best results. Hence, no adjustments from the users will be necessary.

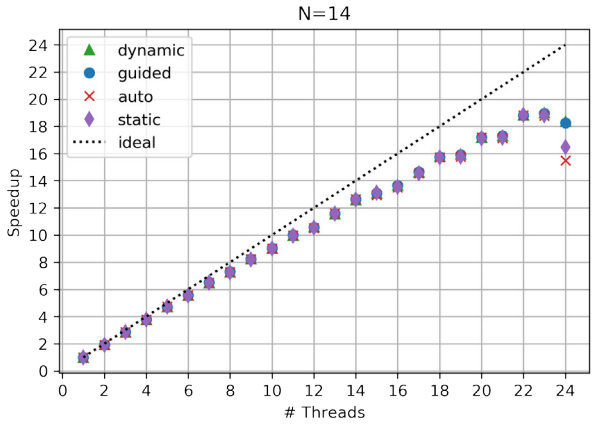

Figure 3. Speedup obtained for the scheduling types *static*, *dynamic*, *guided* and *auto* for model size $N = 14$.

5 Conclusion and Outlook

The presented work contributes to the efficient and parallel Jacobian evaluation using symbolic differentiation. For that, two complementary techniques are proposed to accelerate the computations:

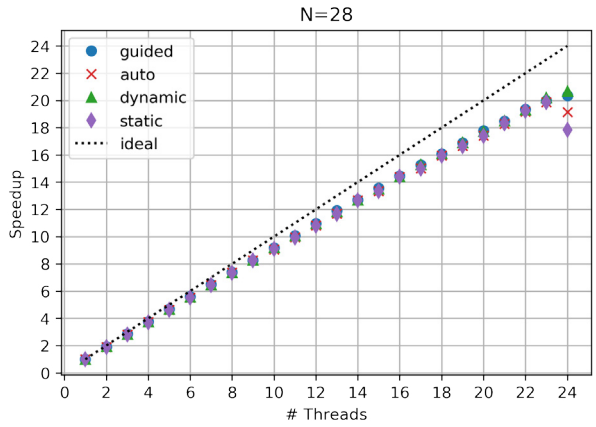

Figure 4. Speedup obtained for the scheduling types *static*, *dynamic*, *guided* and *auto* for model size $N = 28$.

Reuse of Constant Parts

While the speedup for many models from the ScalableTestSuite are enormous there are still examples for which OpenModelica currently is not able to find all constant parts. For example for the SteamPipe models no constant parts are found, despite big potential. The problem is, that `removeSimpleEquation` currently can't handle array equations and equations with function calls inside function calls. We plan to extend the described implementation in this way and expect a significant improvement for all examples inside ScalableTestSuite using Modelica.Fluid and Modelica.Media models.

Parallel Jacobian Evaluation

The parallelization approach allows for concurrent evaluation of the several columns of a Jacobian, which are independent. The OpenMP API is used for the implementation in OpenModelica. As discussed from a theoretical point of view and as shown by benchmarks, the parallel Jacobian evaluation provides near linear scalability. Simulations where Jacobian evaluations have a large share and optimization algorithms can greatly benefit from this. Although, only GCC was considered as compiler set, the scaling property of the approach and the implementation should be independent from the particular tool chain and will also hold for smaller shared-memory systems.

Furthermore the same approach can be adapted to the numerical and colored Jacobian evaluation in a similar way and should speed up the default simulation with OpenModelica significantly.

Sparse Rows Evaluation

As published in (Braun, Gallardo Yances, Link, and Bachmann 2012) the sparsity pattern of the Jacobian matrix can be calculated in OpenModelica and used to compress the matrix by combining columns with no shared non-zero elements in the same rows. This process is well-know as coloring and applied by replacing the identity matrix I in (7) with a compressed matrix, where all non-zero elements of the same color are combined into one compressed column as depict in Figure (5). Although the speed-up by

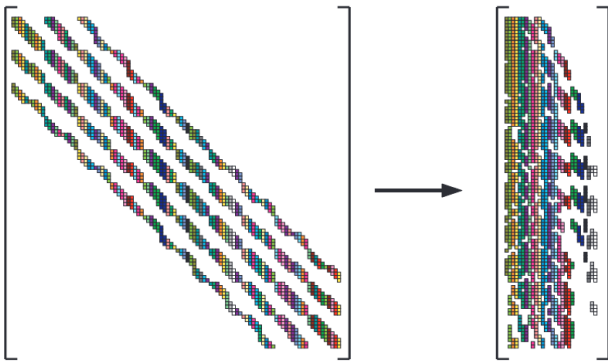

Figure 5. A matrix and its compressed representation from (Gebremedhin, Manne, and Pothen 2005).

compressing Jacobian matrices is enormous, still in every column the full directional derivative (9) is evaluated. This also includes all rows that are structurally zero.

To address this issue we propose to exploit the sparsity pattern further by attributing every equation whether it needs to be evaluated for the current column or not. To make this decision the sparsity pattern is used to mark the output rows of every column. In the next step an algorithm is applied to detect the minimal equation set that is needed to evaluate the marked output rows. The developed algorithm is based on Tarjan's algorithm as proposed in (Manzoni and Casella 2011). A necessary input for this algorithm is the directed graph, which is based on matching of the system (9). The output of the algorithm is mapped to every equation, so that at runtime every equation of (11) includes the dependency characteristic.

Since a large portion of the costly function calls can be reused as shown in section sec:ConstantParts it is not clear how big the remaining impact of this approach is. A first implementation of the described approach, but without the reuse of constant parts, leads to no clear results and needs further study.

Combination of Described Techniques

When writing this paper, the presented techniques are implemented on different branches of OpenModelica. Both branches base on a recent OpenModelica version and are very close to the mainline development branch and should be included into the master branch in the near future and be part of the next release.

Acknowledgment

The presented work is part of the PARADOM project funded by the Federal Ministry of Education and Research (BMBF) under support code 01IH15002.

References

Braun, Willi and Bernhard Bachmann (Feb. 2014). *Generic Differentiation Module and its Application within the OMC Backend*. Annual OpenModelica Workshop, Linköping Universitet, Sweden. Open Source Modelica Consortium. PDF.

Casella, Francesco (2015). "Simulation of Large-Scale Models in Modelica: State of the Art and Future Perspectives". In: *Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015*. 118. Linköping University Electronic Press, Linköpings universitet, pp. 459–468.

Franke, Rüdiger, Marcus Walther, Niklas Worschech, Willi Braun, and Bernhard Bachmann (2015). "Model-based control with FMI and a C++ runtime for Modelica". In: *Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015*. 118. Linköping University Electronic Press, Linköpings universitet, pp. 339–347.

Gebremedhin, Assefaw Hadish, Fredrik Manne, and Alex Pothen (Apr. 2005). "What Color Is Your Jacobian? Graph Coloring for Computing Derivatives". In: *SIAM Rev.* 47.4, pp. 629–705. ISSN: 0036-1445. DOI: 10.1137/S0036144504444711. URL: http://dx.doi.org/10.1137/S0036144504444711.

Kirk, David B and W Hwu Wen-Mei (2016). *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann.

Online Documentation for GNU libgomp: OMP_SCHEDULE (June 2018). Accessed 2018-06-03. GNU. URL: https://gcc.gnu.org/onlinedocs/libgomp/OMP_005fSCHEDULE.html#OMP_005fSCHEDULE.

Manzoni, Vincenzo and Francesco Casella (Mar. 2011). "Minimal Equation Sets for Output Computation in Object-Oriented Models". In: *Proceedings 8th International Modelica Conference*. Ed. by C. Clauss. Modelica Association. Dresden, Germany, pp. 784–790. ISBN: 978-91-7393-096-3. DOI: 10.3384/ecp11063784. URL: http://www.ep.liu.se/ecp/063/088/ecp11063088.pdf.

Braun, Willi, Lennart Ochel, and Bernhard Bachmann (Mar. 2011). "Symbolically Derived Jacobians Using Automatic Differentiation - Enhancement of the OpenModelica Compiler". In: *Proceedings of the 8th International Modelica Conference*. Ed. by Christoph Clauß. Dresden, Germany: Linköping University Electronic Press. DOI: 10.3384/ecp11063.

Braun, Willi, Stephanie Gallardo Yances, Killian Link, and Bernhard Bachmann (Sept. 2012). "Fast Simulation of Fluid Models with Colored Jacobians". In: *Proceedings of the 9th International Modelica Conference*. Ed. by Martin Otter and Dirk Zimmer. Munich, Germany: Linköping University Electronic Press. DOI: 10.3384/ecp12076.

Åkesson, Johan, Willi Braun, Petter Lindholm, and Bernhard Bachmann (Sept. 2012). "Generation of Sparse Jacobians for the Function Mock-Up Interface 2.0". In: *Proceedings of the 9th International Modelica Conference*. Ed. by Martin Otter and Dirk Zimmer. Munich, Germany: Linköping University Electronic Press. DOI: 10.3384/ecp12076.

Olukotun, Kunle and Lance Hammond (2005). "The future of microprocessors". In: *Queue* 3.7, pp. 26–29.

OpenMP Application Programming Interface (June 2018). Accessed 2018-06-01. OpenMP Architecture Review Board. URL: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

Ruge, Vitalij and Bernhard Bachmann (Feb. 2014). *Efficient Built-in Dynamic Optimization Features of OpenModelica*. Annual OpenModelica Workshop, Linköping Universitet, Sweden. Open Source Modelica Consortium. PDF.

Sutter, Herb and James Larus (Sept. 2005). "Software and the Concurrency Revolution". In: *Queue* 3.7, pp. 54–62. ISSN: 1542-7730. DOI: 10.1145/1095408.1095421. URL: http://doi.acm.org/10.1145/1095408.1095421.

Online Documentation for HPC System Taurus (June 2018). Accessed 2018-09-11. TU Dresden. URL: https://doc.zih.tu-dresden.de/hpc-wiki/bin/view/Compendium/HardwareTaurus.