Parallel Model Execution on Many Cores

Hilding Elmqvist Sven Erik Mattsson Hans Olsson
Dassault Systèmes
Ideon Science Park, SE-223 70 Lund, Sweden
Hilding.Elmqvist@3DS.com SvenErik.Mattsson@3DS.com Hans.Olsson@3DS.com

Abstract

Modelica gives the possibility to compose more and more detailed models since model components can be reused. This means that simulation needs to be faster. One possibility is then to use multi-core technology. Recent advances with more than 1000 cores show the potential.

The problem is then how to utilize this enormous processing power in a user friendly way. Partitioning needs to be made automatically. Modelica gives good possibility to automatically partition the model equation execution into separate threads since it is a declarative language based on equations.

This paper describes a method to automatically parallelize model equations implemented in Dymola. A speed-up of 3.4 times has been achieved using 4 cores/8 threads.

Keywords: Modelica; Multi-core; Automatic partitioning

1 Introduction

Modelica gives the possibility to compose more and more detailed models since model components can be reused. This means that simulation needs to be faster. One possibility is then to use multi-core technology. Recent advances with more than 1000 cores show the potential. One example is the Kalray MPPA (Multi-Purpose Processor Array) Technology (http://www.kalray.eu/). The MPPA 256 has 256 cores on one chip.

The problem is then how to utilize this enormous processing power in a user friendly way. Partitioning needs to be made automatically. Modelica gives good possibility to automatically partition the model equation execution into separate threads since it is a declarative language based on equations. (Aronsson, 2002, 2006) discussed automatic partitioning algorithms but speed-up was only achieved in special cases.

This paper describes technology to automatically parallelize model equations. Results using the implementation made in Dymola are included. Thermodynamic and electrical examples are discussed in detail. A speed-up of 3.4 times has been achieved using a laptop with Intel(R) Core i7-3740QM CPU @ 2.70GHz with 4 cores/8 threads.

2 Algorithms for Parallelization

The goal is to make a static scheduling of the equation execution. To find the optimal schedule is very complex, so various heuristics needs to be utilized. The first phase is to investigate which equations that could be executed in parallel. In this phase, the number of possible threads is not utilized.

In order to balance the efforts of executing the parallel sections, we need to have some estimates of the cost of executing different parts. Thus, the equations are analyzed and manipulated in the ordinary way until we have a sequence of calculating derivatives and variables at a given point of time. The calculations may include simple assignments, calls to Modelica functions or external functions and solving linear, nonlinear, mixed discrete/real systems of equations. An element in this sequence will be called a block node and it includes a set of solved equations and a set of variables that are solved by the node. The ordering of the block nodes is typically obtained by making a Block Lower Triangular (BLT) partition of the problem. The BLT partitioning defines a partial ordering meaning that a node can be executed if all block nodes with lower numbers have been executed. There may be other orderings that also may be feasible. If two successive blocks do not need each other's result, the order can be exchanged. They can in fact be executed in parallel. The major steps to investigate what can be executed in parallel are:

1. Build a dependency graph including all block nodes. Let the edges represent the dependencies between the nodes such that if node N_i needs the input of a variable v be-

longing to node N_j there is an edge between N_i and N_j. We can make the edges directed:
   a. For N_i it is a "need" edge meaning that it needs the results of N_j. The BLT partitioning has sorted N_j before N_i.
   b. For N_j it can be viewed as "used by" edge.
2. Parallelization: Search for all **source** blocks in the BLT graph, i.e. blocks which does not depend on other blocks (no outgoing needs edges) and collect them in layer L_i, which corresponds to equations or systems of equations that can be solved independently of each other.
3. Delete all nodes in L_i from the block node graph and delete all edges to them.
4. If there are still nodes in the graph, increase *i* by one and go to step 2.
5. The sets L define a parallelization and the calculations are given starting from *i*=1.

We developed this algorithm in October 2009 and made a test implementation. However, our run-time infrastructure was not completely reentrant, so we were not able to make any simulation tests. Casella proposed in 2013 (Casella, 2013) a very similar algorithm but did not present any run-time speed-up results. However, in step 2, this algorithm searched for **sinks**, i.e. it was a dual algorithm.

Unfortunately, using the result of these basic algorithms for parallelization doesn't give desired speed-up. The reason is that the obtained sections of a layer L_i are normally too small so overhead will give longer execution times. It is necessary to aggregate blocks into a section so the execution of a section outweighs the effort to set up and start the thread execution.

The approach to search for sources can be viewed as performing a calculation as soon it is possible. This approach can be taken bit further, if we consider the block nodes in the order defined by the BLT sorting and if we are collecting block nodes for L_i and a block node only has need edges from one section we add that block node to that section instead of deferring the calculation to L_{i+1}. The approach to search for sinks can on the other hand be viewed as performing the calculations just-in-time. We are then considering the nodes in the reverse order defined by the BLT sorting. If we are collecting block nodes for L_i and a block node only has a needed edges from one section we add that block node to that section instead of already doing the calculations in S_{i+1}. Please, recall that in this approach the sets are ordered in reverse calculation order.

In many cases the sets will include more parallel sections than we have cores. Thus, it is useful to merge sections within a set to make each section bigger to beat overhead. To get the sections balanced we need to have some estimate of the effort needed to perform the calculations of each block node. That is a difficult task for many reasons such as the solving of a non-linear problem may take different number of iterations. We have developed heuristic to get an estimate of the number of arithmetic operations needed to execute a block node. This includes also estimating the complexity of Modelica functions. Fortunately, we can make the result less sensitive to the estimates by generating code for more sections than we have cores. If we order the sections in estimated complexity order with the most complex first, the execution will start with them. If some estimates are bad or one nonlinear system needs more iteration, the other cores can be kept busy by handling remaining sections. Each section should have a minimum estimated effort.

From the two basic approaches discussed above indicates there may be a freedom in which parallel layer a calculation may be put in. To get fast translation it is not possible to check and evaluate all scheduling possibilities. To simplify the problem we focus on the most complex blocks and try to get as many of them in the same layers.

When the parallel sections of each layer have been identified, we need to be concerned with data. If data used and updated by different cores are adjacent, there is a risk of performance degradation due to *false sharing of cache lines*. Even if one core does not write the same double precision number as used by another core, the two variables might reside in the same cache line, typically 64 bytes of memory. We overcome this false sharing by adding padding, of typically 64 bytes, between the variables sets of the different sections.

3 Parallel Code for Equations

Consider the following model:

```
model ParallelCodeGeneration
  input Real u, U;
  Real x1, x2, x3;
  Real y1, y2, y3;
  Real X1, X2, X3;
  Real Y1, Y2, Y3;

equation
  der(x1) = f1(x1, u);
  y1 = g1(x1, u);

  der(x2) = f2(x2, y1);
  y2 = g2(x2, y1);
```


DOI 10.3384/ECP14096363

```
  der(x3) = f3(x3, y2);
  y3 = g3(x3, y2);

  der(X1) = F1(X1, U);
  Y1 = G1(X1);

  der(X2) = F2(X2, Y1);
  Y2 = G2(X2);

  der(X3) = F3(X3, Y2);
  Y3 = G3(X3);
end ParallelCodeGeneration;
```

It consists of 3 dynamical systems with direct term (g1, g2, g3) coupled in series and 3 systems in series without direct term (G1, G2, G3).

The partitioning of the equations into layers of parallel sections as described above corresponds to a fork-join code generation scheme. OpenMP supports this scheme by using simple pragmas in the C-code. The following code will thus be generated by the proposed parallelization algorithm:

```
#pragma omp parallel
{
 #pragma omp sections
 {
  #pragma omp section
  {
   der(x1) := f1(x1, u);
  }
  #pragma omp section
  {
   y1 := g1(x1, u);
  }
  #pragma omp section
  {
   der(X1) := F1(X1, U);
  }
  #pragma omp section
  {
   Y1 := G1(X1);
  }
  #pragma omp section
  {
   Y2 := G2(X2);
  }
 }
}
#pragma omp parallel
{
 #pragma omp sections
 {
  #pragma omp section
  {
   der(x2) := f2(x2, y1);
  }
  #pragma omp section
  {
   y2 := g2(x2, y1);
  }
  #pragma omp section
  {
   der(X2) := F2(X2, Y1);
  }
  #pragma omp section
  {
   der(X3) := F3(X3, Y2);
  }
 }
}
// Sequence of calculations
 der(x3) := f3(x3, y2);

// Result calculations
 y3 := g3(x3, y2);
 Y3 := G3(X3);
```

Two parallel layers will thus be used followed by a sequence (of one block). The last two equations are not needed for solving the differential equations only to enable plotting.

Note that *no* mutual exclusion locks are needed since updating a variable is done only in one block, i.e. in one section and reading is always done in later layers, i.e. after resynchronization.. Between layers of parallel code, usual sequential code is used typically when potential parallel sections would be too small and sequential execution is faster.

4 Example: Electrical Circuit

Consider a Spice3 model of a 4-bit ripple-carry adder. The model is available in the Modelica Standard Library:

Modelica.Electrical.Spice3.Examples.
Spice3BenchmarkFourBitBinaryAdder

The model adds two 4-bit numbers and is built out of four two-bit adders, Figure 1:

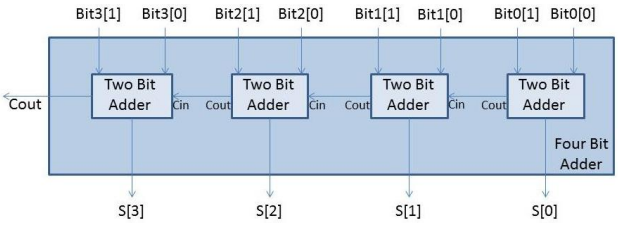

Figure 1: Four-bit adder

The two-bit adders are built out of two one-bit adders, Figure 2.

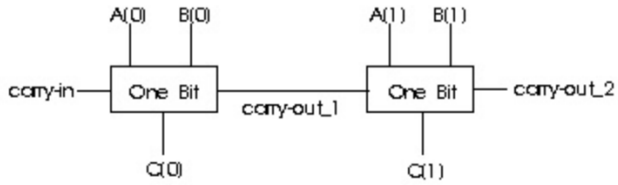

Figure 2: Two-bit adder

The one-bit adder is built out of nine NAND gates, Figure 3.

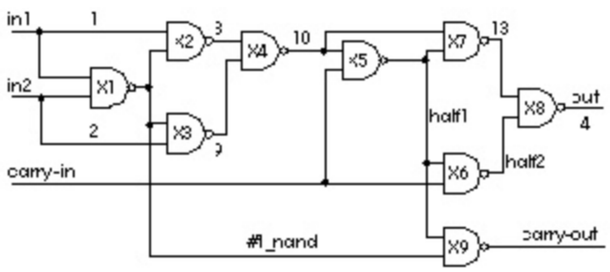

Figure 3: One bit adder

Each NAND gate has 5 BJT transistors and 3 diodes. The total model contains 180 BJT transistors and 108 diodes. There are 4640 unknown variables of which 324 are state variables. Many systems of equations appear:

Sizes of linear systems of equations: {5, 11, 5, 5, 5, 5, 5, 5, 66, 5, 11, 5, 5, 5, 5, 5, 5, 66, 5, 11, 5, 5, 5, 5, 5, 5, 66, 5, 11, 5, 5, 5, 5, 5, 5, 62, 5, 16, 5, 16, 5, 16, 5, 16, 5, 16, 5, 16, 5, 16, 5, 16}
Sizes after manipulation of the linear systems: {0, 3, 0, 0, 0, 0, 0, 0, 26, 0, 3, 0, 0, 0, 0, 0, 0, 26, 0, 3, 0, 0, 0, 0, 0, 0, 26, 0, 3, 0, 0, 0, 0, 0, 0, 24, 0, 7, 0, 7, 0, 7, 0, 7, 0, 7, 0, 7, 0, 7, 0, 7}
Sizes of nonlinear systems of equations: {47, 47, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49, 49}
Sizes after manipulation of the nonlinear systems: {4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4}

Simulating this model for 1e-7 seconds takes 40.9 seconds on using a laptop with Intel(R) Core i7-3740QM CPU @ 2.70GHz with 4 cores/8 threads. This laptop has been used for all timings in this paper.

The BLT (Block Lower Triangular) structure is shown in Figure 4:

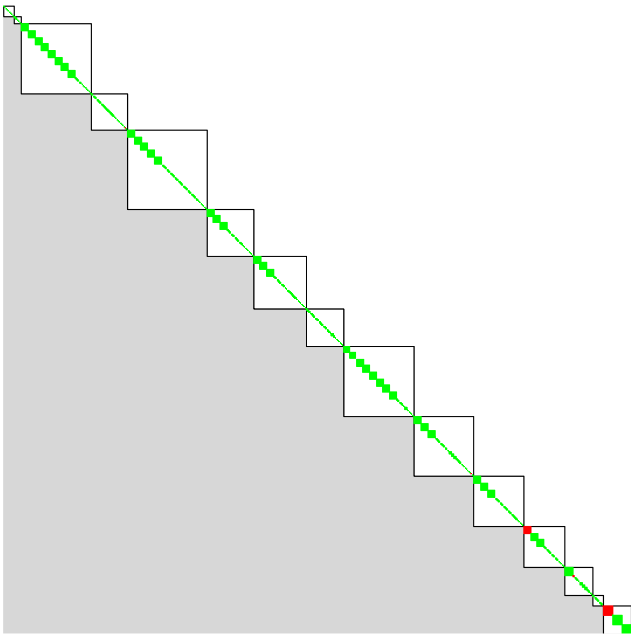

Figure 4: BLT structure for 4 bit adder

The BLT blocks are the small green squares along the diagonal. Note, that the BLT structure is not unique. The grey lower part has either zeros or ones. The rows and columns has been permuted according to the proposed partitioning algorithm (starting by searching *source* blocks) to show the triangles of zeros below the diagonal. These zeros are utilized when the layers of blocks to be executed in parallel are formed.

The layers can be compressed vertically as shown below to illustrate the parallel execution of blocks that don't need input from each other, Figure 5.

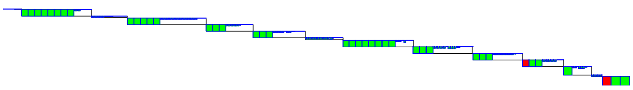

Figure 5: Initial parallel schedule

It can be noticed that many small systems of equations are present for which the cost of evaluation is very small. The cost of evaluation is estimated by counting the number of operations needed. A fixed number of iterations is assumed for nonlinear systems of equations and for for-loops and while-loops within functions. The corresponding diagram when the vertical size has been replaced by estimated number of operations (divided by 50) is shown in Figure 6.

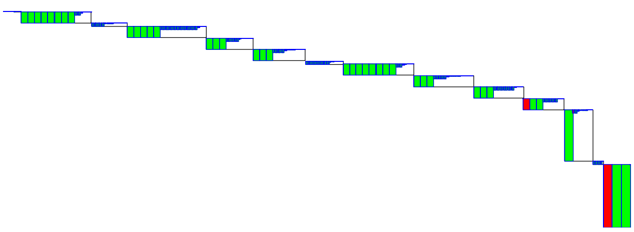

Figure 6: Parallel schedule (cost horizontally)

Blocks on the critical path are marked red. The estimated total number of operations are (sequentialPathLength) 424 198. The criticalPathLength = 28 967. The AmdahlSpeedUpFactor = sequentialPathLength / criticalPathLength = 14.7.

The pathLength for this particular schedule using numberOfLayers=15 and numberOfCores=126 is 80 225. The estimated speedUpFactor = 5.3.

The corresponding schedule produced by the dual algorithm (first searching for sinks) is shown in Figure 7.

DOI 10.3384/ECP14096363

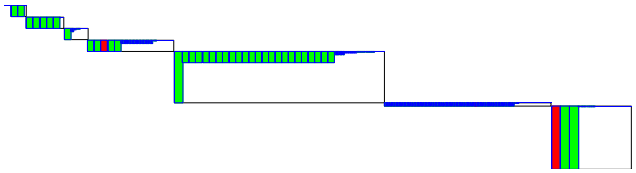

Figure 7: Parallel schedule using dual algorithm

The characteristics of this schedule are: pathLength = 60 697, speedUpFactor = 7.0, numberOfLayers = 15, numberOfCores = 325.

These schedules are not realistic for the target architectures with fewer cores available today. The granularity is too small since some of the sections just have a few operations. So merging of blocks must be done. Furthermore, it can be noted that some large blocks have ended up in different layers. In many cases, there is a freedom to move blocks between layers still not violating the data dependency. By appropriate heuristic rules, it is possible to achieve the schedule in Figure 8. In this case the maximum number of sections has been constrained to 4.

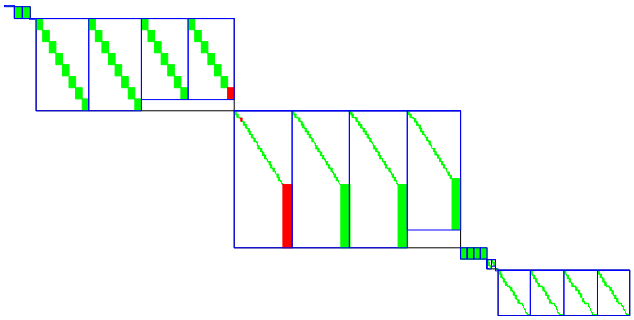

Figure 8: Parallel schedule with max 4 sections

The following characteristics is achieved: pathLength = 114 998, speedUpFactor = 3.7, numberOfLayers = 6, numberOfCores = 4.

The schedules presented are essentially Gantt charts with time advancing downwards. By using fixed width of the rectangles instead of width proportional to the number of unknown variables, we get Figure 9.

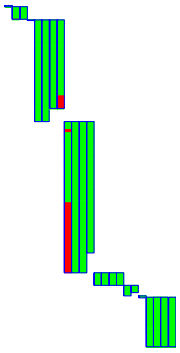

Figure 9: Gantt chart for 4 section schedule

The simulation time using this schedule is 16.5 seconds, i.e. a speedup of 40.9/16.5 = 2.48. For the dual algorithm (first searching for sources), the simulation time becomes 18.0 seconds.

The utilization of the cores is quite low as can be seen in Figure 10:

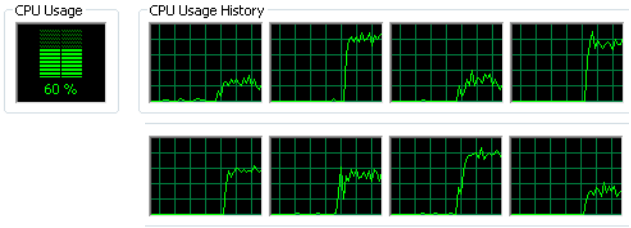

Figure 10: CPU usage with max 4 sections

It makes sense to increase the maximum number of sections to 8, 16 or even 32. The best simulation times achieved was 13.9 seconds, i.e. a **speed-up of 2.94**. The utilization of the cores then becomes 80%, Figure 11.

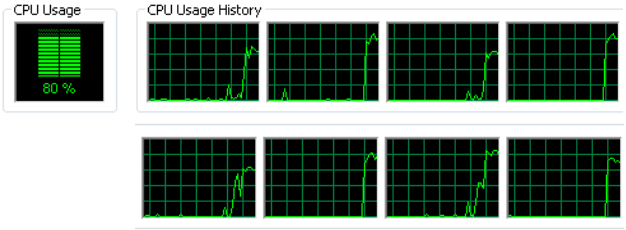

Figure 11: CPU usage with max 16 sections

The actual execution times are logged during simulation using high resolution timers and are presented as follows:

Name of block	Total CPU[s]	Min[us]
Seq 2(71)	0.009	0.22
Par 3[2](59)	1.571	32.85
Seq 4(45)	0.010	0.14
Par 5[16](98)	6.201	105.86
Par 6[4](428)	2.458	86.21
Par 7[4](49)	1.623	32.76
Par 8[2](36)	0.508	10.73
Seq 9(20)	0.010	0.26
Par 10[16](87)	0.988	28.51

Seq i(m) means sequential layer i with m BLT blocks, i.e. no parallelization. Par i[n](m) means parallel layer i with n parallel sections and m BLT blocks. It is then seen that layer 5 has larger execution time than layer 6 contrary to the estimated times shown in the Gantt diagram. The reason is that layer 5 has 16 sections but only 4 cores are available, i.e. some of the sections must be executed sequentially.

5 Example: Evaporator

Consider simulation of a model of an evaporator included in the AirConditioning library as AirConditioning.Examples.Evaporator model and shown in Figure 12.

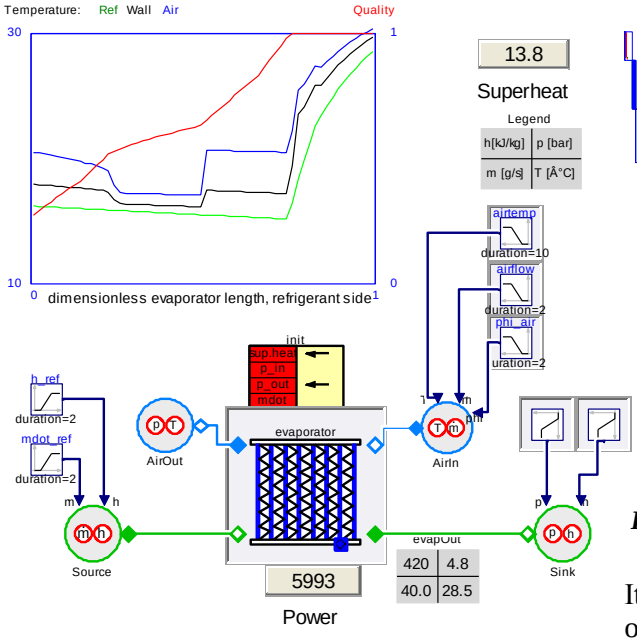

Figure 11: Evaporator model

The geometry of the evaporator for both air and refrigerant are discretized into segments. The example given in AirConditioning has evaporator.n_segAir = 1 and evaporator.n_segRef = 3, which gives that the refrigerant is discretized into 18 cells along the flow direction. Each of these cells as well the sink has pressure and enthalpy as continuous states and there is additionally 18 continuous time states for the wall temperature giving a total of 56 states. The major computational task is to calculate the thermodynamic properties from the state variables. There are 9 non-linear systems of size 23 and 9 non-linear systems of size 21. The difference is due to the geometry of the heat exchanger. The tearing procedure results in problems with 3 iterations variables for all of them. If we set the stop time to 100 s, it takes 2.44 s to simulate it on a normal laptop. We are here mainly interested in the speed-up when parallelize the calculation, so the absolute time is of less interest, however, it is of interest to consider the absolute time and number of calculations, because when the calculation are sufficiently short, the overhead to start and synchronize the parallel calculation the overhead will make the parallelized calculations take longer time than running then in sequence on one processor.

A causality analysis gives that we can first solve 9 of the systems in parallel followed by some other calculations and thirdly the remaining 9 systems.

Partitioning of the code with maximum of 16 sections can be represented in the graph in Figure 12. Green rectangles represent BLT blocks. These are merged into sections which are outlined in blue. The critical path is marked in red.

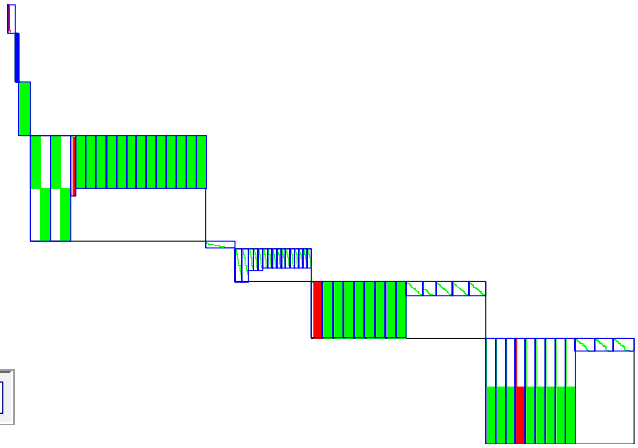

Figure 12: Schedule for Evaporator using max 16 cores

It takes about 45 µs to solve such a nonlinear system of equations. **With 4 cores/8 threads we get a speed-up factor of 2.1.**

Let us make the model more complex by making a finer and more desirable discretization by setting evaporator.n_segAir = 5 and evaporator.n_segRef = 10, which gives that the refrigerant is discretized into 60 cells along the flow direction. Each of these cells as well the sink has pressure and enthalpy as continuous states and there is another 60 continuous time states for the wall temperature giving a total of 182 states. There are now 30 systems of size 112 and 30 systems of size 110. The tearing procedure results in problems with 11 iterations variables for all of them. The simulation time becomes now 58.8 s, compared to 3.7 s for when using a less fine discretization. A causality analysis gives that we can first solve 30 of the system in parallel followed by some other calculations and thirdly the remaining 30 systems can be solved in parallel. It takes 60-90 µs to solve such a system. We can parallelize these computations. **With**

DOI 10.3384/ECP14096363

4 cores/8 threads the factor is 3.4. The calculations between the systems of equations can be split into two parts that can be run in parallel. One part takes 2.2 µs to execute and the other takes 6.7 µs, so running them in series takes 8.9 µs. If we run them in parallel it takes 23 µs, due to the overhead to start up and synchronize parallel threads.

6 Example: Multi-Body Systems

Let us consider simulation of multi-body systems. A major task is to invert the mass-matrix to solve for the accelerations. If the system has kinematic loops there are additional non-linear and linear systems. The factorization of a Jacobian is a major task when solving a system of equations. However, when parallelizing that, it gave faster calculations only if the size of the Jacobian was greater than 300x300.

Another major task is calculation of forces and accelerations. For a tree structured mechanism, there are obvious possibilities to parallelize each branch of the tree into different threads. Such possibilities will be investigated in the future.

6.1 Real-Time Simulation and Inline Integration

What has been discussed above applies to hardware-in-the-loop simulation (HILS) as well as when variable step-size integration methods are used.

For HILS of stiff models such as multibody systems including bushings, implicit methods are needed. Inline integration (Elmqvist, et.al., 2002) can then be used. It means that the discretization formulas are merged with the model equations and the entire merged set of equations is symbolically transformed.

The parallelization method can be used on the merged set of equations. However, implicit methods typically give one large system of non-linear equations to solve, i.e. the above method is not useful as described.

To handle this situation, we have introduced a **decouple operator** which delays the signals one step corresponding to changing from an implicit relationship to an explicit. The result is that the large system of equations decomposes into smaller systems that both are faster to solve and that can be solved in parallel.

Using this technique, we have been successfully simulated vehicles with about 150 DOFs (Degrees-of-Freedom) in real-time using a 1 ms step size and implicit Euler. By decoupling the front and rear wheel suspensions respectively from the chassis (having large mass) and by decoupling each wheel from the wheel suspensions we achieved a speed-up of 1.5. The two decoupling elements decoupling front and rear suspensions from the chassis can be seen in the VDL (Vehicle Dynamics Library) diagram in Figure 13:

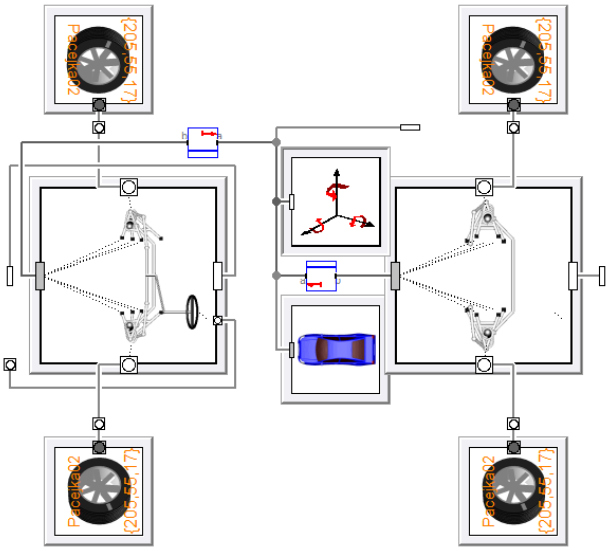

Figure 13: Decoupling of VDL model

Parallelizing the resulting smaller non-linear systems of equations gave a **further speed-up of 2.8** with 4 cores/8 threads.

7 Conclusions

Modelica is well suited for automatic parallelization of equation execution using many cores. We have described a technique for such automatic partitioning. The obtained results are good for models from various domains. For a thermo-dynamic model a speed-up of 3.4 was achieved, for electrical 2.9 and for mechanical HILS 2.8 using 4 cores/8 threads.

References

Aronsson, P., and P. Fritzson (2002). **Multiprocessor Scheduling of Simulation Code from Modelica Models**. In *Proceedings of the 2nd International Modelica Conference*, Oberpfaffenhofen, Germany, Mar. 18–19, 2002.

Aronsson P. (2006): **Automatic Parallelization of Equation-Based Simulation Programs**. *Institutionen för datavetenskap, 2006.*

Casella, F. (2013): **A Strategy for Parallel Simulation of Declarative Object-Oriented Models of Generalized Physical Networks**. *5th International Workshop of Equation-Based Object-Oriented Modeling Languages and Tools,* April 2013, University of Nottingham, UK. Linköping University Electronic Press. http://www.ep.liu.se/ecp_home/index.en.aspx?issue=084.

Elmqvist H., Mattsson S.E., Olsson H. (2002): **New Methods for Hardware-in-the-Loop Simulation of Stiff Models**, *2nd International Modelica Conference,* DLR, Oberpfaffenhofen, Germany, March 18-19.

DOI 10.3384/ECP14096363