Multi-threaded distributed system simulations using the transmission line element method

Braun, R & Krus, P
Division of Fluid and Mechatronic Systems
Department of Management and Engineering
Linköping University, Sweden
E-Mail: robert.braun@liu.se, petter.krus@liu.se

Abstract

By introducing physically motivated time delays, simulation models can be partitioned into decoupled independent sub-models. This enables parallel simulations on multi-core processors. An automatic algorithm is used for partitioning and running distributed system simulations. Methods for sorting and distributing components for good load balancing have been developed. Mathematical correctness during simulation is maintained by a busy-waiting thread synchronization algorithm. Independence between sub-models is achieved by using the transmission line element method. In contrast to the more commonly used centralized solvers, this method uses distributed solvers with physically motivated time delays, making simulations inherently parallel. Results show that simulation speed increases almost proportionally to the number of processor cores in the case of large models. However, overhead time costs mean that models need to be over a certain size to benefit from parallelization.

Key Words: Distributed Solvers, Parallelism, Problem Partitioning, Transmission Line Modelling, System Simulation

1. INTRODUCTION

Utilizing multi-core technology in system simulations is often difficult, due to centralized solver algorithms and many dependencies between the equations. Previous attempts have largely been focused on parallelizing ODE solvers. This includes using block-based parallel corrector-predictor methods [1] and utilizing model topology for parallel matrix inverting [2]. If equations can be divided into independent groups, they can be solved in parallel, for example by using the PVODE solver [3]. Attempts to speed up equation-based simulations using graphical processors (GPU) [4] and to parallelize algorithmic sections in equation-based models [5] have also been made. Although successful, these approaches either provide poor speed-up, affect the numerical stability, can only be applied on a specific model topology, or require time-consuming work on the part of the user. An alternative approach is to divide the equation system into smaller sub-systems, and solve them with distributed solvers. This means that each sub-model, also known as *component*, solves its own equations while communicating only with its neighbouring components. An inevitable consequence of this is that time delays will be introduced in the interfaces between the components. With an appropriate modelling method, however, these can be considered physically motivated.

This paper presents an automatic approach for parallel continuous-time simulation of closely coupled power-transmitting systems. Sub-models are decoupled using the *transmission line element method* (TLM) [6]. An automatic algorithm is used for partitioning the distributed system models for multi-core processors with good load balancing. In contrast to previous implementations, the user can explicitly place the transmission line elements in the model while the parallel execution is automated. In this way, decoupling of sub-models becomes a natural part of the modelling process. The model is parallel already from the beginning, and the user does not need to consider the actual decoupling. Unlike fully automatic methods, the model is never modified without user interaction. Still, the overhead work from the user is kept to a minimum. A synchronization algorithm for running several simulation threads concurrently has also been developed. As a proof of concept, an implementation of this in the Hopsan simulation tool is presented. Results are measured and evaluated by using benchmarking experiments. The feasibility of the method is finally verified on three real simulation models. This has confirmed that the technology can readily be used in real industrial problems.

1.1 Delimitations

The described implementations target only continuous-time system simulation. This includes for example electric circuits, hydraulics, pneumatics and 1D mechanics. All experiments are limited to desktop computers with quad-core processors. This has two reasons. First of all, most users of system simulation software use this kind of computers. Second, most system simulation models are relatively small, heterogeneous and highly non-linear. Benefits from large computer clusters, many-core processors and super-computers are therefore expected to be limited. Communication overhead was found to be large already between two processors on the same computer. For computer clusters communicating over network protocols, communication overhead will be at least one order of magnitude larger. Furthermore, the models at hand exhibit only task parallelism. Hence, the use of graphics processors, which require data parallelism, is not feasible. Data parallelism with TLM can be achieved when modelling field problems, for example electromagnetic wave propagation in 2D or 3D, but this is not within the scope of this work.

All models used in the experiments are using distributed solvers. The equation systems have thus already been decoupled into independent sub-systems. Hence, the difference in simulation speed with and without the TLM elements has not been examined. The decoupling in itself can provide a speed-up, due to the fact that the time for solving an equation system does not scale linear to the size of the system. Such comparisons would therefore not be fair and it would be difficult to generalize the results.

1.2 Related work

Early attempts to run TLM simulation on parallel computers were conducted as described in [7]. Later attempts were made using transputers, which were early predecessors of multi-core processors, for hydraulic systems [8] and electric systems [9]. It was also used for parallel co-simulations of multi-body systems [10]. Parallel simulations of hydraulic systems on multi-core processors were later conducted [11]. It has also been used to parallelize models in the Modelica language for cluster computers [12] and for automatic parallelization on multi-core processors [13]. Other experiments using techniques similar to TLM has been conducted by using dynamic decoupling [14] and by using automatic block diagonalization [15]. The work presented in this paper is unique in that it provides fully automated parallelism, while the placement and parameterization of the TLM elements are still fully visible to the user.

Several authors have studied the use of TLM for simulating electromagnetic wave propagation, known as the transmission line matrix method [16]. This enables a large number of independent nodes, and thus a high degree of parallelism. By mapping a time-stepped TLM model to a parallel discrete-event execution, speed-up could be achieved on up to 25 cores [17] and subsequently 5000 cores [18] could be achieved. Scalable simulation with up to 127,500 cores was later demonstrated [19]. In contrast to these experiments, the work described in this article focuses on heterogeneous non-linear 1D models, which provides a significantly more coarse-grained parallelism.

Nowadays most simulation tools have support for parallel processing. For example, the MATLAB/Simulink environment offers a parallel computing toolbox [20], which for example enables parallel for-loops, batch processing and GPU processing. Likewise, the Mathematica program also contains parallel tools based on executing independent kernel processes concurrently with a distributed memory approach [21]. It is for example possible to manipulate and evaluate symbolic expressions in parallel, and to generate parallel code. These tools can be very useful and provide great performance improvements for many problems. Nevertheless, the problem remains that if an equation system contains dependencies, it must be decoupled before it can be used for parallel simulation. While the mentioned tools do not provide built-in

solutions for this, support for TLM could theoretically be implemented in any simulation tool or code library with support for parallel processing. In this paper, the Threading Building Blocks (TBB) by Intel, a C++ library for shared-memory programming, was used.

Several standards for distributed simulation exist. Examples include *distributed interactive simulation* (DIS) for real-time wargaming [22] and its successor *high-level architecture* (HLA), an interoperability standard for distributed simulation [23]. These standards are mainly designed for communication and synchronization of interacting simulation tools. While they could be used also to solve the synchronized bag-of-task problems described in this article, it is considered an over-complicated solution, since the implementations are made in an existing integrated simulation tool.

2. TRANSMISSION LINE ELEMENT METHOD

The transmission line element method (TLM) is a simulation technique for numerical decoupling of power transmitting systems. These can for example be fluid systems, electrics, mechanics, radio waves, magnetics or acoustics. The method has its origins in bi-lateral delay lines [22], the method of characteristics as used in the HYTRAN simulation tool [23]**,** and in transmission line modelling as described in [24]. It is currently used in the Hopsan simulation environment [25] and for multi-body co-simulation by SKF [26]. Compatibility between TLM and other simulation tools was demonstrated in [27]. In detail, it uses wave propagation to introduce physically motivated time delays between components in the system. This is based on the fact that ultimately all physical interactions have a limited information propagation speed. Consequently, it is possible to introduce time delays directly into the model equations. The length of the delays can then be viewed as model parameters. Hence, any errors introduced by the delays is by definition a modelling error. Numerical accuracy is thus not affected. Modelling errors will occur only if the length of delays does not match the corresponding physical delays.

This method is especially useful when calculating wave propagation phenomena with high precision. The method is also suitable for co-simulation between simulation tools. Furthermore, the time delays can be used to decouple parts of a model into independent sub-models, making this method suitable for parallel execution. If an equation system is divided into smaller subsystems, time delays will be introduced in the intersection variables. If these delays are not handled, they will induce numerical errors. By using physically motivated delays, however, numerical errors can be avoided. In the following derivations fluid systems will be used, which means that the intensity variable will be pressure (p) and the flow variable will be hydraulic flow (q).

Figure 1: With the transmission line element method, capacitive parts of the model can be replaced with impedances and physically motivated time delays.

The fundamental equations (1) and (2) tells us that the pressure at one end of a transmission line is equal to the characteristic impedance (Z_C) times the sum of the flow at this point at time t and the flow at the other end at time $t - \Delta T$ added with the pressure at the other end at time $t - \Delta T$. The time constant ΔT represents the time it takes for a wave to travel between the ends.

$$p_1(t) = Z_C[q_1(t) + q_2(t - \Delta T)] + p_2(t - \Delta T) \quad (1)$$

$$p_2(t) = Z_C[q_2(t) + q_1(t - \Delta T)] + p_1(t - \Delta T) \quad (2)$$

These equations can be decoupled by introducing *wave variables* (c). These represent the information that travels with a wave from one end to the other at time ΔT.

$$c_1(t) = Z_C q_2(t - \Delta T) + p_2(t - \Delta T) \quad (3)$$

$$c_2(t) = Z_C q_1(t - \Delta T) + p_1(t - \Delta T) \quad (4)$$

Combining equations (1-4) yields the decoupled equations:

$$p_1(t) = Z_C q_1(t) + c_1(t) \quad (5)$$

$$p_2(t) = Z_C q_2(t) + c_1(t) \quad (6)$$

In most cases, the characteristic impedance represents a physical property in the component and thereby remains constant over time. For a lossless transmission line, where resistance is zero, the impedance equals the square root of the inductance over capacitance.

$$Z_C = \sqrt{\frac{L}{C}} \quad (7)$$

Systems are divided into Q-type and C-type components, representing the decoupled sub-models and the TLM elements, respectively. Q-types receive impedances and wave variables and use them to calculate and write intensity and flow variables. In fluid systems, these are typically flow restrictors. C-types do the opposite; they receive intensity and flow variables, and use them to calculate impedance and wave variables. These are modelled as capacitances, and should thus be inserted at flexible or compressible parts of the model. A natural consequence is that a component can never be directly connected to another of same type. It also induces that all components at any given time are independent of other components of the same type. By combining this with distributed solvers, it is thus possible to simulate all components of the same type in parallel. [6]

Each component can independently be simulated from time t to the next communication point at time $t + \Delta T$. During this interval it is technically possible for each component to use an arbitrary step-size and numerical solver. This is related to the concept of *lookahead* used in parallel discrete-event simulation [30].

6.1 Illustrating Example

The method is here demonstrated by a minimalistic example model consisting of two traversing bodies connected with a spring, see Fig. 2. The bodies have masses M_A and M_B, positions $x_A(t)$ and $x_B(t)$, and are subjected to external forces $F_A(t)$ and $F_B(t)$, respectively. The model is decoupled by introducing a physically motivated time delay ΔT in the spring. Equation systems for the two bodies are shown in equations 8 and 9. $F_{BA}(t)$ and $F_{AB}(t)$ are the forces exerted by the spring on the bodies. The impedance can be computed as $Z_C = k_s \Delta T$, where k_s is the spring stiffness.

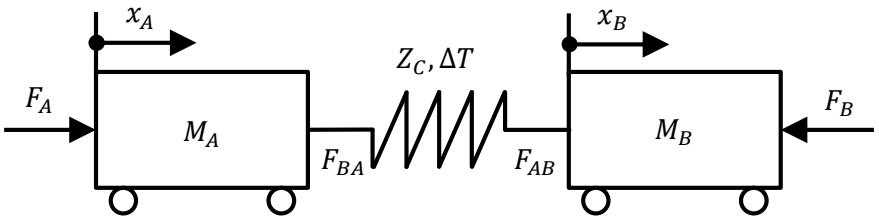

Figure 2: A model of two bodies connected with a spring is used as a minimalistic example.

$$\begin{cases} M_A \dfrac{\partial \dot{x}_A(t)}{\partial t} = F_A(t) - F_{BA}(t) \\ F_{BA}(t) = F_{AB}(t - \Delta T) + Z_C[\dot{x}_A(t) + \dot{x}_B(t - \Delta T)] \end{cases} \tag{8}$$

$$\begin{cases} M_B \dfrac{\partial \dot{x}_B(t)}{\partial t} = F_{AB}(t) - F_B(t) \\ F_{AB}(t) = F_{BA}(t - \Delta T) + Z_C[\dot{x}_B(t) + \dot{x}_A(t - \Delta T)] \end{cases} \tag{9}$$

System 8 contains two equations and two unknowns: $\dot{x}_A$ and F_{BA}. The external force and the delayed variables can be considered to be known variables. Similarly, system 9 contains two equations and the unknowns $\dot{x}_B$ and F_{AB}. Since the two systems share no common unknowns, each system can be solved independently. The parallel execution is performed as follows:

1. Solve systems 8 and 9 on separate processors for time t.
2. Exchange variables $\dot{x}_A(t)$, $\dot{x}_B(t)$, $F_{AB}(t)$ and $F_{BA}(t)$ between the processors.
3. Increase time variable $t = t + \Delta T$.
4. Start over from point 1.

Adding more components to the system is trivial. Algebraic loops will not occur, since there is always a time delay between any two components. The sequential execution time of a decoupled model will always equal the sum of the execution time for solving each independent system.

3. MULTI-THREADED DISTRIBUTED SIMULATIONS

A multi-core processor contains several processing cores, which can perform different tasks simultaneously. Computer programs are executed in threads and each core can only work on one thread at a time. Threads are assigned to cores automatically by the task scheduler in the operating system. The application thus has no control over the mapping between threads and cores [28]. By dividing the simulation code into one thread per processor core, the parallel speed-up can theoretically be at most proportional to the number of cores. The number of threads must not necessarily match the number of processing units. How many parts the model is divided into is more generally known as the *degree of parallelization*, p. With the TLM method, this is limited by the locations in the model where time delays of reasonable lengths can be introduced. For simplicity, it is assumed that all cores have the same computational speed. This is true for most modern desktop multi-core computers.

In comparison to centralized solver algorithms, distributed solver techniques with communication time delays have the advantage of being parallel by nature. Even in a single-threaded simulation, all components are executed independently of each other. It is for this reason a very suitable method for taking advantage of multi-core architecture. This can be exploited by two different approaches. The first is to create vectors with all components of the same type and then simulate subsets of these in one thread per processor core. This results in a fine-grained partition, increasing the possibilities of good load balancing. It can also be fully automated. The second method is to divide the model into subsystems, and manually assign each subsystem to a thread. This paper only discusses the first approach. Due to the high frequency of data exchange, only shared memory parallelism has been considered.

An important aspect of multi-threaded programming is *thread safety*, meaning that threads must not write to the same memory address simultaneously or read where another thread is

simultaneously writing. If they did, the resulting data would likely become invalid. With the TLM method this is naturally guaranteed, because a component is never simulated at the same time as the components it communicates with.

The Hopsan simulation environment also has a third type of component, viz. the signal (arithmetic) components, called S-type. These must be simulated in the order they are connected to ensure mathematical correctness. They can only be parallelized if they can be divided into independent groups, with no connections in-between. An algorithm that identifies such groups was implemented, but the improvement in speed was found to be limited, due to the fact that this kind of components is usually very simple compared to physics-based components. Another task that must be carried out each time step is to log the model variables. This work is parallel by nature and each thread can simply be assigning a subset of the variables.

The parallelization algorithm will suffer from overhead time costs due to thread spawning and synchronization. Assuming no speed-up in signal components, the total time for one iteration step T_{step} with p processors and perfect load balancing will thus be:

$$T_{step} = \frac{T_C}{p} + \frac{T_Q}{p} + T_S + \frac{T_{log}}{p} + T_{spawn} + T_{sync} \tag{10}$$

T_{spawn} and T_{sync} are the times required for thread spawning and synchronization. T_C, T_Q, T_S and T_{log} are the times required for simulating the different kind of components and log data variables. These cannot be changed at a system level, and can thus be considered constant. The objective of the parallel algorithm is therefore to minimize the overhead time for spawning and synchronizing simulation threads.

4. PARTITIONING OF COMPONENTS

Achieving good load balancing is important for maximizing the benefits of parallelism. The workload must be divided as equally as possible among the processing cores. The easiest approach is to assign a processor to each component manually. Better results can however be achieved by using automatic scheduling algorithms. In these experiments an offline scheduler, where the partitioning remains constant during execution, was found sufficient. Each component requires a different amount of time to simulate. One way to obtain this information is to use *performance-aware components*, which can inform the scheduler of their individual workload. An alternative solution is to measure the simulation time for each component by using *pre-simulation profiling*. This is performed by executing the model single-threaded for a few time steps and measuring the time spent on each component. Both these methods of course only hold if the simulation time for each component remains the same during the simulation. This is however a reasonable assumption when each decoupled sub-system is relatively small. The simulation is initialized in the following steps:

1. Measure simulation times with pre-simulation profiler
2. Sort components by measured simulation time
3. Distribute components evenly over p vectors for each type
4. Distribute data nodes evenly over p vectors
5. Spawn p simulation threads
6. Assign one vector of each type to each thread
7. Execute all threads in parallel

The partition problem can be seen as a special case of the subset-sum problem [29]. Consider a set S of components, each with a different measured simulation time. The goal is to partition S into several subsets $S_1, S_2, \dots, S_p$, where p is number of processing cores, such that the sum of

the measured times in all subsets are equal. For natural reasons it is not possible to find an exact solution in most cases. The problem can be expressed as:

Find a partition $S = \{S_1, S_2, \dots, S_p\}$ such that
$max(\sum S_1, \sum S_2, \dots, \sum S_p)$ is minimized.

This problem is NP-complete, and going through all possible combinations is obviously not feasible. An optimal solution to this problem can be calculated by a dynamic programming solution. This is however not suitable for floating-point precision problems. The remaining option is to use heuristic methods to obtain an approximate solution. One such method is the greedy algorithm [30], which goes through the set in descending order (from slowest to fastest) and assigns each component to the thread with the currently smallest total time. This results in a 4/3-approximation of the optimal solution. Hence, the makespan of the sorted components will be at most 33% longer than the optimal solution, which is assumed acceptable in comparison to the errors in the profiling results. Another heuristic method is the differencing algorithm [31], which iterates over the set and moves the n slowest components to a temporary subset until the main set is empty. It then iterates backwards through the temporary subsets and inserts the components to the final subsets, depending on their total measured time. This method has a slightly higher chance of finding an optimal solution. Both the heuristic algorithms have a running time of $O(n^2)$. In the experiments in this paper, the greedy algorithm is used.

5. SYNCHRONIZATION OF SIMULATION THREADS

An important criterion in the transmission line element method is that components remain synchronized during the simulation. A component must never begin an iteration before all components of the previous type have finished. This would not only result in the risk of the component working on outdated variables; it would also endanger the thread safety and thereby possibly cause a program crash. These problems are avoided by implementing a barrier synchronization. This is suitable because all threads must stop and wait for each other at the same time. A busy-waiting implementation with a common lock variable is used. Each thread increments a common atomic counter variable when they arrive at the barrier. The first thread is the "master" simulation thread. It will wait for all other threads to increase the counter, and then unlock the barrier so that all threads can continue, see figure Fig. 3. Source code for the barrier class and simulation loop is listed in Appendix A. Barrier counters are not reset until after the next barrier to prevent deadlocks caused by one thread reaching one barrier before another has exited the previous one.

Proof of correctness: Assume n+1 threads and initial conditions ctr = 0 and $lock_i$ = true. When all threads have arrived then ctr = n. This implies that ctr→0, $lock_{i+1}$→true, $lock_i$→false and that the master thread will continue. Setting $lock_i$ = false will in turn allow all slave threads to continue. Hence, if all threads arrive at the barrier then a) all threads will continue, b) no thread will continue before next barrier is locked and c) no thread will continue before the counter variable is reset.

It is assumed that all threads finish their execution of each step in finite time. If not, the remaining threads will never be allowed to continue and the simulation will halt. While this can be problematic in some cases, it can also be useful. Components that stop and wait for external input can, for example, be used for step-wise debugging or real-time synchronization. In general, the creator of each sub-model must be responsible for its stability properties.

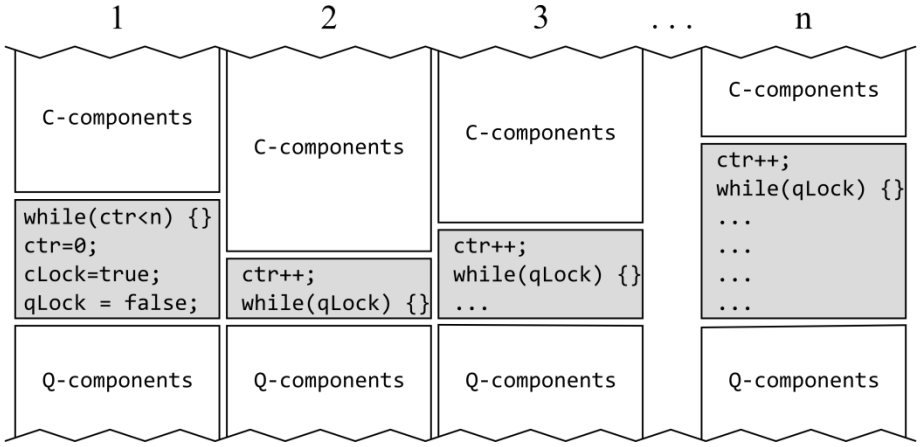

Figure 3: Simulation threads are synchronized by a busy-waiting barrier. Locking and unlocking are handled by the master thread.

6. PRACTICAL EXPERIMENTS

Hopsan is an open-source integrated cross-platform distributed-solver simulation environment developed at Linköping University. It is primarily intended for simulating fluid and mechatronic systems. The simulation core is fully object-oriented and uses pre-compiled component libraries, so that no compilation is needed at runtime. Component objects are separated by independent node connection objects, where variable data is stored (see Fig. 4). [32] [25]

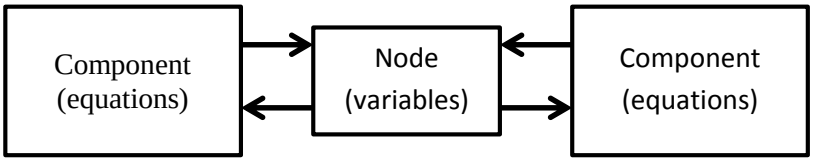

Figure 4: Components in Hopsan are represented as objects, separated by independent node objects. This is used to ensure stability and thread safety.

This approach offers implicit synchronization, since two directly connected components are never simulated at the same time. The multi-threaded algorithm presented in this paper was implemented by using the Threading Building Blocks (TBB) package from Intel, a C++ template library for writing programs with platform independent shared-memory parallelism [33]. Cross-platform support was considered important for the application at hand.

The experiments were performed on a quad-core GNU/Linux system, a six-core Windows system and a dual quad-core Windows system. All non-critical applications and services were turned off before starting the experiments. The Hopsan process was also assigned high priority.

6.1 Synthetic models

Special benchmark component models of C- and Q-type were created for the experiments. These are included in the official release of Hopsan, to make it possible for other researchers to verify the results. Each benchmark component receives one input signal, for which it calculates the faculty. By this method, each component's workload is always proportional to the input signal. This makes it easy to adjust the model's computational load. A benchmark model consisting of several benchmark components of each type was used. In each experiment, 100

simulations of 10000 iterations were run with 1, 2, 4, 6 or 8 threads. Another 100 simulations were run single-threaded to measure parallelization slowdown (see below). A random workload was assigned to the model at each simulation. Perfect load balancing was assumed. It was found early on that the number of connections between components had no significant effect on simulation performance. This was therefore not analysed further.

Results were measured by four performance metrics. The first two describe the speed-up. Absolute speed-up SU_{abs} is defined as the multi-threaded time t_A compared to single-threaded time t_s, see eq. (11). Relative speed-up SU_{rel} compares multi-threaded speed with running just one thread using the multi-threaded algorithm, as shown in eq. (12). Another way to describe speed-up is by relative efficiency EF, defined as relative speed-up over number of threads p, see eq. (13). Finally, the parallelization slowdown SL, defined as time with one thread using the multi-threaded algorithm divided by single-threaded time, was investigated according to eq. (14).

$$SU_{abs}(p,n) = \frac{t_s(n)}{t_A(p,n)} \tag{11}$$

$$SU_{rel}(p,n) = \frac{t_A(1,n)}{t_A(p,n)} \tag{12}$$

$$EF(p,n) = \frac{SU_{rel}(p,n)}{p} \tag{13}$$

$$SL(n) = \frac{t_A(1,n)}{t_s(n)} \tag{14}$$

The first experiment was performed on a GNU/Linux system with an Intel Core i7 quad-core processor. First, the parallelization slowdown was measured to be 1.015, which means that the multi-threaded algorithm by itself increases execution time by 1.5. Benchmarking results are shown in Fig. 5. The speed-up is close to linear with four threads, although a significant static overhead from thread creation exists.

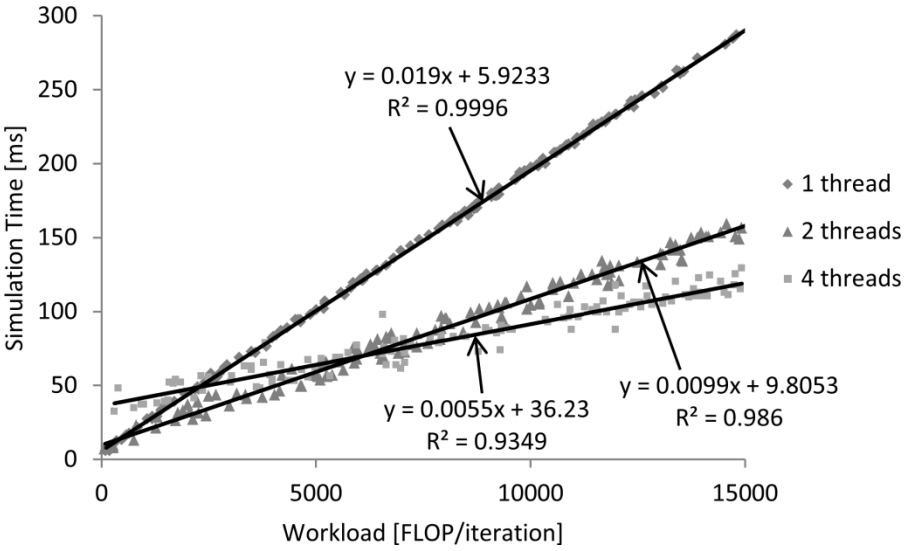

Figure 5: The synthetic model was executed on a quad-core processor with one, two and four threads. Simulation time is plotted against total model workload.

The same test was repeated on a Windows system with an Intel Xeon processor with six cores. Parallelization slowdown was measured to be 1.4. Measurement results are shown in Fig. 6. Speed-up is still increasing linearly with the number of threads, but the overhead is greater than with the quad-core processor.

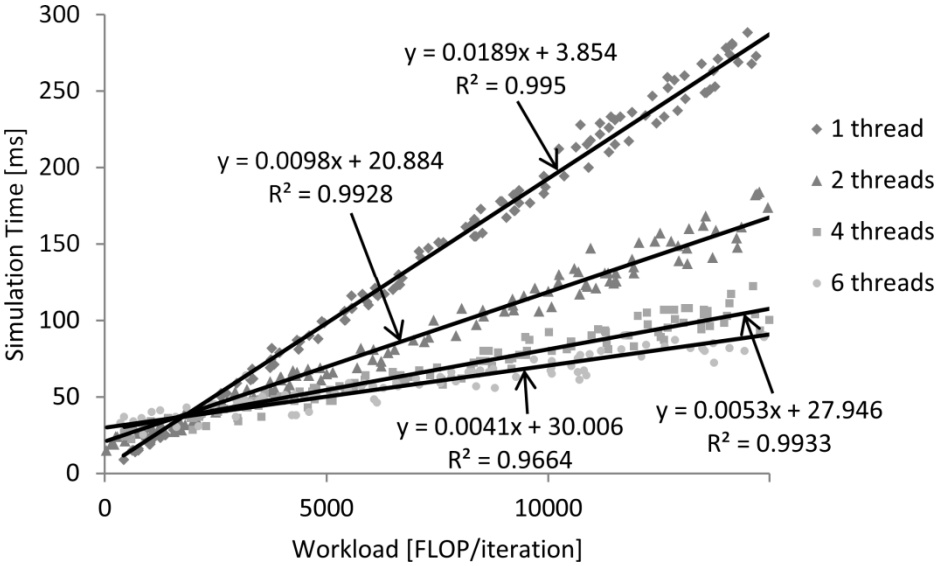

Figure 6: The synthetic model was executed on a six-core processor with one, two, four and six threads. Simulation time is plotted against total model workload.

To investigate the results from parallel processors, the same experiments were performed on a Windows system with two separate quad-core processors. Parallelization slow-down was measured to be 0.5. Benchmarking was performed for one, two, four and eight threads. According to Fig. 7, results show an almost linear speed-up up to four threads. With eight threads, simulation speed is still increased, but is no longer linear to the number of threads. Benchmarking results are shown in Table I.

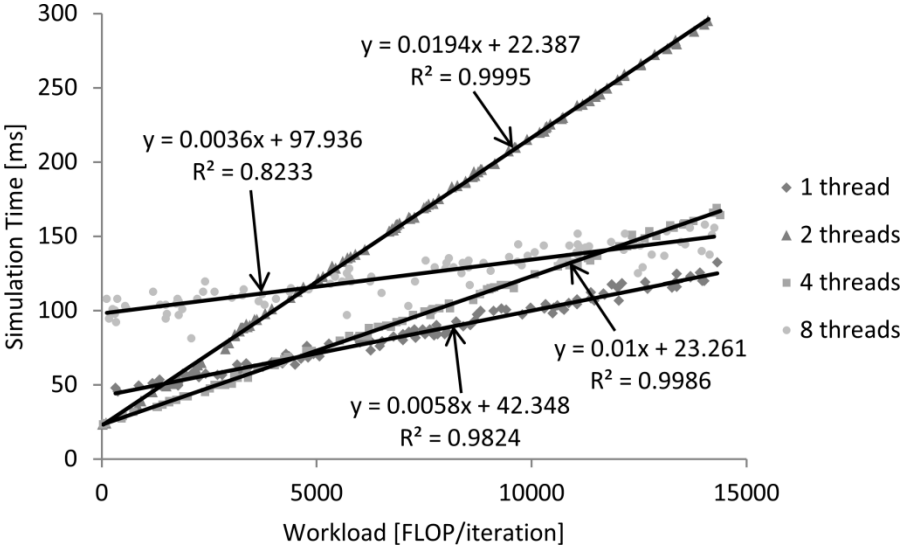

Figure 7: The synthetic model was executed on a dual quad-core processor with one, two, four and eight threads. Simulation time is plotted against total model workload.

Table I: Benchmarking results for the synthetic model on three different processors

	$SU_{abs}(p,n)$	$SU_{rel}(p,n)$	$EF(p,n)$

Linux, 4 cores, 2 threads	1.95	1.92	96 %
Linux, 4 cores, 4 threads	3.51	3.46	86 %
Windows, 6 cores, 2 threads	1.95	1.93	97 %
Windows, 6 cores, 4 threads	3.61	3.57	89 %
Windows, 6 cores, 6 threads	4.67	4.61	77 %
Windows, 2x4 cores, 2 threads	1.95	1.94	97 %
Windows, 2x4 cores, 4 threads	3.36	3.35	84 %
Windows, 2x4 cores, 8 threads	5.42	5.39	67 %

6.2 Real models

As a final verification, the algorithm was tested on three realistic simulation models. All of them have been used actively in industrial and/or academic projects. The first model is provided by Atlas Cocpo Rock Drills. It consists of a hydraulic rock drill for mining machinery. The TLM method is especially useful for simulation of wave propagation. In this case, this is used for very accurate simulations of shock waves. The drill steel is modelled as one or more TLM elements. The model also contains the hydraulic system, auxiliary systems and a model of the rock. While simulation time for this model is less than a few minutes, it is mainly used for design optimization which requires very many simulations. Running one such optimization overnight is often not possible without parallel execution. Faster simulations may save much time and money in the development process, and also shorten the time-to-market for new products.

A recent trend in fluid power is digital hydraulics, where large control valves are replaced by many smaller ones. The second test model is currently used in a research project on digital hydraulics for aircraft applications [34]. It consists of four linear actuators, each controlled by 27 valves. One such actuator is shown in Fig. 8. A 4-chamber piston is supplied by 3 pressure lines, which enables 81 different force levels. Models of such systems are especially suitable for parallel simulations. The main difficulty is to reduce overhead costs. The model will be very fine grained, because every independent valve will consist of only a few equations. While sequential simulation time is only a few seconds, such models are typically used for developing and tuning control algorithms, for time-consuming high-dimension design optimization, and for real-time hardware-in-the-loop simulations.

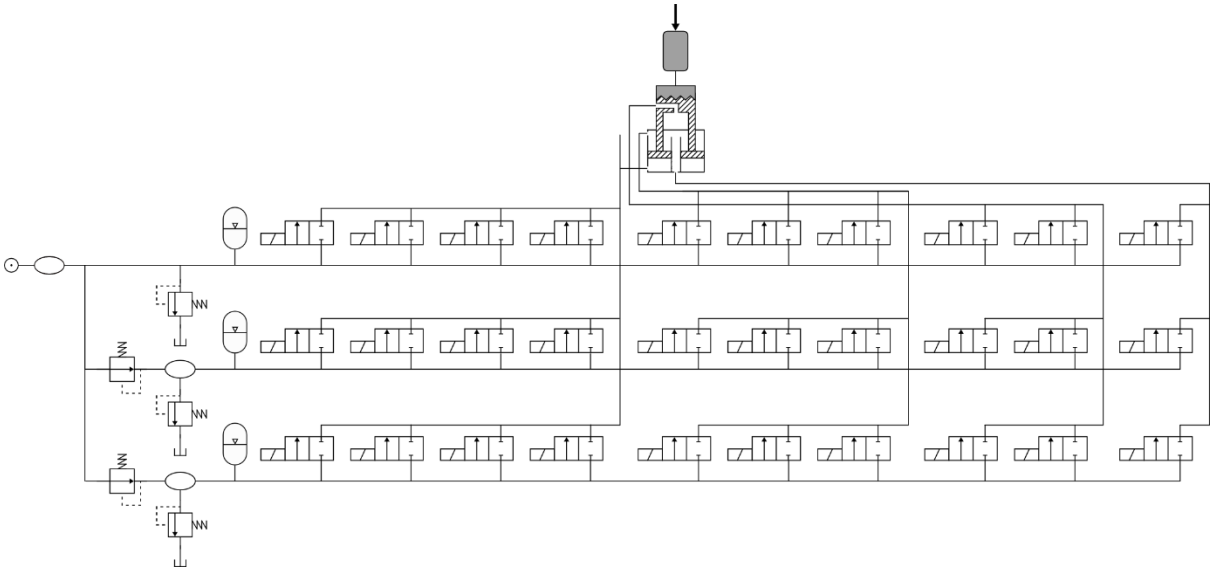

Figure 8: A model of a digital hydraulic system was used to benchmark the method.

As a final test, a multi-body mechanics model is used. Such models are usually difficult to parallelize. However, it has been shown that also stiff mechanical joints can be replaced by TLM elements [35]. In this case, a pendulum with eight arms is used. A similar experiment using parallel co-simulation showed that increasing the number of arms had a very small impact on simulation time [26]. Multi-body mechanics require a small step size for avoiding numerical instability, which makes them time consuming to simulate.

All three models were simulated on quad-core, six-core and dual-quad core processors. Results are shown in table II. The quad-core processor gives good speed-up for the first two models, and an adequate result for the pendulum model. With six cores, the speed-up is actually lower than with four cores. The reason for this is likely that the model size is too small, so that overhead costs become more noticeable. The dual quad-core processor performs poorly for all models, but especially for the pendulum model, which are much slower than the sequential measurement. Overhead from communication between the processors are apparently a large issue.

Table II: Speed-up results for the three models on quad-core, six-core and dual quad-core processors

	$SU_{abs}(4)$	$SU_{abs}(6)$	$SU_{abs}(8)$
Hydraulic rock drill	4.01	3.31	1.72
Digital hydraulic system	3.66	3.45	2.09
2D pendulum	2.70	1.66	0.36

FUTURE WORK

An offline scheduler algorithm using busy-waiting barrier synchronization was developed. Dynamic scheduling methods such as task pools or work-stealing algorithms would probably work as well. These were not investigated in this paper because the barrier algorithm was considered sufficient. However, they could be useful for models where the workload varies over time. Evaluating such methods would be a valuable continuation to this work. Other future work could be to efficiently implement this method in real-time simulations and embedded systems. For models with a large number of identical components, such as the digital hydraulics test model, vectorization for SIMD architectures, for example GPU cards, could be a competitive alternative.

The increasing number of processor cores raises the issue of at which level a simulation shall be parallelized. This article deals with parallelism at the model level. However, it might also be possible to use parallel equation solvers inside each decoupled component. Another possibility is to run several models in parallel on different processor cores, for example for optimization or sensitivity analysis. Finding an optimal strategy for exploiting the available processing units in the most economical way constitutes an interesting future research challenge.

CONCLUSIONS

It is shown that simulation time can be four times shorter with a quad-core processor in the case of large models and decent load balancing. With six cores, the speed-up is increased further. Parallel processors are also examined. These provide significant speed-up, but not close to linear. This is likely due to communication delays between the processors. The experiments are performed in the Hopsan simulation tool. Since the method is based on modifying the model equations it should, however, be applicable also on other programs. A great benefit of the method is that no numerical errors are introduced, but only modelling errors, which can be controlled by the model parameters. Apart from faster simulations, the method is also useful for avoiding numerical errors at the software interfaces in co-simulations.

ACKNOWLEDGEMENT

This work was supported by ProViking research school and the Swedish Foundation for Strategic Research (SSF). The rock drill benchmark model was provided by Atlas Copco Rock Drills, who also supported the work with feedback and tested the implementation on real applications.

REFERENCES

[1] D. Voss and S. Abbas, “Block predictor-corrector schemes for the parallel solution of ODEs,” *Computers and Mathematics with Applications,* vol. 33, no. 6, 1997.

[2] P. Nordling and A. Sjö, “Parallel solution of modular ODEs with application to rolling bearing dynamics,” *Mathematics and Computers in Simulation,* vol. 44, 1997.

[3] G. D. Byrne and A. C. Hindmarsch, “PVODE, an ODE solver for parallel computers,” *International Journal of High Performance Computing Applications,* vol. 13, 1999.

[4] K. Stavåker, Contributions to parallel simulation of equation-based models on graphics processing units, Linköping: Linköping University, 2011, p. 96.

[5] M. Gebremedhin, ParModelica: Extending the algorithmic subset of Modelica with explicit parallel language constructs for multi-core simulation, Linköping University, 2011.

[6] P. Krus, A. Jansson and J. O, "Distributed simulation of hydro-mechanical system," in *The Third Bath International Fluid Power Workshop*, Bath, England, 1990.

[7] A. Jansson and P. Krus, "Real-time simulation using parallel processing," in *The Second Tampere International Conference On Fluid Power*, Tampere, Finland, 1991.

[8] J. D. Burton, K. A. Edge and C. R. Burrows, "Partitioned simulation of hydraulic systems using transmission-line modelling," *ASME WAM*, 1993.

[9] K. Fung and S. Hui, "Transputer simulation of decoupled electrical circuits," *Mathematics and Computers in Simulation,* vol. 42, pp. 1-13, 1996.

[10] D. Fritzson and J. Ståhl, "Transmission line co-simulation of rolling bearing applications," *The 48th Scandinavian conference on simulation and modeling,* 2007.

[11] R. Braun, P. Nordin, B. Eriksson and P. Krus, "High performance system simulation using multiple processor cores," in *The Twelfth Scandinavian International Conference On Fluid Power*, Tampere, Finland, 2011.

[12] K. Nyström and P. Fritzson, "Parallel simulation with transmission lines in modelica," in *5th International Modelica Conference*, Vienna, Austria, 2006.

[13] M. Sjölund, M. Gebremedhin and P. Fritzson, "Parallelizing Equation-Based Models for Simulation on Multi-Core Platforms by Utilizing Model Structure," in *Proceedings of the 17th Workshop on Compilers for Parallel Computing,* 2013.

[14] A. V. Papadopoulos and A. Leva, "Automating efficiency-targeted approximations in modelling and simulation tools: dynamic decoupling and mixed-mode integration," *SIMULATION,* vol. 90, no. 10, pp. 1158-1176, 2014.

[15] A. B. Khaled, M. B. Gaid and D. Simon, "Parallelization approaches for the time-efficient simulation of hybrid dynamical systems: Application to combustion modeling," in *Proceedings of the 5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, 2013.

[16] P. Johns and R. Beurle, "Numerical solution of 2-dimensional scattering problems using a transmission-line matrix," in *Proceedings of the Institution of Electrical Engineers*, 1971.

[17] D. W. Bauer Jr and E. H. Page, "Optimistic parallel discrete event simulation of the event-based transmission line matrix method," in *Winter Simulation Conference*, 2007.

[18] D. W. Bauer Jr, C. D. Carothers and A. Holder, "Scalable time warp on blue gene supercomputers," in *roceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, 2009.

[19] S. K. Seal and K. S. Perumalla, "Reversible parallel discrete event formulation of a tlm-based radio signal propagation model," *ACM Transactions on Modeling and Computer Simulation (TOMACS),* vol. 22, no. 1, p. 4, 2011.

[20] The MathWorks Inc, "Parallel computing toolbox documentation," 2014.

[21] Wolfram Research, "Parallel computing - Wolfram Mathematica 9 documentation," 2014.

[22] R. C. Hofer and M. L. Loper, "DIS today [Distributed interactive simulation]," *Proceedings of the IEEE,* vol. 83, no. 8, pp. 1124-1137, 1995.

[23] U.S. Department of Defense, "High Level Architecture Interface Specification Version 1.3," 1998.

[24] D. M. Auslander, “Distributed System Simulation with Bilateral Delay-Line Models,” *Journal of Basic Engineering*, pp. 195-200, June 1968.

[25] Air Force Aero Propulsion Laboratory, Aircraft hydraulic system dynamic analysis, 1997.

[26] P. B. Johns and M. A. O'Brian, “Use of the transmission line modelling (t.l.m.) method to solve nonlinear lumped networks,” *The Radio And Electronic Engineer*, vol. 50, no. 1, pp. 59-70, 1980.

[27] M. Axin, R. Braun, A. Dell'Amico, B. Eriksson, P. Nordin, K. Pettersson, I. Staack and P. Krus, “Next generation simulation software using transmission line elements,” in *Fluid Power and Motion Control*, Bath, England, 2010.

[28] A. Siemers, D. Fritzson and I. Nakhimovski, “General meta-model based co-simulations applied to mechanical systems,” *Simulation Modelling Practice and Theory*, vol. 17, no. 4, pp. 612--624, 2009.

[29] J. Larsson, “A Framework for Implementation-Independent Simulation Models,” *SIMULATION*, vol. 82, no. 9, pp. 563-579, 2006.

[30] Y.-B. Lin and E. D. Lazowska, “Exploiting lookahead in parallel simulation,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 4, pp. 457-469, 1990.

[31] M. Herlihy and N. Shavit, The art of multiprocessor programming, Burlington, MA: Morgan Kaufmann, 2008.

[32] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, “The subset-sum problem,” in *Introduction to Algorithms*, vol. 2, MIT Press, 2001, pp. 1043-1056.

[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, “Greedy Algorithms,” in *Introduction to Algorithms*, 2 ed., MIT Press, 2001, pp. 370-404.

[34] N. Karmarker and R. M. Karp, The differencing method of set partitioning, University of California at Berkeley, 1983.

[35] B. Eriksson, P. Nordin and P. Krus, “Hopsan NG, a C++ implementation using the TLM simulation technique,” in *The 51st Conference On Simulation And Modelling*, Oulu, Finland, 2010.

[36] J. Reinders, Intel Threading Building Blocks: Outfitting C++ for multi-core processor parallelism, O'Reilly Media, 2007.

[37] H. C. Belan, C. C. Locateli, B. Lantto, P. Krus and V. J. D. Negri, “Digital secondary control architecture for aircraft application,” in *Proceedings of the Sixth Workshop on Digital Fluid Power*, Linz, Austria, 2015.

[38] P. Krus, “Modeling of mechanical systems using rigid bodies and transmission line joints,” *Journal of dynamic systems, measurement, and control*, vol. 121, no. 4, pp. 606--611, 1999.

APPENDIX A: BARRIER SYNCHRONIZATION SOURCE CODE

```
class BarrierLock {
public:
    BarrierLock(size_t nThreads) {
        mnThreads=nThreads;
        mCounter = 0;
        mLock = true;
    }

    inline void lock() {
        mCounter=0;
        mLock=true;
    }

    inline void unlock() {
        mLock=false;
    }

    inline bool isLocked() {
        return mLock;
    }

    inline void increment() {
        ++mCounter;
    }

    inline bool allArrived() {
        return (mCounter == (mnThreads-1));
    }
private:
    int mnThreads;
    std::atomic<int> mCounter;
    std::atomic<bool> mLock;
};
```

Listing 1: Public member functions in the barrier lock class

```
for(size_t s=0; s<numSimSteps; ++s) {
    while(!pBarrier_S->allArrived()) {}
    pBarrier_C->lock();
    pBarrier_S->unlock();

    [simulate signal components]

    while(!pBarrier_C->allArrived()) {}
    pBarrier_Q->lock();
    pBarrier_C->unlock();

    [simulate C-type components]

    while(!pBarrier_Q->allArrived()) {}
    pBarrier_N->lock();
    pBarrier_Q->unlock();

    [simulate Q-type components]

    while(!pBarrier_N->allArrived()) {}
    pBarrier_S->lock();
    pBarrier_N->unlock();

    [log data variables]
}
```

Listing 2: Execution loop for the master thread.

```
for(size_t s=0; s<numSimSteps; ++s) {
    pBarrier_S->increment();
    while(pBarrier_S->isLocked()){}

    [simulate signal components]

    pBarrier_C->increment();
    while(pBarrier_C->isLocked()){}

    [simulate C-type components]

    pBarrier_Q->increment();
    while(pBarrier_Q->isLocked()){}

    [simulate Q-type components]

    pBarrier_N->increment();
    while(pBarrier_N->isLocked()){}

    [log data variables]
}
```

Listing 3: Execution loop for the slave threads.