Journal Title
XX(X):1–19

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

A PDEVS Simulator Supporting Multiple Synchronization Protocols: Implementation and Performance Analysis

Ben Cardoen[1], Stijn Manhaeve[1], Yentl Van Tendeloo[1], and Jan Broeckhove[1]

Abstract

With the ever increasing complexity of simulation models, parallel simulation becomes necessary to perform simulation within reasonable time bounds. The built-in parallelism of Parallel DEVS is often insufficient to tackle this problem on its own. Several synchronization protocols have been proposed, each with their distinct advantages and disadvantages. Due to the significantly different implementation of these protocols, most Parallel DEVS simulation tools are limited to only one such protocol. In this paper, we present a Parallel DEVS simulator, grafted on C++11 and based on PythonPDEVS, supporting both conservative and optimistic synchronization protocols. The simulator not only supports both protocols but also has the capability to switch between them at runtime. We evaluate the performance gain obtained by choosing the most appropriate synchronization protocol. A comparison is made to adevs in terms of CPU time and memory usage, to show that our modularity does not hinder performance. We further allow for an external component to gather simulation statistics, on which runtime switching between the different synchronization protocols can be based. Model allocation is also studied to see how our conservative and optimistic synchronization protocols are affected by good and bad allocations.

Submission for the *Special Issue of Simulation: SpringSim 2016 special issue*.

Introduction

DEVS [1] is a popular formalism for modelling complex dynamic systems using a discrete-event abstraction. In fact, it can serve as a simulation "assembly language" to which models in other formalisms can be mapped [2]. A number of tools have been constructed by academia and industry that allow the modelling and simulation of DEVS models.

But with the ever increasing complexity of simulation models, parallel simulation becomes necessary to perform the simulation within reasonable time bounds. Whereas Parallel DEVS [3] was introduced to increase parallelism, its inherent parallelism is often insufficient [4]. Several synchronization protocols from the discrete event simulation community [5] have been applied to DEVS simulation [6]. Through synchronization protocols, different simulation cores can be at different points in simulated time, increasing parallelism significantly, at the

[1]University of Antwerp, Belgium

Corresponding author:
Yentl Van Tendeloo
University of Antwerp
Middelheimlaan 1
2020 Antwerp, Belgium
Email: Yentl.VanTendeloo@uantwerpen.be

cost of synchronization overhead. While several parallel DEVS simulation kernels exist, they are often limited to a single synchronization protocol. The reason for different synchronization protocols, however, is that their distinct nature makes them applicable in different situations, each outperforming the other in specific models [7]. The applicability of parallel simulation capabilities of current tools is therefore constrained to specific domains.

This paper introduces *DEVS-Ex-Machina*[*] ("*dxex*"): our simulation tool [8] which offers multiple synchronization protocols: no synchronization (sequential execution), conservative synchronization, or optimistic synchronization. The selected synchronization protocol is transparent to the simulated model: users only determine which protocol to use. Users who simulate a wide variety of models, with different ideal synchronization protocols, can run the same model with different synchronization protocols. Since the ideal synchronization protocol might change throughout the simulation, we also support runtime switching between these protocols. This runtime switching can be based on performance metrics, which are logged during simulation. Information is made available to a separate component, where a choice can be made about which synchronization protocol to use next. Additionally, we investigate in this paper how model allocation influences the performance of our synchronization protocols. To this end, we have included an allocation component to our simulation kernel.

Our tool is based on *PythonPDEVS*, but implemented in C++11 for increased performance, using features from the new C++14 standard when supported by the compiler. Unlike *PythonPDEVS*, *dxex* only supports multicore parallelism, thus no distributed simulation.

We implemented a model that, depending on a single parameter, changes its ideal synchronization protocol. Using several models, we demonstrate the factors influencing the performance under a given synchronization protocol. *Dxex*, then, is used to compare simulation using exactly the same tool, but with a varying synchronization protocol. With *dxex* users can always opt to use the fastest protocol available, and through its modularity, users could even implement their own. To verify that this modularity does not hinder performance, we compare to another Parallel DEVS simulation kernel, called *adevs* [9].

The remainder of this paper is organized as follows: Section BACKGROUND introduces the necessary background on synchronization protocols. Section DEVS-EX-MACHINA elaborates on our design that enables this flexibility. In Section PERFORMANCE EVALUATION, we evaluate performance of our tool by comparing its different synchronization protocols, and compare to *adevs*. We continue by introducing runtime switching of synchronization protocols and different options for model allocation in Section RUNTIME SWITCHING and Section MODEL ALLOCATION, respectively. Related work is discussed in Section RELATED WORK. Section CONCLUSIONS AND FUTURE WORK concludes the paper and gives future work.

Background

This section briefly introduces the synchronization protocols used by *dxex*: conservative and optimistic synchronization.

Conservative Synchronization

The first synchronization protocol we introduce is *conservative synchronization* [5]. In conservative synchronization, a node progresses independent of all other nodes, up to the point in time where it can guarantee that no causality errors happen. When simulation reaches this point, the node blocks until it can guarantee a new time until which no causality errors occur. In practice, this means that all nodes are aware of the current simulation time of all other nodes, and the time it takes an event to propagate (called *lookahead*). Deadlocks can occur due to a dependency cycle of models. Multiple algorithms are defined in the literature to handle both the core protocol, as well as resolution schemes to handle or avoid the deadlocks [5].

The main advantage of conservative synchronization is its low overhead if the lookahead is high. Each node then simulates in parallel, and sporadically notifies other nodes about its local simulation time. The disadvantage, however, is that the amount of parallelism is explicitly limited by the lookahead. If a node can influence another

[*] **https://bitbucket.org/bcardoen/devs-ex-machina**

(almost) instantaneously, no matter how rarely it occurs, the amount of parallelism is severely reduced. The user is required to define the lookahead, using knowledge about the model's behaviour. Defining lookahead is far from a trivial task if there is no detailed knowledge of the model. Even slight changes in the model or its allocation can change the lookahead, and can therefore have a significant influence on simulation performance.

Optimistic Synchronization

A completely different synchronization protocol is *optimistic synchronization* [10]. Whereas conservative synchronization prevents causality errors at all costs, optimistic synchronization allows them to happen, but corrects them. Each node simulates as fast as possible, without taking note of any other node. It assumes that no events occur from other nodes, unless it has explicitly received one at that time. When this assumption is violated, the node rolls back its simulation time and state to right before the moment when the event has to be processed. As simulation is rolled back to a time prior to the event must be processed, the event can then be processed as if no causality error ever occurred.

Rolling back simulation time requires the node to store past model states, such that they can be restored later. All incoming and outgoing events need to be stored as well. Incoming events are injected again after a rollback, when their time has been reached again. Outgoing events are cancelled after a rollback, through the use of anti-messages, as potentially different output events have to be generated. Cancelling events, however, can cause further rollbacks, as the receiving node might also have to roll back its state. In practice, a single causality error can have significant repercussions on performance.

Further changes are required for unrecoverable operations, such as I/O and memory management. These are only executed after the lower bound of all simulation times, called *Global Virtual Time* (GVT) [5], has progressed beyond their execution time.

The main advantage is that performance is not limited by a small lookahead, caused by a very infrequent event. If an (almost) instantaneous event rarely occurs, performance is only impacted when it occurs, and not at every simulation step. The disadvantage is unpredictable performance due to the arbitrary cost of rollbacks and their propagation. If rollbacks occur frequently, state saving and rollback overhead can cause simulation to grind to a halt. Apart from overhead in CPU time, a significant memory overhead is present: intermediate states are stored up to a point where it can be considered *irreversible*. Note that, while optimistic synchronization does not explicitly depends on lookahead, performance still implicitly depends on lookahead. Instead of depending on the theoretically defined safe lookahead, performance is related to the actually perceived lookahead.

DEVS-Ex-Machina

Historically, *dxex* is based on *PythonPDEVS* [11]. Python is a good language for prototypes, but performance has proven insufficient to compete with other simulation kernels [12]. *Dxex* is a C++11-based Parallel DEVS simulator, based on the design of *PythonPDEVS*. Whereas the feature set is not too comparable, the architectural design, simulation algorithms, and optimizations, are highly similar.

We will not make a detailed comparison with *PythonPDEVS* here, but only mention some supported features. *Dxex* supports, similarly to *PythonPDEVS*, the following features: direct connection [13], Dynamic Structure DEVS [14], termination conditions [15], and a modular tracing and scheduling framework [11]. We do not elaborate on these features in this paper. But whereas *PythonPDEVS* only supports optimistic synchronization, *dxex* support multiple synchronization protocols (though only in parallel). This is in line with the design principle of *PythonPDEVS*: allow users to pass performance hints to the simulation kernel. In our case, a user can pass the simulation kernel the synchronization protocol to use for this model, or even switch the synchronization protocol during runtime. Our implementation in C++11 also allows for optimizations which were plainly impossible in an interpreted language. *Dxex* will use new optimizations from the C++14 standard when possible.

Since there is no universal DEVS model standard, *dxex* models are incompatible with *PythonPDEVS* and vice versa. This is due to *dxex* models being grafted on C++11, whereas *PythonPDEVS* models are grafted on Python.

In the remainder of this section, we will elaborate on our prominent new feature: the efficient implementation

of multiple synchronization protocols within the same simulation tool, which are offered transparently to the model.

Synchronization protocols

We previously explained the existence of different synchronization protocols, each optimized for a specific kind of model. As no single synchronization protocol is ideal for all models, a general purpose simulation tool should support multiple protocols. Currently, most parallel simulation tools choose only a single synchronization protocol due to the inherent differences between protocols. We argue that a real general purpose simulation tool should support sequential, conservative, and optimistic synchronization, as is the case for *dxex*.

These different protocols relate to three different model characteristics. Conservative synchronization for when high lookahead exists between different nodes, and barely any blocking is necessary. Optimistic synchronization for when lookahead is unpredictable, or there are rare (almost) instantaneous events. Finally, sequential simulation is still required for models where parallelism is bad, where all protocols actually slow down simulation.

Sequential Our sequential simulation algorithm is very similar to the one found in *PythonPDEVS*, including many optimizations. Minor modifications were made, though, such that it can be overloaded by different synchronization protocol implementations. This way, the DEVS simulation algorithm is implemented once, but parts can be overridden as needed. In theory, more synchronization protocols (*e.g.*, other algorithms for conservative synchronization) can be added without altering our design.

An overview of *dxex*'s design is given in Figure 1. It shows that there is a simulation `Core`, which simulates the `AtomicModels` connected to it. The superclass `Core` represents the sequential simulation core. Subclasses define specific variants, such as `ConservativeCore` (conservative synchronization), `OptimisticCore` (optimistic synchronization), and `DynamicCore` (Dynamic Structure DEVS).

Conservative For conservative synchronization, each node must determine the nodes it is influenced by. Each model needs to provide a lookahead function, which determines the lookahead depending on the current simulation state. Within the returned time interval, the model promises not to raise an event. A node aggregates this information to compute its Earliest Output Time (EOT). This value is communicated to other nodes through shared memory.

Reading and writing to shared memory is done through the use of the new C++11 synchronization primitives. Whereas this was also possible in previous versions of the C++ standard, by falling back to non-portable C functions, it was not a part of the C++ language standard. C++11 further allows us to make the implementation portable, as well as more efficient: the compiler makes further optimizations to heavily used components.

Optimistic For optimistic synchronization, each node must be able to roll back to a previous point in time. This is often implemented through the use of state saving. This needs to be done carefully in order to avoid unnecessary copies, and minimize the overhead. We use the default: explicitly save each and every intermediate state. Mattern's algorithm [16] is used to determine the GVT, as it runs asynchronously and uses only $2n$ synchronization messages. Once the GVT is found, all nodes are informed of the new value, after which fossil collection is performed, and irreversible actions are committed.

The main problem we encountered in our implementation is the aggressive use of memory. Frequent memory allocation and deallocation caused significant overheads, certainly when multiple threads do so concurrently. This made us switch to the use of thread-local (using *tcmalloc*) memory pools. Again, we made use of specific new features of C++11, that are not available in Python, or even previous versions of the C++ language standard.

Synchronization Protocol Transparency

We define synchronization protocol transparency as having a single model, which always can be executed on each supported synchronization kernel, without any modifications. User should thus only provide one model, implemented in C++11, which can be simulated using sequential execution, conservative synchronization, or optimistic synchronization. The synchronization protocol to use is a simple configuration option. The exception is conservative synchronization, where a lookahead function

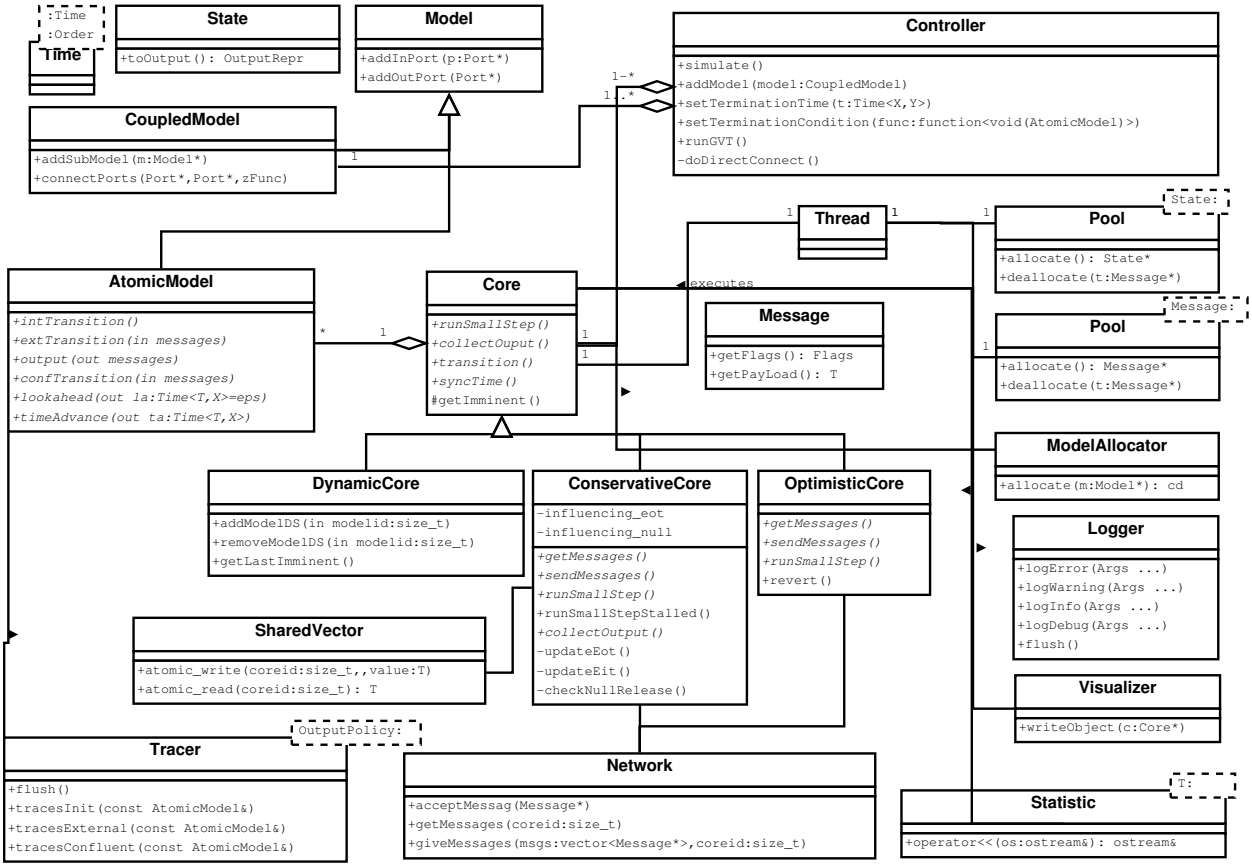

Figure 1. *Dxex* design.

is required, which is not used in other synchronization kernels. Two options are possible: either a lookahead function must always be provided, even when it is not required and possibly not used, or we use a default lookahead function if none is defined.

Always defining a lookahead function might seem redundant, especially should the user never want to use conservative synchronization. Defining the lookahead is difficult and can often be unpredictable. The more attractive option is for the simulation tool to provide a default lookahead function, such that simulation can run anyway, but likely not at peak performance. Depending on the model, simulation performance might still be faster than sequential simulation.

Defining a lookahead function is therefore recommended in combination with conservative synchronization, but is not a necessity, as a default ϵ (*i.e.*, the smallest representable timestep) is used otherwise. Providing this default implementation has no impact in sequential or optimistic simulation, since the function is never called the compiler will optimize it out. By providing this default implementation in the model base class we ensure that a model can run in sequential, optimistic and conservative simulation by default.

Increasing Parallelism

The goal of our contribution is to increase simulation performance as much as possible, leveraging parallel computation in the process. Parallelizing the simulation

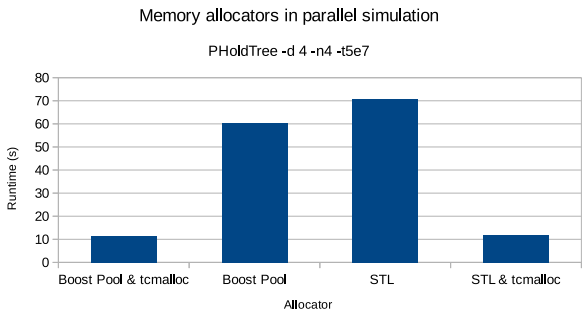

Figure 2. Effect of memory allocators on parallel execution time.

kernel goes further, however, than merely implementing the different synchronization protocols.

We observed that after implementing all synchronization protocols, performance was still not within acceptable levels. Profiling revealed that most of the overhead was caused by two issues: memory management and random number generation. For both, it is already known that they can have significantly impact on parallelizability of code, since they introduce sequential blocks. Both were tackled using approaches that are in common use in the parallel programming world. We briefly mention how the application of these techniques influences our performance.

Memory Management Memory management is traditionally seen as one of the major bottlenecks in parallel computation [17]: memory bandwidth doesn't increase as fast as the number of cores using it. While this is always a problem, it is aggravated in *dxex* by providing automatic memory management for events and states. A model written for sequential simulation will run correctly in a conservative or optimistic simulation without altering (from the point of view of the modeller) the (de)allocation semantics of events or states.

Furthermore, allocating and deallocating memory by making calls to the operating system, as typically done by calls such as `malloc`, happens sequentially. To counter this, our memory allocators are backed by a thread-aware pooling library. In a sequential simulation kernel no allocated event persists beyond a single time advance, even allowing the use of an arena-style allocator. Conservative and optimistic simulation need to use generic pool allocators since events are shared across kernels and thus have a different lifetime.

Intra-kernel events are pooled aggressively, whereas inter-kernel events need a GVT algorithm to determine when safe deallocation can occur, even in conservative synchronization. A simulation with many inter kernel events suffers a performance hit, whereas the impact of many intra-kernel events can be minimized using arena allocators.

Dxex uses *Boost Pool* [18] allocators in parallel simulation kernels and arena-style allocators for sequential simulation. The latter can be faster, but at the cost of extra configuration. The allocators are supplemented by the library *tcmalloc* [19], which reduces lock contention in `malloc` calls.

We primarily investigate this for optimistic simulation, as this is the most memory consuming mode of simulation [5]. Simulation execution times for all four combinations are shown in Figure 2. Optimistic simulation greatly benefits from the use of *tcmalloc*, regardless of the allocator. Nonetheless the pool allocator also reduces the allocation overhead, though only by a relatively small fraction. Both techniques are required to reduce the overhead of memory allocations in *dxex*, and are turned on by default.

Both pools and *tcmalloc* try to keep memory allocated instead of returning it to the Operating System (OS). As a result, the OS will usually report memory consumption that is higher than the actual amount of stored data.

Random Number Generators Random Number Generators (RNG) are another aspect of the program that results in sequentialization. All accesses to the RNG will result in the modification of a global (*i.e.*, shared between threads) variable. This easily becomes a bottleneck in simulation, since random numbers are a common occurence in simulation [20]. As such, a non-trivial amount of time in a simulation is often spent waiting for an RNG.

We of course still need to guarantee determinism and isolation between the calls to the RNG, as well as avoiding excessive synchronization. *Dxex* uses the Tina RNG collection (TRNG) [21] as an alternative random number generator with performance and multithreading in mind. Since the RNG is an implicit part of the state in the

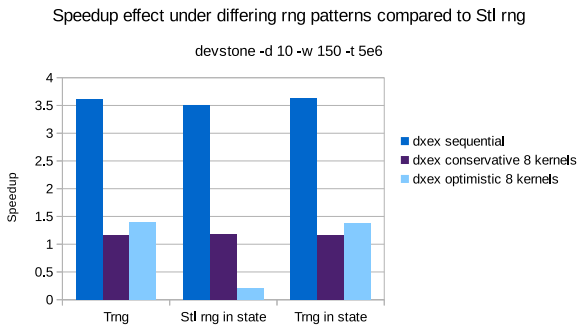

Figure 3. Speedup with different RNG usage patterns compared to STL random number generator.

Parallel DEVS formalism, though often not implemented as such, we evaluated performance for both approaches: one global RNG per thread, and one RNG per atomic DEVS model.

We see in Figure 3 that storing the RNG in the state is very expensive for the default STL random number generator. This is primarily caused by the significant difference in size: 2504 bytes for the STL random number generator, and 24 bytes for the Tina random number generator. *Dxex*'s sequential and conservative kernels are insensitive to storing the RNG object in the atomic model state, since no copying/state saving occurs in *dxex* conservative simulation. The optimistic kernel is clearly affected, as it needs to copy more bytes in every transition due to state saving.

Figure 3 shows that *dxex* in sequential simulation is three times faster using TRNG compared to using the STL RNG. For parallel simulation, the synchronization overhead seems to be the main bottleneck, as seen by the big speedup gap between sequential and parallel simulation. Conservative synchronization is almost insensitive to the changing of the RNG, though a slight increase in performance can be noted. Optimistic synchronization slows down significantly when the RNG becomes part of the model state, since the state needs to be copied as well. This becomes a significant overhead when using the STL RNG, since performance plummets to a fraction of the original. Using TRNG avoids this problem completely, as the size of the RNG state is negligible.

Performance Evaluation

In this section, we evaluate the performance of different synchronization protocols in *dxex*. We also compare to *adevs* [9], currently one of the most efficient simulation kernels [22], to show that our modularity does not impede performance. CPU time and memory usage is compared for both sequential and parallel simulation.

We start off with a comparison of sequential simulation, to show how *adevs* and *dxex* relate in this simple case. For the parallel simulation benchmarks, results are presented for both conservative and optimistic synchronization.

For all benchmarks, results are well within a 5% deviation of the average, such that only the average is used in the remainder of this section. The same compilation flags were used for both *adevs* and *dxex* benchmarks ("`-O3 -flto`"). To guarantee comparable results, no I/O was performed during benchmarks. Before benchmarking, simulation traces were used to verify that *adevs* and *dxex* return exactly the same simulation results. Benchmarks were performed using Linux, but our simulation tool works equally well on Windows and Mac. The exact parameters for each benchmark can be found in the repository, as well as the data used in this paper.

Benchmark Models

We use three different benchmark models, covering different aspects of the simulation kernel.

First, the *Queue* model, based on the *HI* model of DEVStone [23], creates a chain of hierarchically nested atomic DEVS models. A single generator pushes events into the queue, which get consumed by the processors after a fixed or random delay. It takes two parameters: the width and depth of the hierarchy. This benchmark shows how the complexity of the simulation kernel behaves for an increasing amount of atomic models, and an increasingly deep hierarchy. An example for width and depth equal to 2 is shown in Figure 4.

Second, the *Interconnect* model, a merge of PHOLD [24] and the *HI* model of DEVStone [23], creates n atomic models, where each model has exactly one output port. Similar to PHOLD, all models are connected to one another, but all through the same port: every model receives each generated event. The model takes one parameter: the number of models. This benchmark shows the complexity of event creation, event

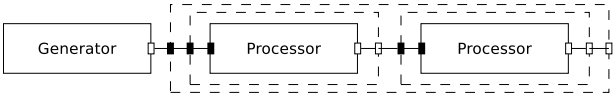

Figure 4. Queue model for depth and width 2.

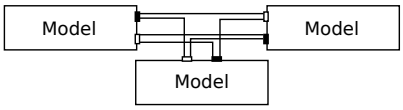

Figure 5. Interconnect model for three models.

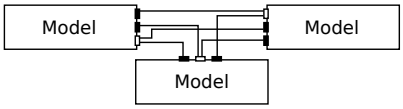

Figure 6. PHOLD model for three models.

routing, and simultaneous event handling. An example for three models is shown in Figure 5.

Third, the *PHOLD* model [24], creates n atomic models, where each model has exactly $n-1$ output ports. Each atomic model is directly connected to every other atomic model. After a random delay, an atomic model sends out an event to a randomly selected output port. Output port selection happens in two phases: first it is decided whether the event should be sent within the same node, or outside of the node. Afterwards, a uniform selection is made between the remaining models. The model takes two parameter: the percentage of remote events, which determines the fraction of messages routed to other nodes, and the percentage of priority events. Priority events are events generated in a very short time after the previous event. This benchmark shows how the simulation kernel behaves in the presence of many local or remote events, in combination with a varying percentage of high-priority event.s An example for four models, split over two nodes, is shown in Figure 6.

Sequential Simulation

We start by evaluating sequential simulation performance, in order to obtain a baseline for our comparison of parallel simulation performance.

Queue For the first benchmark, we tested the effect of hierarchical complexity of the model in the performance

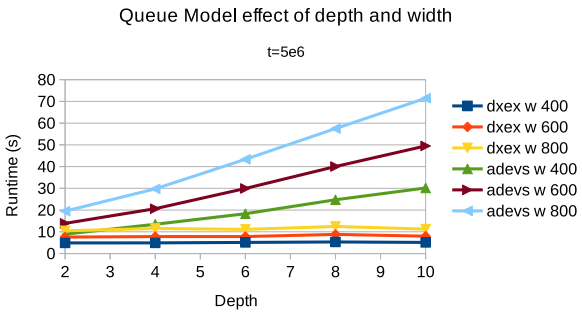

Figure 7. Queue benchmark results for sequential simulation.

of the simulator. A set of three tests was performed, where each test has the same number of models but an increasing depth. The results can be seen in Figure 7. Since *dxex* performs direct connection on the model, there is no performance hit when the depth is increased. Direct connection only needs to be done at initialization, so it is a one time cost that is negligible when simulation termination end time is sufficiently large. *Adevs*, on the other hand, suffers from the increased depth, even though some similar (but not identical) optimization to event passing was made [25]. With every new hierarchical layer, routing an event from one atomic model to the next becomes more expensive, resulting in an increase in runtime.

Interconnect In the Interconnect model, we increase the number of atomic models, quadratically increasing the number of couplings and the number of external transitions. As shown in Figure 8, *adevs* now outperforms *dxex* by a fair margin. Analysis showed that this is caused by the high amount of events: event creation is much slower in *dxex* than it is in *adevs*, despite *dxex*'s use of memory pools. To shield the user from threading and deallocation concerns *dxex* provides an event superclass from which the user can derive to create a specialized event type. Copying, deallocation, and tracing are done at runtime, adding significant overhead when events happen frequently. Profiling the benchmarks clearly shows the increasing cost of output generation and deallocation as the determining factor in the gap in performance.

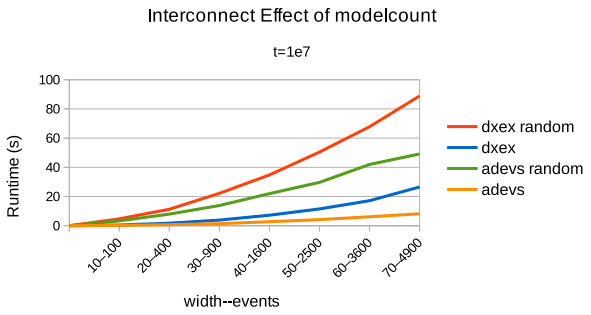

Figure 8. Interconnect benchmark results for sequential simulation.

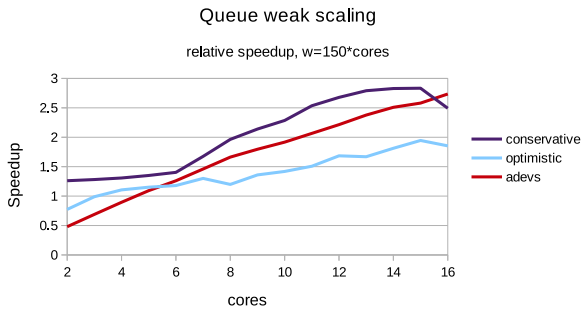

Figure 10. Queue model weak scaling speedup compared to *dxex* sequential.

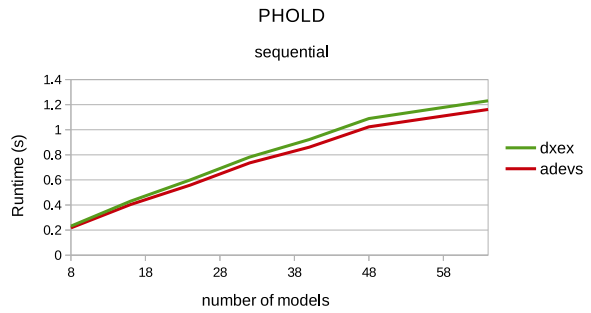

Figure 9. PHold benchmark results for sequential simulation.

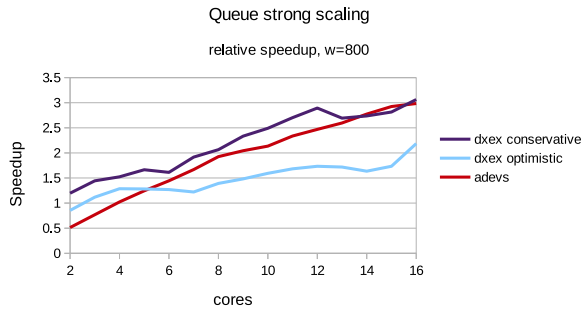

Figure 11. Queue model strong scaling speedup compared to *dxex* sequential.

PHold The PHold model is very similar to the Interconnect model. The biggest difference is that the amount of messages sent is much lower. The number of events scales linear with the number of models, not quadratic. Figure 9 shows that the performance of *dxex* and *adevs* are very close to each-other, with *adevs* slightly outperforming *dxex*.

Parallel Simulation

We now continue by describing our parallel simulation performance for different synchronization protocols, and compared to *adevs*. The speedup of *adevs* is computed with the corresponding *dxex* sequential benchmark. This was done to take into account the performance difference observed in sequential simulation.

Queue The Queue model is one single chain of models, resembling a pipeline. This structure can be exploited to prevent cyclic dependencies in the parallel simulation.

Figure 11 shows the speedup compared to sequential simulation for a fixed problem size. As the number of kernels increases, the optimistic kernel quickly becomes the worst choice. This is mainly caused by the pipeline structure of the model: the last models in the queue only respond to incoming messages and therefore have to be rolled back frequently. The difference between *dxex* conservative and *adevs* becoms smaller when more and more cores are used. The same effect can be seen for weak scaling in Figure 10.

Interconnect In the Interconnect model, we determine how broadcast communication is supported across multiple nodes. The number of models is now kept

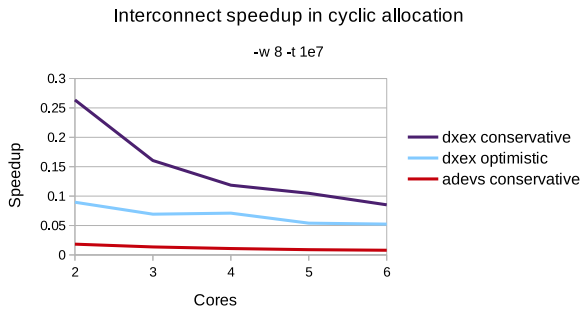

Figure 12. Interconnect benchmark results for parallel simulation.

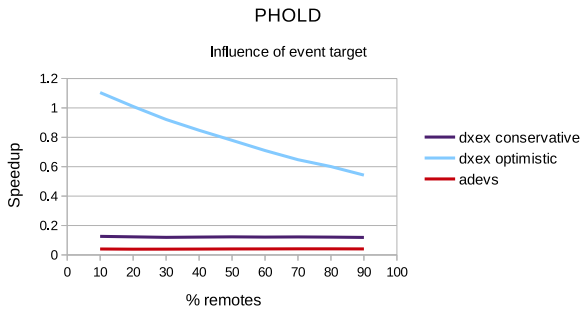

Figure 13. PHOLD benchmark results for parallel simulation using four kernels, four atomics per node, with varying percentage of remote events.

constant at eight. Results are shown in Figure 12. When the number of nodes increases, performance decreases due to increasing contention in conservative simulation and the increasing number of rollbacks in optimistic simulation. All models depend on each other and have no computational load whatsoever, negating any possible performance gain by executing the simulation in parallel.

PHOLD In the PHOLD model, we first investigate the influence of the percentage of remote events on the speedup. A remote event in this context is an event that is sent from a model on one kernel to a model on another simulation kernel. When remote events are rare, optimistic synchronization rarely has to roll back, thus increasing performance. With more common remote events, however, optimistic synchronization quickly slows down due to frequent rollbacks. Conservative synchronization, on the other hand, is mostly unconcerned with the number of remote events: the mere fact that a remote event can happen, causes it to block and wait. Even though a single synchronization protocol is always ideal in this case, it already shows that different synchronization protocols respond differently to a changing model.

Adevs is significantly slower during conservative synchronization. Analysis of profiling callgraphs shows that exception handling in *adevs* is the main cause. To keep the models equivalent, the *adevs* version does not provide the {begin,end}Lookahead methods, which accounts for the exception handling. These functions require the user to implement a state saving method. But

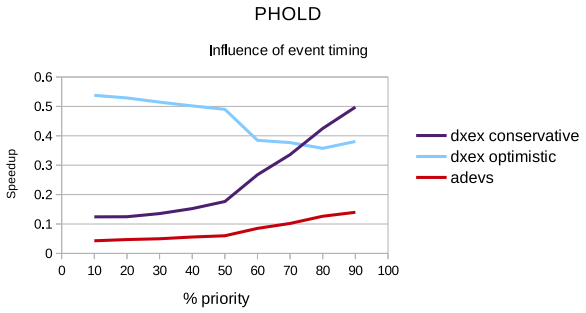

Figure 14. PHOLD benchmark results for parallel simulation using four kernels, with varying amount of high-priority events.

in contrast to *PythonPDEVS* and *dxex*, which handle this inside the kernel, users need to manually define this. We feel this would lead to an unfair comparison as we would like to keep the models agnostic of the underlying protocols across all benchmarks.

We slightly modified the PHOLD benchmark to include high-priority events. Contrary to normal events, high-priority events happen almost instantaneously, restricting lookahead to a very small value. Even when normal events occur most often, conservative synchronization always blocks until it can make guarantees. Optimistic synchronization, however, simply goes forward in simulation time and rolls back when these high-priority events happen. This situation closely mimics the model

typically used for comparing conservative and optimistic synchronization [5].

Figure 14 shows how simulation performance is influenced by the fraction of these high-priority events. If barely any high-priority events occur, conservative synchronization is penalized due to its excessive blocking, which often turned out to be unnecessary. When many high-priority events occur, optimistic synchronization is penalized due to its unconditional progression of simulation, which frequently needs to be rolled back. Results show that there is no single perfect synchronization algorithm for this model: depending on configuration, either synchronization protocol might be better.

Memory Usage

Apart from simulation execution time, memory usage during simulation is also of great importance. While execution time only becomes a problem if it takes way too long, coming short only a bit of memory can make simulation unfeasible. We therefore also investigate memory usage of different synchronization protocols, and again compare to *adevs*.

We do not tackle the problem of states that become too large for a single machine to hold. This problem can be mitigated by distribution over multiple machines, which neither *dxex* or *adevs* support.

Remarks Both *dxex* and *adevs* use *tcmalloc* as memory allocator, allowing for thread-local allocation. Additionally, dxex uses memory pools to further reduce the frequency of expensive system calls (*e.g.*, malloc and free). *Tcmalloc* only gradually releases memory back to the OS, whereas our pools will not do so at all. Due to our motivation for memory usage analysis, we will only measure peak allocation in maximum resident set size as reported by the OS.

Results Figure 15 shows the memory used by the different benchmarks. Results are in megabytes, and show the total memory footprint of the running application (*i.e.*, text, stack, and heap). Note the logarithmic scale due to the high memory consumption of optimistic synchronization.

Unsurprisingly, optimistic synchronization results show very high memory usage due to the saved states.

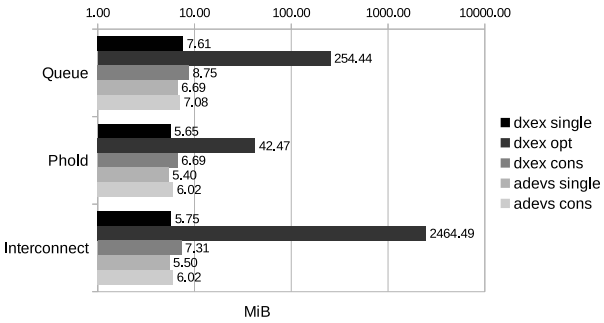

Figure 15. Memory usage results.

Note the logarithmic scale that was used for this reason. Optimistic synchronization results vary heavily depending on thread scheduling by the operating system, as this influences the drift between nodes. Comparing similar approaches, we notice that *dxex* and *adevs* have very similar memory use.

Conservative simulation always uses more memory than sequential simulation, as is to be expected. Additional memory is required for the multiple threads, but also to store all events that are processed simultaneously.

Conclusions on Performance Evaluation

We have shown that our contribution is invaluable for high performance simulation: depending on the expected behaviour, modellers can choose the most appropriate synchronization protocol. But even with the right synchronization protocol, we have seen that two problems remain.

First, although one of both synchronization protocols might be ideally suited for specific model behaviour, nothing guarantees that the model will exhibit the same behaviour throughout the simulation. Similarly to the polymorphic scheduler [12], we wish to make it possible for the ideal option to be switched during simulation. When changes to the model behaviour are noticed, the used synchronization protocol is modified as well.

Second, the allocation of models is tricky and has a significant impact on performance. While our parallel speedup for the Queue model, for example, was rather high, this is mostly due to characteristics of the model: the dependency graph does not contain any cycles. When

cycles were introduced, as in the Interconnect model, performance became disastrous.

In the next two sections, we elaborate on these two problems.

Runtime Switching

Simply because a synchronization protocol is ideal at the start of the simulation, does not mean that it will still be ideal during the simulation. It is well known, and repeated in the previous section, that model behaviour significantly influences the ideal synchronization protocol. Contrary to many modelling formalisms, the DEVS formalisms makes it possible to model basically any kind of discrete event model. As such, it is possible for the model to significantly change its behaviour throughout the simulation.

Defining the ideal synchronization protocol at the start of the simulation, when information about future model behaviour is scarce, might therefore not offer the best possible performance. In *dxex*, we not only make it possible to define the synchronization protocol to use, but also to change this decision throughout simulation. To do this, all kernels are notified of the switch and they are forced to stop simulation. When stopped, each kernel instantiates a new core, with the new synchronization protocol, that is provided with the simulation state of the previous core. Simulation is then resumed with the new cores after the previous ones are destroyed.

As usual, switching imposes an overhead and should thus only be done if the benefits outweigh the induced overhead. This overhead depends on the size of the model and the number of simulation cores. For a simple model and a few cores, the overhead is less than a second.

Although we currently only support manual switches between different synchronization protocols, this is not necessarily the case. Ideally, a new component is added to the simulation kernel, which monitors model behaviour and simulation performance, and toggles between them automatically. Our interface is augmented with the necessary bindings for such a decision component. Also, our interface is augmented with an interface for statistics gathering and model behaviour analysis. The implementation of such a component is currently left open, but we envision algorithms heavily based on machine learning.

Statistics Gathering

Traditionally, models are not exposed to simulation kernel details due to the wrong level of abstraction. Simulation models only care about being simulated, and not about how this is being done. This is different for a new simulation kernel component that has to monitor the behaviour of not only the model, but the simulator as well.

We add performance metrics in the simulation kernel, which logs relevant performance metrics and processes them for use in other components. These metrics include the number of events created and destroyed, the number of inter- and intra-kernel events, the number of rollbacks, the measured lookahead, details of the Global Virtual Time (GVT) and Earliest Output Time (EOT) calculations, and information on the fairness between simulation kernels. With all these metrics, a component can get a fair view on both model and simulation kernel behaviour.

For example, if the actually seen lookahead is significantly higher than the defined lookahead, it might be interesting to switch to optimistic synchronization. When the number of rollbacks is excessively high, switching to conservative synchronization might be considered.

Visualization of Communication To provide more insight in our benchmark models, we created a simple visualization of their simulation trace. This trace visualizes the allocation of the model and all defined connections. For each connection, the number of events transferred is annotated. Examples are shown for the three benchmark models used before: Figure 16, Figure 17, and Figure 18 show traces for the Queue, Interconnect, and PHOLD models respectively. Using this information, we notice that the Interconnect benchmark indeed has a lot of inter-kernel events. Despite the similar structure, the PHOLD model does not have as many inter-kernel events. These results are relevant information that can be used by the hotswapping component.

Model Allocation

Although the synchronization protocol is one of the defining factors in simulation performance, model allocation has a significant impact on which protocol is ideal. Depending on the model structure, and how it is mapped to the different cores, it might not even

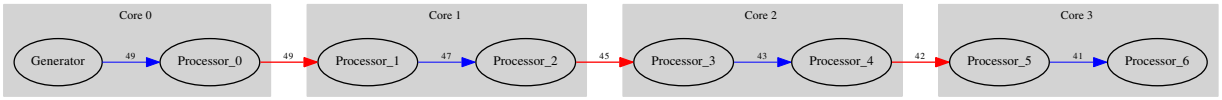

Figure 16. Queue model simulation trace across 4 kernels.

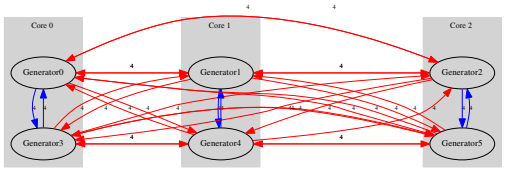

Figure 17. Interconnect simulation trace for 6 models on 3 kernels.

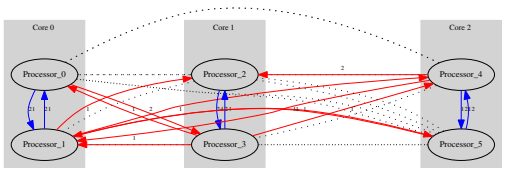

Figure 18. Phold simulation trace for parallel simulation using three kernels.

make sense to parallelize at all. Indeed, if the model is distributed such that frequent communication is necessary between cores, parallelism is naturally reduced. This brings us to the topic of model allocation, as also implemented in *PythonPDEVS* [26].

The modeller can specify which kernel a model should be allocated to, should such manual intervention be required. This is handled by the default model allocator. If no preference is given, a simple striping scheme is used but this is often insufficient. By overriding the default allocator, a modeller tunes the allocation scheme for a specific model, maximizing parallel speedup. This interface can be linked to graph algorithms for automatic allocation scheme generation, for example to avoid cycles in the dependency graph.

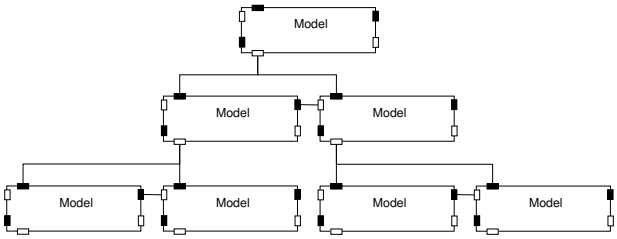

Figure 19. PHOLDTree model for depth 1 and width 2.

Performance Evaluation

To evaluate the influence of model allocation, we define a new model, based on PHOLD [24]. The model structure resembles a tree: an atomic model can have a set of children, with children being connected to each other recursively.

Unlike the Queue model, the width of the hierarchy is still present in the topology of the atomic models after direct connection. The PHOLDTree model allows us to investigate parallel speedup in terms of model allocation, by modifying the depth and width (fanout) model parameters.

The PHOLDTree model is similar in structure to models of gossiping in social networks [27]. The lookahead of an atomic node is the minimally allowed ϵ, indicating uncertainty, as is often the case in realistic models. We demonstrate the importance of allocation by comparing performance of a breadth-first versus a depth-first scheme. Both options are automated ways of allocation that are independent of the model.

PHoldtree, like Queue, is a highly hierarchical model but one where the flattened structure cannot be partitioned into a chain, as was the case in the Queue model. This topology is interesting since it highlights the effects of allocation. First, we evaluate the model in sequential simulation to provide a baseline for parallel simulation.

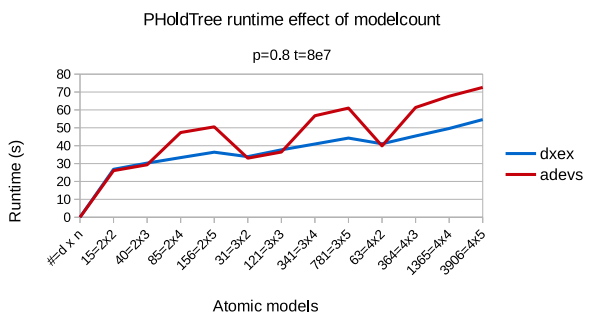

Figure 20. Effect of hierarchy in sequential simulation.

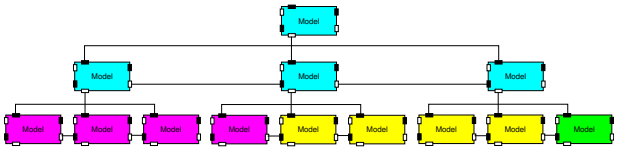

Figure 21. PHOLDTree model breadth first allocation with 4 kernels.

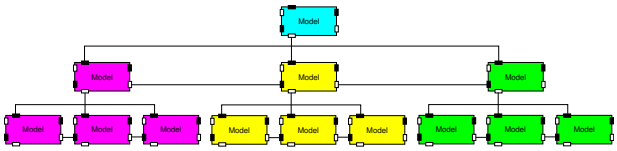

Figure 22. PHOLDTree model depth first allocation with 4 kernels.

Sequential Simulation Since *adevs* does not use direct connection, we expect a noticable performance difference between *dxex* and *adevs*. This is shown in Figure 20, where the fanout (n) determines the performance penalty *adevs* suffers compared to *dxex*. Profiling indeed indicates that an increase in width per subtree (n) leads to higher overheads in *adevs* due to the lack of direct connection. *Dxex* uses direct connection, making it independent on fanout. Performance of *dxex* is, in this model, only dependent on the number of models. Slight deviations can still be seen, though, caused by the initialization overhead of direct connection. Both *adevs* and *dxex* scale linearly in the number of atomic models.

Parallel Simulation Next, we run the model using two different model allocation schemes: breadth-first and depth-first. But first, we explain what we mean by both allocation schemes.

With breadth-first allocation, we traverse the tree in a breadth-first way, allocating subsequently visited atomic models to the same node. This means, intuitively, that atomic models at the same level in the tree, but not necessarily siblings, are frequently allocated to the same node. Since there is only infrequently some communication between siblings, and even never between different subtrees, this does not sound an intuitive allocation. This allocation strategy is shown in Figure 21.

With depth-first allocation, we traverse the tree in a depth-first way, allocating subsequently visited atomic models to the same node. This means, intuitively, that subtreees are frequently allocated to the same node. This allocation strategy is shown in Figure 22.

Both allocators will try to divide models evenly over kernels. The effects of varying the number of models per kernel are already evaluated in the previous section on scaling. Here we want to highlight the overhead of communication and inter-kernel dependence.

The breadth first allocation scheme results in a dependency chain with multiple branches, much like in the Queue model. Such a linear dependency chain can result in a parallel speedup as we demonstrated with the Queue model. This is not always true though: a single kernel with an unbalanced number of atomic models or unequal computational load in transition functions slows down the remainder of the chain. This effect is also apparent if the thread a kernel runs on is not fairly scheduled. With conservative synchronization this leads to excessive polling of the EOT of the other kernels. With optimistic synchronization this leads to a cascade of rollbacks, since dependent kernels will simulate ahead of the slower kernel.

After simulation the traces can be visualized for both breadth-first and depth-first allocation. Using a breadth-first allocation scheme, as shown in Figure 23, we notice that many events get exchanged between cores. This is caused by the high number of inter-core connections and the high number of events exchanged over these connections. The number of connections between nodes at the same simulation core is also rather low. Using a depth-first allocation scheme, however, as shown

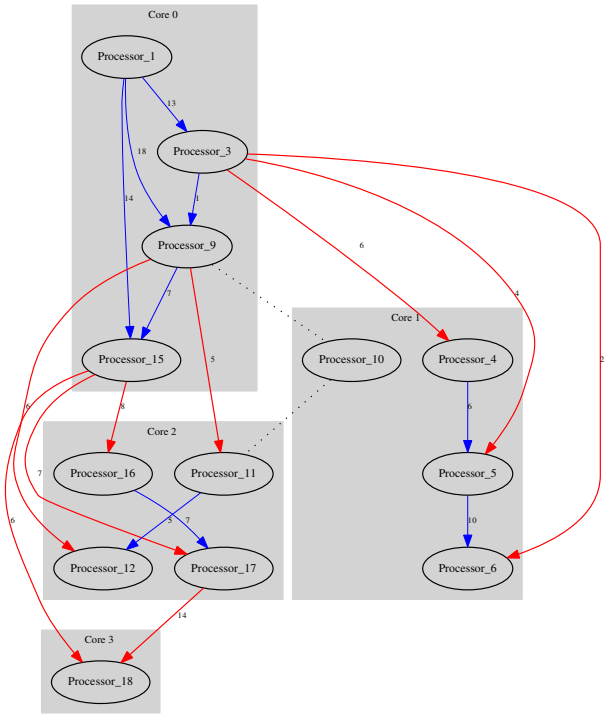

Figure 23. Visualization of a PHOLDTree simulation with breadth first allocation and 4 kernels.

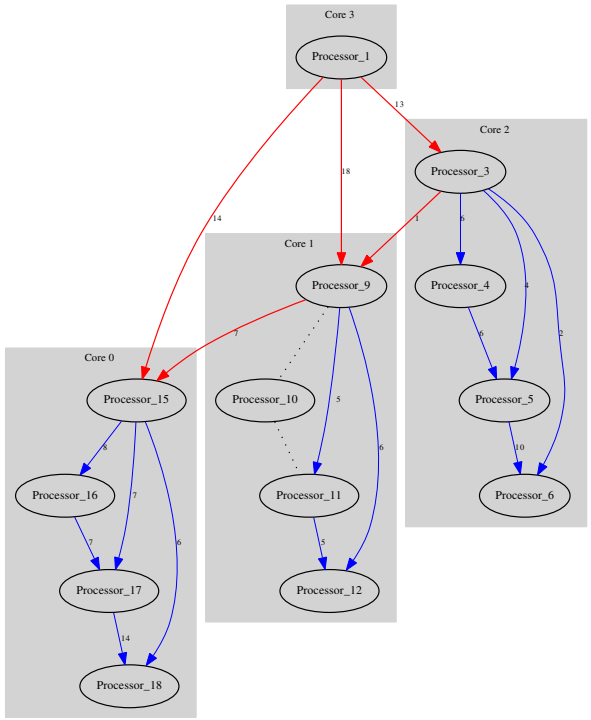

Figure 24. Visualization of a PHOLDTree simulation with depth first allocation and 4 kernels.

in Figure 24, minimizes inter-core connections while maximizing intra-core connections.

Simulation results are shown in Figure 25 for both allocation schemes in combination with both synchronization protocols. We see that for both synchronization protocols, the depth-first allocation is significantly better than breadth-first allocation. This is what we expected for this model: depth-first allocation maintains locality better than breadth-first allocation. Whereas this is the case in this example, this is not true in general, as the ideal allocation depends on the model being simulated.

The most prominent aspect of these results is the low performance for conservative depth-first allocation for two cores. This is mostly caused by the difference between a sequential simulation and a parallel simulation: suddenly we need to take into account synchronization and passing around of lookahead values. And since the number of cores is low, the overhead dominates any further speedup. Optimistic is less sensitive to the number of models per core as it does not need to poll each model for a lookahead, this explains the lower runtime penalty observed for optimistic.

Interestingly, we see that optimistic synchronization is less influenced by the allocation than conservative synchronization. This is likely caused due to the lower number of connections to take into account in conservative synchronization. Whereas conservative synchronization needs to take into account even scarcely used connections, optimistic synchronization does not. The same is true in the opposite direction, though, where optimistic synchronization is slower when a good allocation is chosen. Conservative synchronization will then be able to make better estimates, whereas optimistic synchronization does not make estimations.

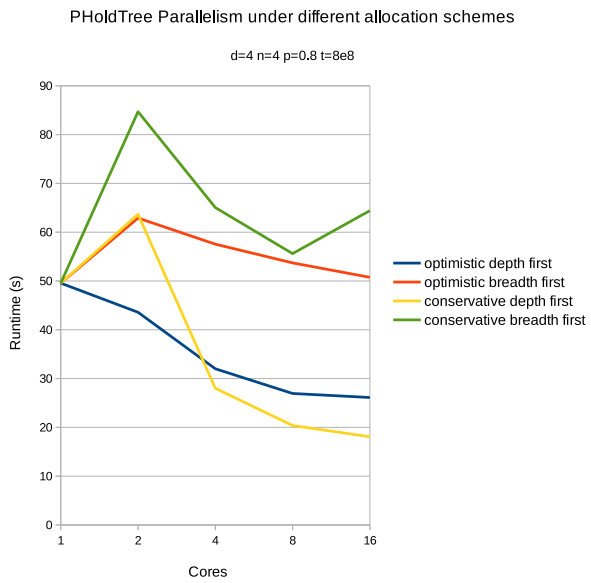

Figure 25. PHOLDTree model performance using different allocation schemes.

Related Work

Several similar DEVS simulation tools have already been implemented, though they differ in several key aspects. We discuss several dimensions of related work, as we try to compromise between different tools.

In terms of code design and philosophy, *dxex* is most related to *PythonPDEVS* [11]. Performance of *PythonPDEVS* was still decent through the use of "hints" from the modeler. In this spirit, we offer users the possibility to choose between different synchronization protocols. This allows users to choose the most appropriate synchronization protocol, depending on the model. Contrary to *PythonPDEVS*, however, *dxex* doesn't support distributed simulation [15], model migrations [26], or activity hints [28].

Although *PythonPDEVS* offers very fast turnaround cycles, simulation performance was easily outdone by compiled simulation kernels. In terms of performance, *adevs* [9] offered much faster simulation, at the cost of a significant compilation time. The turnaround cycle in *adevs* is much slower though, specifically because the complete simulation kernel is implemented using templates in header files. As a result, the complete simulation kernel has to be compiled again every time. Similarly to *vle* [29] and *PowerDEVS* [30], *dxex* compromises by separating the simulation kernel into a shared library. After the initial compilation of the simulation tool, only the model has to be compiled and linked to the shared library. This significantly shortens the turnaround cycle, while still offering good performance. In terms of performance, *dxex* is shown to be competitive with *adevs*. Despite its high performance, *adevs* does not support optimistic synchronization, which we have shown to be highly relevant.

Previous DEVS simulation tools have already implemented multiple synchronization protocols, though none have done it in a strictly modular way that allows straightforward protocol switching for a single given model. For example, *adevs* only supports conservative synchronization, and *vle* only supports experiment-level parallelism (*i.e.*, run multiple experiments concurrently). Closest to our support for multiple synchronization protocols is *CD++* [31]. For *CD++*, both a conservative (*CCD++* [32]) and optimistic (*PCD++*) [33]) variant exist. Despite the implementation of both protocols, they are different projects entirely. Some features might therefore be implemented in *CCD++* and not in *PCD++*, or vice versa. And while this might not yet be that significant a problem to this day, this problem will only get worse when each project follows its own course. *Dxex*, on the other hand, is a single project, where the choice of synchronization protocol is a simple configuration. *CD++*, however, implements both conservative and optimistic synchronization for distributed simulation, whereas we limit ourselves to parallel simulation. By limiting our approach to parallel simulation, we are able to achieve higher speedups through the use of shared memory communication.

In the PDES community, the problem of choosing between synchronization protocols is well known and documented [34]. The challenges of implementing such runtime switching have previously been explored already [35], and an implementation was given by, for example [36]. Our contribution entails bringing this same feature to the Parallel DEVS community, further expanding upon our support for multiple synchronization protocols.

Model allocation and its impact on parallel performance has previously also been studied in the PDES community [37]. Referenced as partitioning of the simulation model, most studies distinguish between communication overhead and computational distribution (load balancing) as the two dimensions to partition over. Partitioning a simulation model is identified as an issue to achieve scalability [38]. Some research in the context of Parallel DEVS has been done, where they turn their attention to load balancing and communication overhead [39]. Our contribution studies the effect of partitioning with emphasis on the effect of communication between processes and in the presence of a flattened hierarchy. We focus on static partitioning since this is a limiting factor for our conservative synchronization implementation which does not support model relocation. Model relocation, as implemented by *PythonPDEVS* [26], might be an interesting addition to only model allocation at the start of simulation.

In summary, *dxex* tries to find the middle ground between the concepts of *PythonPDEVS*, the performance of *adevs*, and the multiple synchronization protocols of *CD++*. To further profit from our multiple synchronization protocols in a single tool, we further added runtime switching between synchronization protocols and model allocation support.

Conclusions and Future Work

In this paper, we introduced *DEVS-Ex-Machina* ("*dxex*"), a new C++11-based Parallel DEVS simulation tool. Our main contribution is the implementation of multiple synchronization protocols for parallel multicore simulation. We have shown that there are indeed models which can be simulated significantly faster using either synchronization protocol. *Dxex* allows the user to choose between either conservative or optimistic synchronization at the start of simulation. Depending on observed model behaviour and simulation performance, runtime switching between synchronization protocols can be used.

Notwithstanding our modularity, *dxex* achieves performance competitive to *adevs*, another very efficient DEVS simulation tool. Performance is measured both in elapsed time, and memory usage. Our empirical analysis shows that allocation of models over kernels is critical to enable a parallel speedup. Furthermore we have shown when and why optimistic synchronization can outperform conservative and vice versa. Finally we investigated the effect of memory (de)allocation on parallel simulation.

Future work is possible in several directions. First, our implementation of optimistic synchronization should be more tolerant to low-memory situations. In its current state, simulation will simply halt with an out-of-memory error. The use of transactional memory can offer several advantages in this project. If it becomes part of the new C++17 standard it would be of great interest to see if it can help reduce the performance effects of memory allocation and synchronization. Having simulation control, which can throttle the speed of nodes that use up too much memory, has been shown to work in these situations [5]. Faster GVT implementations [40, 41] might further help to minimize this problem. Second, the runtime switching between synchronization protocols can be driven using machine learning techniques. The simulation engine is already capable of collecting data to inform such a process, and is designed to listen for commands from an external component. Third, automatic allocation might be possible by analysis of the connections between models. This information is already used in *dxex* to construct the dependency graph in conservative synchronization. A graph algorithm that distributes models, while avoiding cycles, could be used to offer a parallel speedup in either optimistic or conservative synchronization. Similarly, it could serve as a default parallel allocation scheme that can be improved by the user.

ACKNOWLEDGMENTS

This work was partly funded with a PhD fellowship grant from the Research Foundation - Flanders (FWO). Partial support by the Flanders Make strategic research centre for the manufacturing industry is also gratefully acknowledged.

References

[1] Zeigler BP, Praehofer H and Kim TG. *Theory of Modeling and Simulation*. 2nd ed. Academic Press, 2000.

[2] Vangheluwe H. DEVS as a common denominator for multi-formalism hybrid systems modelling. In

IEEE International Symposium on Computer-Aided Control System Design. pp. 129–134.

[3] Chow ACH and Zeigler BP. Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 1994 Winter Simulation Multiconference*. pp. 716–722.

[4] Himmelspach J and Uhrmacher AM. Sequential processing of PDEVS models. In *Proceedings of the 3rd European Modeling & Simulation Symposium*. pp. 239–244.

[5] Fujimoto RM. *Parallel and Distributed Simulation Systems*. 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 1999.

[6] Kim KH, Seong YR, Kim TG et al. Distributed simulation of hierarchical DEVS models: Hierarchical scheduling locally and time warp globally. *Transactions of the SCS* 1996; 13(3): 135–154.

[7] Jafer S and Wainer G. Conservative vs. optimistic parallel simulation of DEVS and Cell-DEVS: A comparative study. In *Proceedings of the Summer Computer Simulation Conference*. pp. 342–349.

[8] Cardoen B, Manhaeve S, Tuijn T et al. Performance analysis of a PDEVS simulator supporting multiple synchronization protocols. In *Proceedings of the 2016 Symposium on Theory of Modeling and Simulation - DEVS*. pp. 614 – 621.

[9] Nutaro JJ. adevs. **`http://www.ornl.gov/~1qn/adevs/`**, 2015.

[10] Jefferson DR. Virtual time. *ACM Transactions on Programming Languages and Systems* 1985; 7(3): 404–425.

[11] Van Tendeloo Y and Vangheluwe H. The modular architecture of the Python(P)DEVS simulation kernel. In *Proceedings of the 2014 Spring Simulation Multiconference*. pp. 387–392.

[12] Van Tendeloo Y. *Activity-aware DEVS simulation*. Master's Thesis, University of Antwerp, Antwerp, Belgium, 2014.

[13] Chen B and Vangheluwe H. Symbolic flattening of DEVS models. In *Proceedings of the 2010 Summer Simulation Multiconference*. pp. 209–218.

[14] Barros FJ. Modeling formalisms for dynamic structure systems. *ACM Transactions on Modeling and Computer Simulation* 1997; 7: 501–515.

[15] Van Tendeloo Y and Vangheluwe H. An overview of PythonPDEVS. In RED CW (ed.) *JDF 2016 – Les Journées DEVS Francophones – Théorie et Applications*. pp. 59 – 66.

[16] Mattern F. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing* 1993; 18(4): 423–434.

[17] Drepper U. What every programmer should know about memory. https://lwn.net/Articles/250967/, 2007.

[18] Cleary S and Bristow A P. Boost Pool: Fast memory pool allocation. **`http://www.boost.org/doc/libs/1_61_0/libs/pool/doc/html/`**, 2011.

[19] Ghemawat S and Menage P. TCMalloc: Thread-caching malloc. **`http://goog-perftools.sourceforge.net/doc/tcmalloc.html`**, 2005.

[20] L'Ecuyer P. Random numbers for simulation. *Communications of the ACM* 1990; 33(10): 85–97.

[21] Bauke H and Mertens S. Random numbers for large-scale distributed Monte Carlo simulations. *Physical Review E* 2007; 75(6): 066701:1–066701:14.

[22] Wainer G, Glinsky E and Gutierrez-Alcaraz M. Studying performance of DEVS modeling and simulation environments using the DEVStone benchmark. *SIMULATION* 2011; 87(7): 555–580.

[23] Glinsky E and Wainer G. DEVStone: a benchmarking technique for studying performance of DEVS modeling and simulation environments. In *Proceedings of the 9th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*. pp. 265–272.

[24] Fujimoto RM. Performance of Time Warp under synthetic workloads. In *Proceedings of the SCS Multiconference on Distributed Simulation*. pp. 23–28.

[25] Muzy A and Nutaro JJ. Algorithms for efficient implementations of the DEVS & DSDEVS abstract simulators. In *1st Open International Conference on Modeling and Simulation (OICMS)*. pp. 273–279.

[26] Van Tendeloo Y and Vangheluwe H. PythonPDEVS: a distributed Parallel DEVS simulator. In *Proceedings of the 2015 Spring Simulation Multiconference*. pp. 844–851.

[27] Jelasity M. *Gossip*. Springer Berlin Heidelberg, 2011. pp. 139–162.

[28] Van Tendeloo Y and Vangheluwe H. Activity in PythonPDEVS. In *Proceedings of the workshop on Activity-based Modeling and Simulation*. pp. 2:1–2:10.

[29] Quesnel G, Duboz R, Ramat E et al. VLE: a multi-modeling and simulation environment. In *Proceedings of the Summer Simulation Multiconference*. pp. 367–374.

[30] Bergero F and Kofman E. PowerDEVS: a tool for hybrid system modeling and real-time simulation. *Simulation* 2011; 87(1-2): 113–132.

[31] Wainer G. CD++: a toolkit to develop DEVS models. *Software: Practice and Experience* 2002; 32(13): 1261–1306.

[32] Jafer S and Wainer G. Flattened conservative parallel simulator for DEVS and Cell-DEVS. In *Proceedings of International Conferences on Computational Science and Engineering*. pp. 443–448.

[33] Troccoli A and Wainer G. Implementing Parallel Cell-DEVS. In *Proceedings of the Spring Simulation Symposium*. pp. 273–280.

[34] Jha V and Bagrodia RL. A unified framework for conservative and optimistic distributed simulation. In *Proceedings of the eighth workshop on Parallel and distributed simulation*. pp. 12–19.

[35] Das SR. Adaptive protocols for parallel discrete event simulation. In *Proceedings of the Winter Simulation Conference*. pp. 186–193.

[36] Perumalla KS. μsik-a micro-kernel for parallel/distributed simulation systems. In *Workshop on Principles of Advanced and Distributed Simulation*. pp. 59–68.

[37] Bahulkar K, Wang J, Abu-Ghazaleh N et al. Partitioning on dynamic behavior for parallel discrete event simulation. In *Proceedings of the ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*. pp. 221–230.

[38] Nicol DM. Scalability, locality, partitioning and synchronization PDES. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*. pp. 5–11.

[39] Himmelspach J, Ewald R, Leye S et al. Parallel and distributed simulation of Parallel DEVS models. In *Proceedings of the 2007 Spring Simulation Multiconference - Volume 2*. pp. 249–256.

[40] Fujimoto RM and Hybinette M. Computing Global Virtual Time in shared-memory multiprocessors. *ACM Transactions on Modeling and Computer Simulation* 1997; 7(4): 425–446.

[41] Bauer D, Yaun G, Carothers CD et al. Seven-O'Clock: A new distributed GVT algorithm using network atomic operations. In *Proceedings of the Workshop on Principles of Advanced and Distributed Simulation*. pp. 39–48.