Journal Title
XX(X):1–30

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

Performance analysis of a PDEVS simulator supporting multiple synchronization protocols

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 the 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, but based on PythonPDEVS, which supports both conservative and optimistic synchronization protocols with the capability to switch during simulation between those protocols. We evaluate the performance gain obtained by choosing the most appropriate synchronization protocol and demonstrate the influence model allocation and uncertainty has on this choice. Performance results are compared to adevs, in terms of CPU time and memory usage.

[1]University of Antwerp, Belgium

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

Prepared using sagej.cls *[Version: 2015/06/09 v1.01]*

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. And while Parallel DEVS [3] was introduced to increase parallelism, this is often insufficient. Several synchronization protocols from the discrete event simulation community [4] have been applied to DEVS simulation. 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. The applicability of parallel simulation capabilities of current tools is therefore limited.

This paper introduces DEVS-Ex-Machina* ("dxex"), our simulation tool 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 should merely determine, which protocol they wish to use. Users who simulate a wide variety of models, with different ideal synchronization protocols, can simply run the same model with different synchronization protocols. We investigate in this paper how model allocation and uncertainty determine the choice between synchronization protocols. The synchronization overhead is demonstrated by reducing the computational load of a model to near zero.

Our tool is based on PythonPDEVS, but implemented in C++11 for increased performance, using features from the new C++14 standard when possible. Unlike PythonPDEVS dxex only supports multicore parallelism.

We implemented a model that, depending on a single parameter, changes its ideal synchronization protocol. We demonstrate using several models the factors influencing the performance under a given synhronization 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. To verify that our flexibility does not counter performance, we compare to adevs, currently one of the fastest DEVS simulation tools available [5, 6].

Dxex offers visualization of the simulation and in depth statistics. A modeller can then make a more informed decision on which synchronization protocol to use or even intervene during simulation and request a switch between protocols.

The remainder of this paper is organized as follows: Section 2 introduces the necessary background on synchronization protocols. Section 3 elaborates on our design that enables this flexibility. In Section 4, we evaluate performance of our tool by

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

comparing its different synchronization protocols, and by comparing to adevs. Related work is discussed in Section 5. Section 6 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* [4]. 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 [4].

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 (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 not always a trivial task if there is no detailed knowledge of the model. Even slight changes in the model can change the lookahead, and can therefore have a significant influence on simulation performance.

Optimistic Synchronization

A completely different synchronization protocol is *optimistic synchronization* [7]. 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.

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), 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: all 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.

DEVS-Ex-Machina

Historically, dxex is based on PythonPDEVS [5]. Python is a good language for prototypes, but performance has proven insufficient to compete with other simulation kernels [8]. Dxex is a C++11-based implementation of PythonPDEVS, but implements only a subset of PythonPDEVS, while making some of its own additions. So while the feature set is not too comparable, the architectural design, core simulation algorithm, 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 [9], Dynamic Structure DEVS [10], termination conditions, and a modular tracing and scheduling framework [5]. 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. An uninformed choice on which one to implement is insufficient, as performance will likely be bad. 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 `AtomicModel`s connected to it. The superclass `Core` is merely the sequential simulation core, but can be used as-is. 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 written out in shared memory, where it can be read out by all other nodes.

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 might know of optimizations specific to atomic variables or constant expressions which are heavily used in dxex.

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 [11] 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.

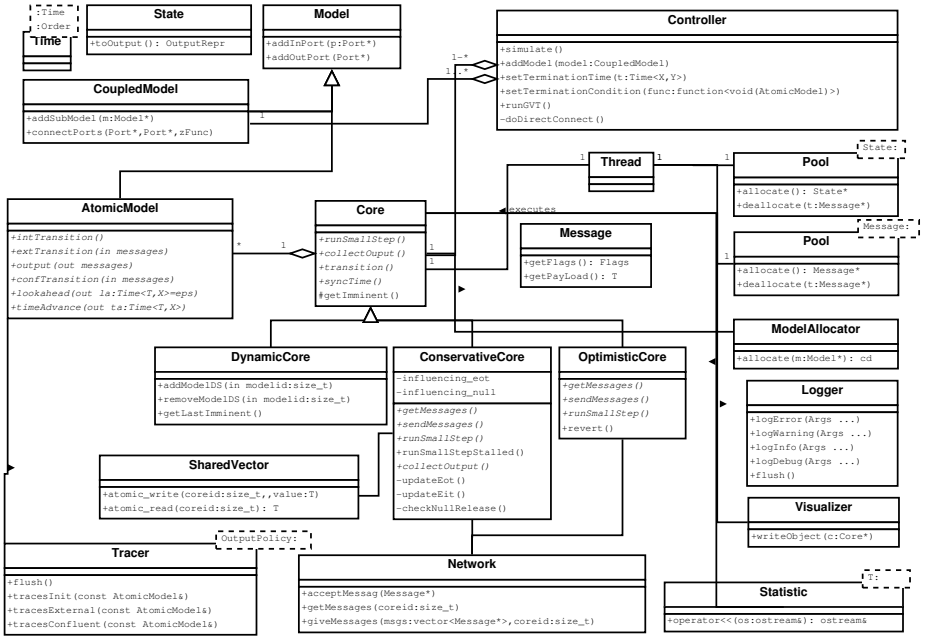

Figure 1. Dxex design.

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 were 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 either using sequential execution, using conservative synchronization, or using optimistic synchronization. Switching between simulation kernels is as simple as altering the simulation termination time. The exception is conservative synchronization, where a lookahead function 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 if users will never use conservative synchronization. Especially since defining the lookahead is often non-trivial and dependent on intricate model details. The more attractive option is for the simulation tool to provide a default lookahead function, such that simulation can

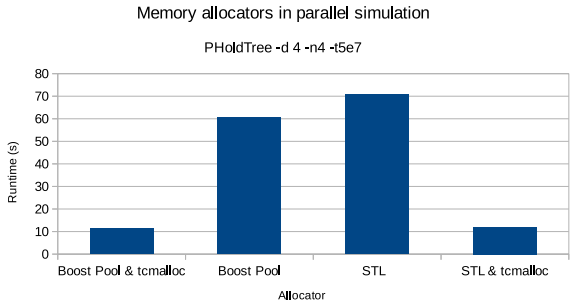

Figure 2. Effect of memory allocators on parallel execution time.

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 $epsilon$ (*i.e.*, the smallest representable timestep) is used otherwise.

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 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 [12]: 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 a sequential simulation will run correctly in a conservative or optimistic simulation without altering (from the point of view of the model author) 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 will persist 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 [13] 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* [14], which reduces lock contention in `malloc` calls.

We primarily investigate this for optimistic simulation, as this is the most memory consuming mode of simulation [**?**]. 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 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 [**?**]. 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) [15] as an alternative random number generator with performance and multithreading in mind. Since the RNG is an implicit part of the state in the 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 gets three times faster by using TRNG, than when 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 gets much slower when the RNG becomes part of the model state, since the state needs to be copied as well. This becomes a significant overhead

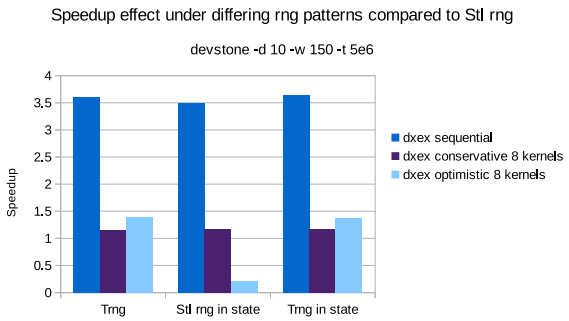

Figure 3. Speedup with different RNG usage patterns compared to Stl rng.

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, currently one of the most efficient simulation kernels [6], 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.

Benchmarks

We use four different benchmarks, which cover different aspects of the simulation kernel:

1. The *Queue* model, based on the *HI* model of DEVStone [16], creates a chain of hierarchically nested atomic DEVS models. A single generator pushes events into the queue, which are consumed by the processors after a fixed or random delay. It takes two parameters: width and depth, which determine 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 2 is shown in Figure 4.

2. The *PHOLD* model, presented by [17], 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 to an atomic model at the same node. Afterwards, a uniform selection is made between the remaining ports. 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 and when the average time advance is larger than the lookahead in conservative simulation. An example for four models, split over two nodes, is shown in Figure 6.
3. The *Interconnect* model, a merge of PHOLD [17] and the *HI* model of DEVStone [16], 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 investigates the complexity of event routing, and how the simulation kernel handles many simultaneous events. An example for four models is shown in Figure 5.

Sequential Simulation

Queue For the first benchmark, we tested the effect of hierarchical complexity of the model in the performance 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 symbolically flattens the model, there is no performance hit when the depth is increased. The overhead of running the directconnect algorithm is one time only and negligible when the end time of the simulation is sufficiently large. Adevs on the other hand does suffer from the increased depth. 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, thus quadratically increasing the number of couplings and the number of external transitions. As can be seen in Figure 8, adevs 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 and deallocation semantics and tracing are solved by the kernels at a runtime cost in simulations where event frequency is very high. Profiling the benchmarks clearly shows the increasing cost of output generation and deallocation as the determining factor in

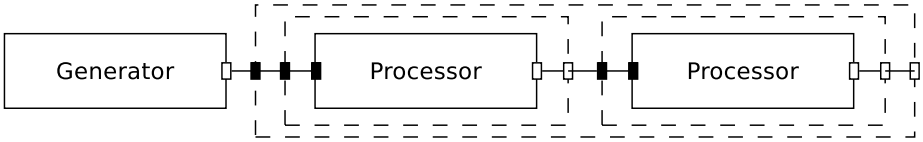

Figure 4. Queue model for depth and width 2.

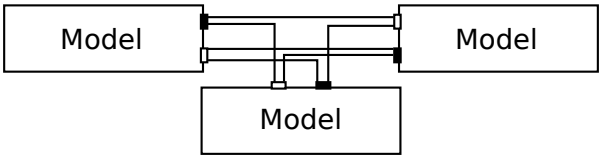

Figure 5. Interconnect model for three models.

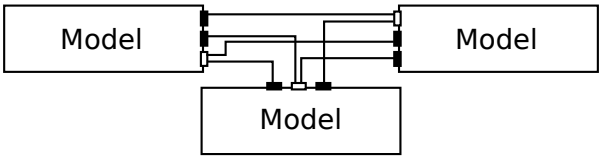

Figure 6. PHOLD model for three models.

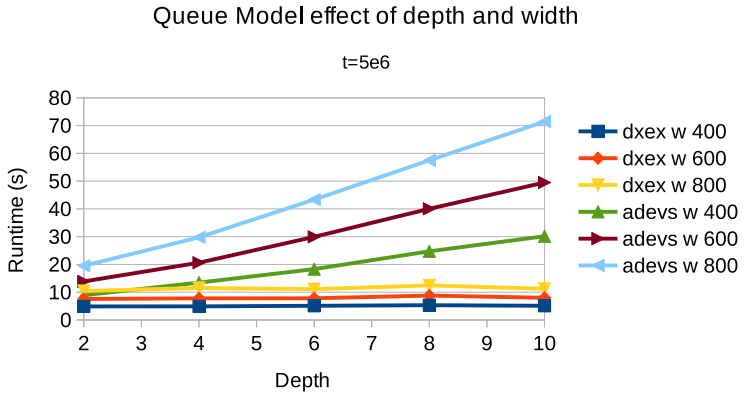

Figure 7. Queue benchmark results for sequential simulation.

the gap in performance. We refer the interested reader to the dxex repository for the profiling call graphs for the different benchmarks.

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

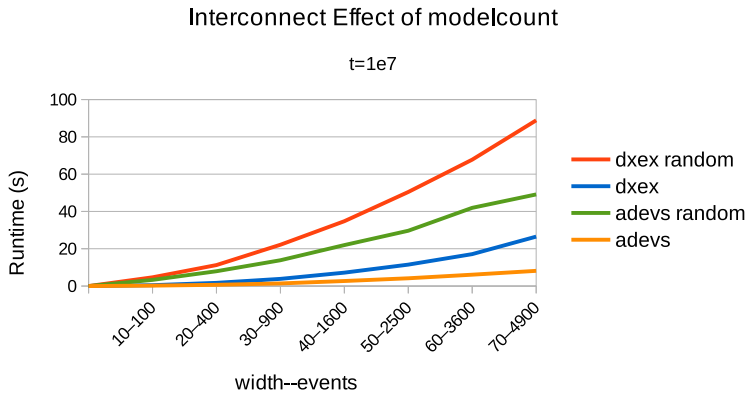

Figure 8. Interconnect benchmark results for sequential simulation.

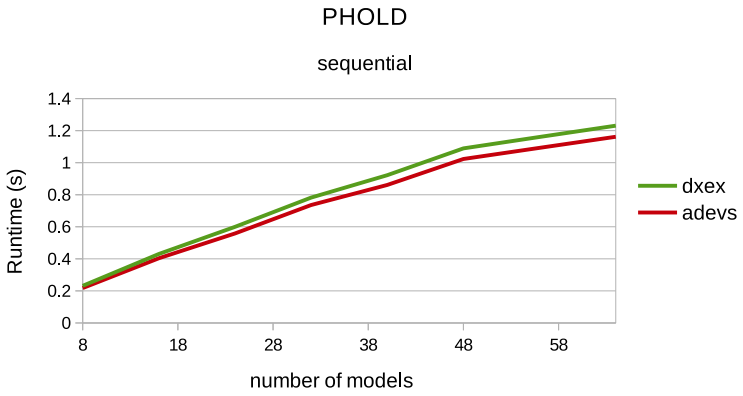

Figure 9. PHold benchmark results for sequential simulation.

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

Queue

Allocation The Queue model is one single chain of models. Each kernel gets one connected part of this chain. The result is that the kernels themselves also form a chain where events only travel in one direction.

Strong and Weak Scaling Figure 11 shows the speedup compared to dxex sequential for a fixed problem size. As the amount of kernels increases, the optimistic

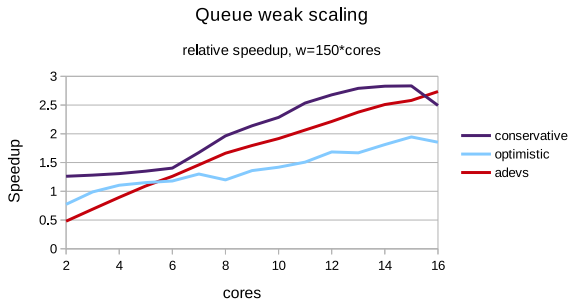

Figure 10. Queue model weak scaling speedup compared to dxex sequential.

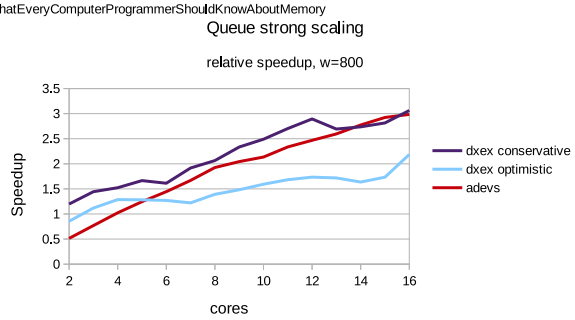

Figure 11. Queue model strong scaling speedup compared to dxex sequential.

kernel quickly becomes the worst choice. The difference between dxex conservative and adevs becoming smaller. The same effect can be seen for weak scaling in Figure 10.

The speedup of adevs is always in comparison with the runtime of the corresponding dxex sequential benchmark.

Interconnect In the Interconnect model, we determine how broadcast communication is supported across multiple nodes. The number of models is now kept 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 an increasing number of 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. In Interconnect there is no allocation scheme possible that avoids cyclic dependencies between simulation kernels, as shown in the trace 17 of a simulation with 4 models. Such a cycle forces sequential operation of the kernels with no speedup possible.

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

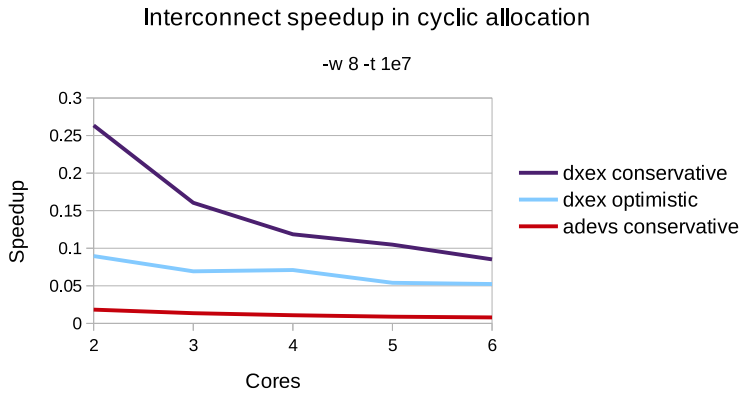

Figure 12. Interconnect benchmark results for parallel simulation.

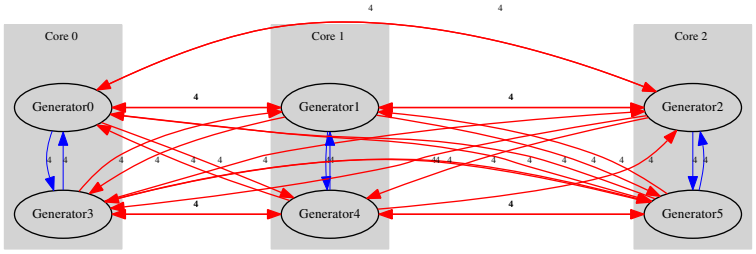

Figure 13. Interconnect parallel simulation trace for 6 models on 3 kernels.

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 in contrast to PythonPDEVS and dxex's optimistic kernels which handle this inside the kernel. We feel this would lead to an unfair comparison as we would like to keep the models synchronization-agnostic 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,

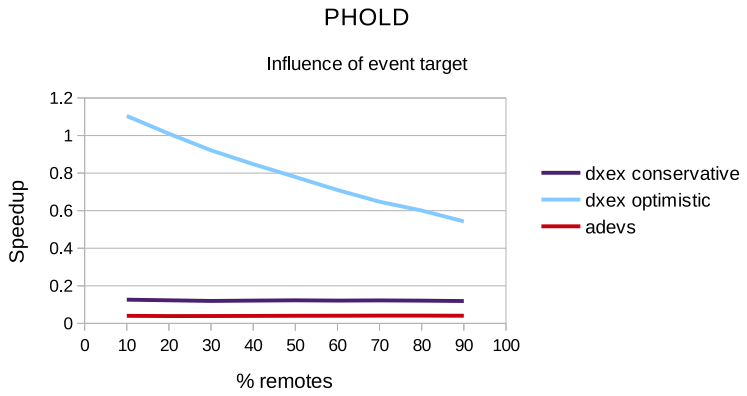

Figure 14. Phold benchmark results for parallel simulation using four kernels, four atomics per node, with varying percentage of remote events.

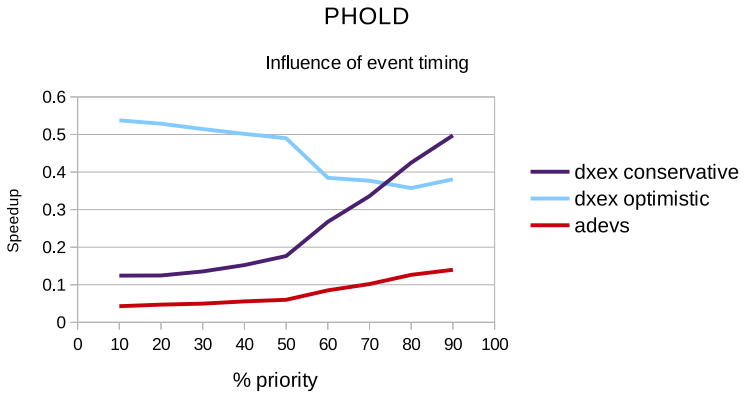

Figure 15. Phold benchmark results for parallel simulation using four kernels, with varying amount of high-priority events.

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 case made in the comparison between both synchronization algorithms by [4]. In Figure 18 it is clear that in Phold it is possible for dependency cycles to form between kernels which as we have shown in Interconnect degrades performance for both optimistic and conservative. This is also the cause of the sublinear speedup observed in our Phold benchmark.

Figure 15 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 mindless 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.

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 [8], 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.

Hotswapping the Synchronization Protocol

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 has been frequently shown, 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 might be 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 using the current synchronization core. When stopped, each kernel instantiates a new core 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 single second.

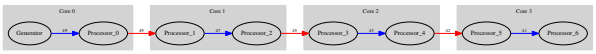

Figure 16. Queue model (d=2, w=7, t=5000, random timeadvance) allocation and simulation trace across 4 kernels.

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 thus augmented with the necessary bindings for such a component to be defined. 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: algorithms can be heavily based on machine learning or similar approaches.

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 basic information on the fairness between different 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 very high, conservative synchronization might be ideal. And when neither of these two is an option, the simulation might just (temporarily) have to fall back to sequential simulation.

For performance reasons, statistics gathering is optional due to the imposed overhead.

Visualization of Communication To provide some more insight in the models we used as benchmarks previously, we created a simple visualization of the 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: Figures 16, 17, and 18 shows traces for the Queue, Interconnect, and PHOLD models respectively.

Naturally, results similar to this are relevant information that can be used by the hotswapping component.

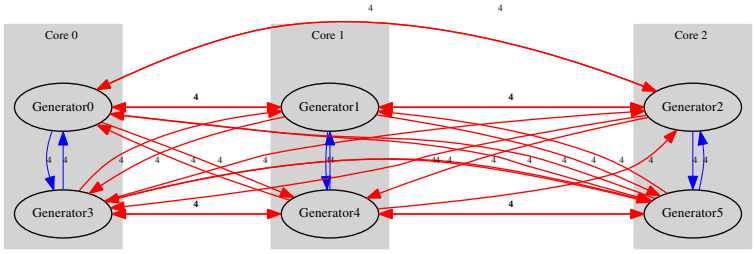

Figure 17. Interconnect parallel simulation trace for 6 models on 3 kernels.

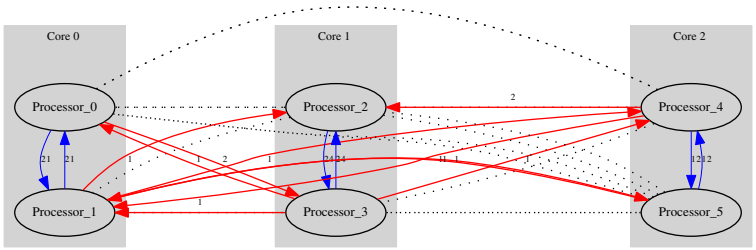

Figure 18. Phold benchmark trace for parallel simulation using three kernels.

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, and even whether or not parallelization would make sense. Indeed, if the model is distributed in such a way that frequent communication is necessary between cores, parallelism is naturally reduced. This thus brings us to the topic of model allocation. Model allocation was one of the features also implemented by PythonPDEVS [18].

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 followed but this is not sufficient in most simulations to achieve a speedup in parallel. By overriding the default allocator a modeller tunes the allocation scheme for a specific model, maximizing parallel speedup. This interface can be used to employ graph algorithms for an automatic allocation scheme, for example avoiding cycles in the dependency graph.

Performance Evaluation

In order to evaluate the influence of model allocation, we define a new model, based on PHOLD [17]. The model structure resembles a tree: an atomic model can have a set of children, with children being connected to each other as well. Connections can be uni- or bidirectional.

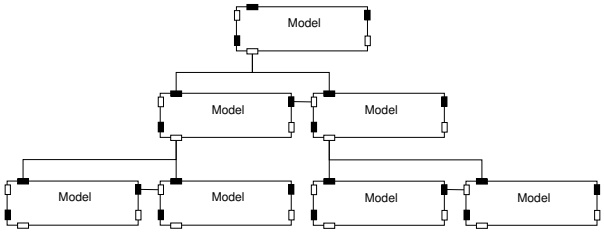

Figure 19. PHOLDTree model for depth 1 and width 2.

Unlike the Queue model, the width of the hierarchy is still present in the topology of the atomic models after the direct connection stage. 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 [**?**], hierarchical resource sharing [**?**] or load balancing [**?**]. 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 a breadth-first versus a depth-first scheme.

PHoldtree, like Queue, is a highly hierarchical model but one where the flattened structure cannot be partitioned into a chain as in Queue. This topology is interesting since it highlights the effects of allocation. This model allows us to investigate in depth the effects of non-cyclic allocation strategies and measure parallel speedup. First, we evaluate the model in sequential simulation to provide a baseline for parallel simulation.

Sequential Simulation Since adevs does not use direct connection, we expect a noticable performance difference between dxex and adevs. This is shown in Figure **??**, where the fanout determines the performance penalty adevs suffers compared to dxex. Profiling indeed indicated 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, but only of 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. We expect to see a similar distinction in parallel simulation.

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 This allocation strategy is shown in Figure **??**.

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

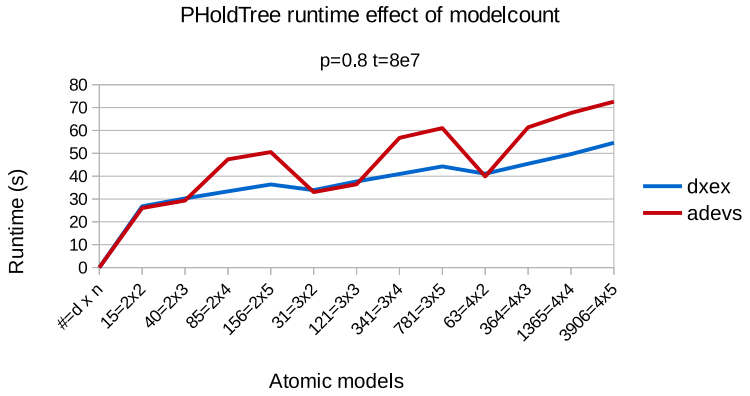

Figure 20. Effect of hierarchy in sequential simulation.

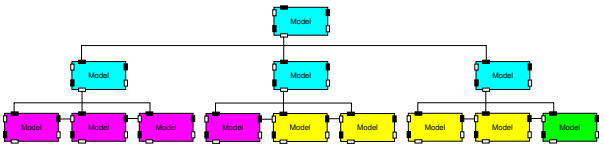

Figure 21. PholdTree model breadth first allocation with 4 kernels.

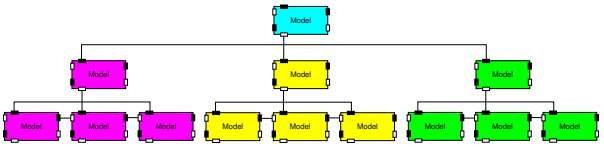

Figure 22. PholdTree model depth first allocation with 4 kernels.

subtreees are frequently allocated to the same node. This allocation strategy is shown in Figure **??**.

The breadth first allocation scheme will result in kernels that will form 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, but this is not always true as we will demonstrate in this section. A single kernel that has an unbalanced number of atomic models or unequal computation load in transition functions will slow down the remainder of the chain. This effect is also apparent if the thread a kernel runs on is not fairly scheduled. In conservative this will lead to excessive polling of the other kernels' eot values, in optimistic this will lead to a cascade of reverts since dependent kernels will simulate ahead of the slower kernel. In Figures 23 and 24 the simulation trace is visualized for both allocation schemes highlighting the remarks made in this paragraph.

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

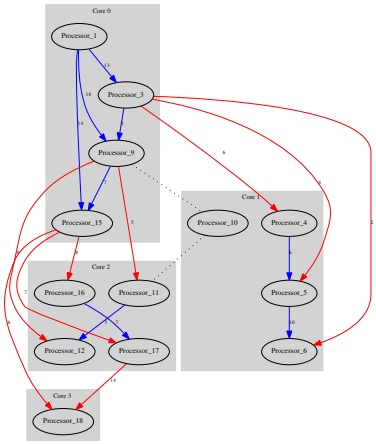

Figure 23. Visualization of PholdTree (d=1,n=3,t=5000) parallel simulation with breadth first allocation and 4 kernels.

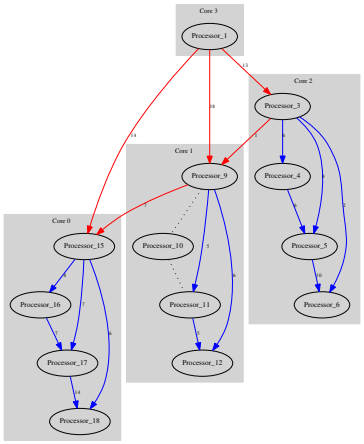

Figure 24. Visualization of PholdTree (d=1,n=3,t=5000) parallel simulation with depth first allocation and 4 kernels.

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 in Figure 24,

Simulation results are shown in Figure **??** 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: depth-first allocation maintains locality better than breadth-first allocation.

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.

Strong Scaling In Figure 26 we see that the difference in performance for all 4 kernel configurations compared to that shown in Figure 25 is a constant factor. The probability of the priority message has no extra impact on performance other than that shown in sequential performance.

The probability of a high priority event in this model does not affect the performance difference between conservative and optimistic. The key parameter quickly becomes the load of a kernel in models. Our conservative implementation in an uncertain simulation is very sensitive to a high load in models, whereas optimistic has no such limitation.

The difference between depth first and breadth first allocated kernels is striking, the first results in sublinear speedup for both synchronization protocols. The breadth first allocation scheme will lead to a very high number of inter-kernel connections, which is detrimental for any parallel synchronization algorithm. An event exchanged between kernels cannot be securely deallocated without a GVT algorithm and all the complexity this entails. Even a fast asynchronous GVT algorithm will require some form of inter kernel synchronization and span a timeframe during which allocated memory cannot be reused, forcing new allocations. Similar to the Queue benchmark the breadth first allocation scheme leads to a topology for the kernels resembling a chain but with more branches in the chain. In Queue there is only a single model on the edge of a kernel exchanging messages to a single other model in a neighbouring kernel, this is not the case in the PholdTree under breadth first allocation. This explains the difference in speedup between Queue and PholdTree despite the similarity in kernel topology.
Depth first allocation still has a higher inter kernel connection count, but not on the same order as breadth first allocated PholdTree. Depth first allocation will converge to a star topology which in this benchmark leads to a significant speedup.
The performance drop observed for 2 kernels for all allocation schemes and synchronization protocols is in part due to the higher link count between kernels. With more kernels these connections will be spread across more kernels and result in a relative lower performance penalty.
Conservative synchronization suffers a further penalty in this benchmark when there are only a few kernels. The PholdTree model has a lookahead of ϵ leading the kernel to query each allocated atomic model for a lookahead value on each simulation round. As the atomic models are spread over more kernels this effect is reduced leading to an increase in performance. With 4 kernels conservative surpasses optimistic in speedup. This is clear evidence that conservative synchronization is a good candidate for parallel simulation even in simulations with uncertainty.

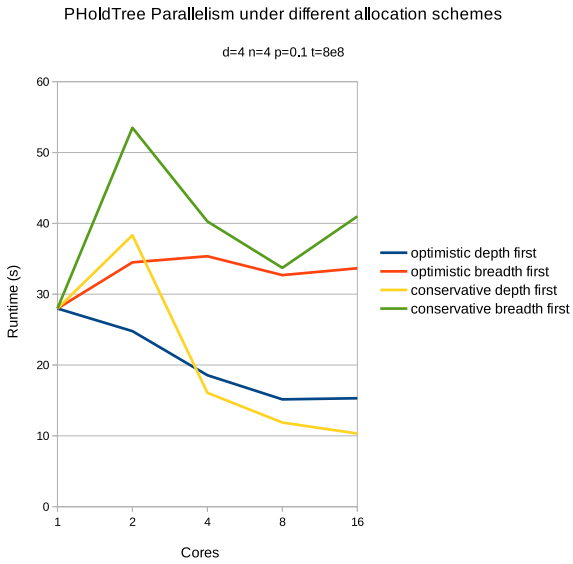

Figure 25. PholdTree model performance under different allocation schemes with low message probability

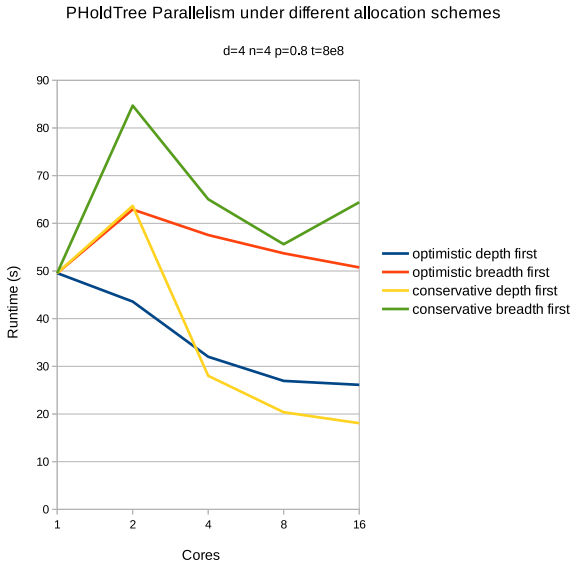

Figure 26. PholdTree model performance under different allocation schemes with high message probability

Weak Scaling In this paragraph we investigate dxex's parallel performance when the the number of atomic models distributed over a fixed number of kernels varies. From **??** we know that increasing d will lead to a very rapid increase in models. We want to observe what happens in a closer to linear increase in model count per kernel by varying n.

Adevs' conservative implementation cannot handle the uncertainty (lack of non ϵ lookahead) here and is omitted from the comparison. Its parallel performance is two orders of magnitude slower than the sequential implementation. Upon investigation we conclude that this slowdown is caused by not implementing the {begin/end}lookahead() functions, which when not overridden in a model trigger an exception on each invocation. This exception handling completely stalls the kernel. We do not implement this function in our PholdTree variant for adevs since this would force us to implement state saving in the model code and not use the provided state saving functionality in the kernel, as is done in dxex or PythonPDEVS's optimistic. In dxex state saving is never required for a conservative simulation, this is handled transparently for the user who need not implement this behaviour. By using the state saving technique inside the model code we feel we would no longer compare identical models across different synchronization algorithms, although we do not doubt adevs' increased performance when these functions are overridden.

In Figure 27 we observe that dxex's optimistic kernels are slightly sensitive to the number of atomic models allocated to them. In the worst case with 2 kernels there is no real speedup observable, as soon as the number of kernels increases we see a converging speedup trend, that except for 32 kernels is almost constant despite the increase in n. Note that only depth first allocated kernels are measured here, breadth first as we can see in 26 has no speedup advantage. The probability parameter is kept at 0.1 for the same reason, it only induces a linear increase in load.

Conservative has a more nuanced speedup behaviour in this benchmark. We have detailed how a conservative kernel in dxex scales linear in the number of atomic models (if lookahead is ϵ), this effect is more clearly visible here.
It is clear that as the fanout (n) of the model increases the kernel performance degrades rapidly. The intersection of the performance graph of each configuration is shifted to the right, the actual 1-speedup point increases as the kernelcount increases. For a d=4, n=5 configuration with 4 kernels each kernel has a load of 1300 atomic models when it crosses the 1-speedup line.
An 8 kernel configuration with d=4 n=6 can sustain a load of 2700 atomic models before it reaches that point. From the results we see that the 32-kernel configuration is not yet slowing down with the current parameters.
The modelcount is only a part of the explanation of this effect, as the kernelcount increases the amount of inter kernel messages will be split into more distinct sets which can be handled with more concurrency in dxex's architecture. That this effect is not always true can be observed from the d=4 n=3 32-kernel datapoint. Note that in this configuration 341 atomic models are distributed over 32 kernels, a relative low model load can more easily highlight synchronization overhead, even if allocation is optimal.
As with optimistic we do not include breadth first allocated benchmarks in this speedup plot since none of those achieve a speedup higher than 1.

If we combine both to determine which synchronization protocol is more optimal we see in Figure 29 that for these configurations conservative with 8 kernels is an ideal configuration up until n=5, where the modelcount starts to degrade performance. Optimistic with the same kernelcount is almost insensitive to the increase in modelcount and is therefore a more robust choice. Note how optimistic and conservative with 32 kernels at this point still have a reasonable speedup which is surprising given the synchronization overhead in such a large set of kernels.

Linking Back

Related Work

Several similar DEVS simulation tools have already been implemented, though they differ in 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 [5]. Performance of PythonPDEVS was still decent, through the use of simulation and activity hints from the modeler. This allowed the kernel to optimize its internal data structures and algorithms for the specific model being executed. All changes were completely transparent to the model, and were completely optional. 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, model migrations [18], or activity hints [19].

While PythonPDEVS offers very fast turnaround cycles, due to the use of an interpreted language, simulation performance was easily outdone by compiled simulation kernels. In terms of performance, adevs [20] 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. Dxex compromises, as vle [21] or PowerDEVS [22], 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 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 CD++ [23] has both a conservative (CCD++ [24]) and optimistic (PCD++) [25]) variant. Despite the implementation of both protocols, they are different projects entirely, and are incompatible with modern compilers. Dxex, on the other hand, is a single project, where switching between different synchronization protocols is as simple as switching any other configuration parameter. CD++, however, implements both conservative and optimistic synchronization for distributed simulation, whereas we limit ourselves to

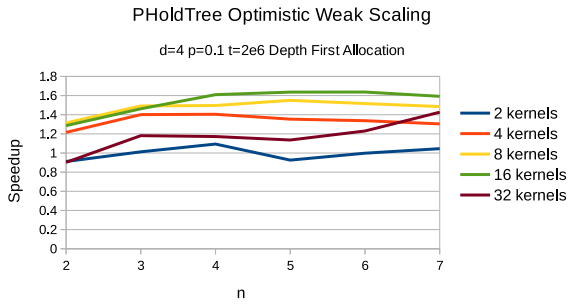

Figure 27. PholdTree model weak scaling under varying fanout and optimistic synchronization

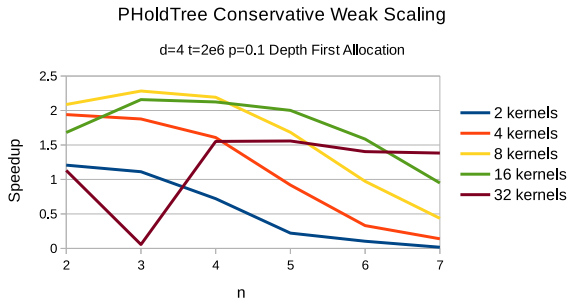

Figure 28. PholdTree model weak scaling under varying fanout and conservative synchronization

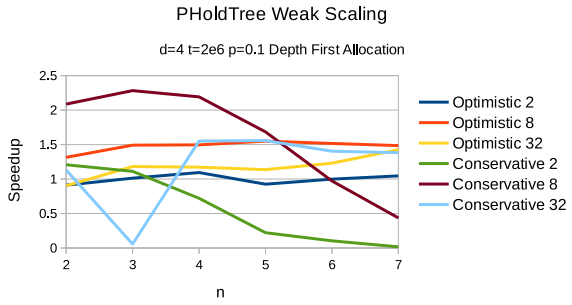

Figure 29. PholdTree model weak scaling under varying fanout and different synchronization algorithms

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 [26]. In the work of [27] the challenges facing

the implementation of runtime switching between protocols is explored. A PDES implementation of such a feature was presented in [28]. Our contribution entails bringing this feature to the PDEVS community.

Our emphasis on model allocation as a key factor in parallel performance has been studied in detail by the PDES community [29]. 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 in the work of [30] to achieve scalability. In [31] partitioning is studied in a PDEVS context with attention to load balancing and communication overhead. 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.

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++.

Conclusions and future work

In this paper, we introduced DEVS-Ex-Machina ("dxex"), a new C++14-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 as simple as any other configuration option. Notwithstanding this 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.

Future work is possible in several directions. Firstly, 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. Having simulation control, which can throttle the speed of nodes that use up too much memory, has been shown to work in these situations [4]. Faster GVT implementations, such as those presented by [32] and [33], might further help to minimize this problem. Secondly, 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. Thirdly, activity algorithms, as already implemented by PythonPDEVS, can also be implemented in dxex, to determine how they influence simulation performance. Finally automatic allocation is 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 in the resulting kernel topology could be used to offer

a parallel speedup in either optimistic or conservative synchronization, or 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).

References

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

[2] Vangheluwe H. DEVS as a common denominator for multi-formalism hybrid systems modelling. *CACSD Conference Proceedings IEEE International Symposium on Computer-Aided Control System Design* 2000; : 129–134.

[3] Chow ACH and Zeigler BP. Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 26th Winter Simulation Conference*. SCS. ISBN 0-7803-2109-X, pp. 716–722. URL **http://dl.acm.org/citation.cfm?id=193201.194336**.

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

[5] Van Tendeloo Y and Vangheluwe H. The Modular Architecture of the Python(P)DEVS Simulation Kernel. In *Spring Simulation Multi-Conference*. SCS, pp. 387 – 392.

[6] Franceschini R, Bisgambiglia PA, Touraille L et al. A survey of modelling and simulation software frameworks using Discrete Event System Specification. In *2014 Imperial College Computing Student Workshop*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 40–49.

[7] Jefferson DR. Virtual time. *ACM Transactions on Programming Languages and Systems* 1985; 7(3): 404–425. DOI:10.1145/3916.3988. URL **http://doi.acm.org/10.1145/3916.3988**.

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

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

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

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

[12] Drepper U. What every programmer should know about memory 2007; : 1–112.

[13] 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/**, 2001-2005, 2011.

[14] Ghemawat S and Menage P. TCMalloc : Thread-Caching Malloc. **http://goog-perftools.sourceforge.net/doc/tcmalloc.html**, 2005.

[15] Bauke H and Mertens S. Random numbers for large-scale distributed monte carlo simulations. *Phys Rev E* 2007; 75: 066701. DOI:10.1103/PhysRevE.75.066701. URL **http://link.aps.org/doi/10.1103/PhysRevE.75.066701**.

[16] Glinsky E and Wainer G. DEVStone: a benchmarking technique for studying performance of DEVS modeling and simulation environments. In *Proceedings of the 2005 9th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*. pp. 265–272.

[17] Fujimoto RM. Performance of Time Warp under synthetic workkloads. In *Proceedings of the SCS Multiconference on Distributed Simulation*.

[18] Van Tendeloo Y and Vangheluwe H. PythonPDEVS: a distributed Parallel DEVS simulator. In *Proceedings of the 2015 Spring Simulation Multiconference*. SpringSim '15, Society for Computer Simulation International, pp. 844–851.

[19] Van Tendeloo Y and Vangheluwe H. Activity in PythonPDEVS. In *Activity-Based Modeling and Simulation*.

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

[21] Quesnel G, Duboz R, Ramat E et al. VLE: a multimodeling and simulation environment. In *Proceedings of the 2007 Summer Simulation Multiconference*. pp. 367–374.

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

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

[24] 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.

[25] Troccoli A and Wainer G. Implementing Parallel Cell-DEVS. In *Proceedings of the 2003 Spring Simulation Symposium*. pp. 273–280.

[26] Jha V and Bagrodia RL. A unified framework for conservative and optimistic distributed simulation. *SIGSIM Simul Dig* 1994; 24(1): 12–19. DOI:10.1145/195291.182480. URL **http://doi.acm.org/10.1145/195291.182480**.

[27] Das SR. Adaptive protocols for parallel discrete event simulation. In *Proceedings of the 28th Conference on Winter Simulation*. WSC '96, Washington, DC, USA: IEEE Computer Society. ISBN 0-7803-3383-7, pp. 186–193. DOI:10.1145/256562.256602. URL **http://dx.doi.org/10.1145/256562.256602**.

[28] Perumalla KS. μsik-a micro-kernel for parallel/distributed simulation systems. In *Workshop on Principles of Advanced and Distributed Simulation (PADS'05)*. IEEE, pp. 59–68.

[29] Bahulkar K, Wang J, Abu-Ghazaleh N et al. Partitioning on dynamic behavior for parallel discrete event simulation. In *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*. PADS '12, Washington, DC, USA: IEEE Computer Society. ISBN 978-0-7695-4714-5, pp. 221–230. DOI:10.1109/PADS.2012.32. URL **http://dx.doi.org/10.1109/PADS.2012.32**.

[30] Nicol DM. Scalability, locality, partitioning and synchronization pdes. In *Proceedings of the Twelfth Workshop on Parallel and Distributed Simulation*. PADS '98, Washington, DC, USA: IEEE Computer Society. ISBN 0-8186-8457-7, pp. 5–11. DOI:10.1145/278008.278010. URL **http://dx.doi.org/10.1145/278008.278010**.

[31] 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*. SpringSim '07, San Diego, CA, USA: Society for Computer Simulation International. ISBN 1-56555-313-6, pp. 249–256. URL **http://dl.acm.org/citation.cfm?id=1404680.1404720**.

[32] 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. DOI:10.1145/268403.268404. URL **http://doi.acm.org/10.1145/268403.268404**.

[33] Bauer D, Yaun G, Carothers CD et al. Seven-o'clock: A new distributed GVT algorithm using network atomic operations. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*. PADS '05, Washington, DC, USA: IEEE Computer Society. ISBN 0-7695-2383-8, pp. 39–48. DOI:10.1109/PADS.2005.27. URL **http://dx.doi.org/10.1109/PADS.2005.27**.