Performance analysis of a parallel PDEVS simulator handling both conservative and optimistic protocols

Ben Cardoen† **Stijn Manhaeve**† **Tim Tuijn**†
{firstname.lastname}@student.uantwerpen.be

Yentl Van Tendeloo† **Kurt Vanmechelen**†
Hans Vangheluwe†‡ **Jan Broeckhove**†
{firstname.lastname}@uantwerpen.be

† University of Antwerp, Belgium
‡ McGill University, Canada

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 algorithms have been proposed, each with a specific kind of simulation model in mind. Due to the significant differences between these algorithms, current Parallel DEVS simulation tools restrict themself to only one such algorithm. In this paper, we present a Parallel DEVS simulator, grafted on C++11, which offers both conservative and optimistic simulation. We evaluate the performance gain that can be obtained by choosing the most appropriate synchronization protocol. Our implementation is compared to adevs using hardware-level profiling on a spectrum of benchmarks.

1. INTRODUCTION

DEVS [18] 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 [17]. A number of tools have been constructed by academia and industry that allow the modelling and simulation of DEVS models.

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 still insufficient. Several synchronization algorithms from the discrete event simulation community [7] have been applied to DEVS simulation in particular. While several parallel DEVS simulation kernels exist, they often restrict themselves to a single synchronization protocol. This is a logical choice, as these synchronization protocols have nearly no commonalities. But the exact reason for different synchronization protocols, is that their distinct nature makes them very applicable to specific situations. As such, current DEVS simulation kernels always allow parallel simulation, but good performance can only be expected from some models.

Users that wish to simulate a set of distinct models, with different synchronization protocols, are therefore out of luck: Either they accept the lower performance for some of the models, or they use two distinct simulation kernels. Neither of these options is acceptable for the simulation of complex models, as the performance can become arbitrary worse, and the simulation kernels can diverge significantly.

In this paper, we introduce 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 transparant to the simulated model. Users should merely determine, at the start of simulation, which protocol they wish to use. Our tool is implemented in C++11, to increase both performance and portability across different platforms.

A simple model is used to prove that the ideal synchronization algorithm is indeed dependent on model details. In our case, a single parameter of the model can have a significant impact on simulation performance, and especially on which is the ideal synchronization protocol. In order to show that this flexibility does not counter our performance, we compare our tool to Adevs, currently one of the fastest DEVS simulation tools available [16, 5].

We introduce the necessary background for this paper in Section 2. Section 3 elaborates on our features and design. In Section 4, we evaluate performance of our tool by comparing its different synchronization protocols. Related work is discussed in Section 5. Section 6 concludes the paper and gives future work.

2. BACKGROUND

This section briefly introduces two distinct synchronization protocols, as used by dxex. Furthermore, we make note of several new features of C++11.

2.1 Conservative Synchronization

The first synchronization protocol that we will introduce, is *conservative synchronization* [7]. In conservative synchronization, a node is allowed to progress in simulated time, in-

SpringSim-TMS/DEVS 2016 April 3-6, Pasadena, CA, USA

dependent of all other nodes, up to the point where it can guarantee that no causality errors can happen. When this point in time is reached, the node has to block until it is allowed to progress any further. In practice, this means that all nodes need to be aware of the current simulation time of all other nodes, and the time it takes an event to propagate (called *lookahead*). Several algorithms are defined in the literature to implement this behaviour. An overview is given in [7].

Deadlock can occur when a dependency cycle occurs and the amount of exchanged messages is low. Again, multiple algorithms are defined to handle this situation. Algorithms range from deadlock avoidance to deadlock recovery.

The main advantage of this approach is that it has a very low overhead in cases of high parallelism between different nodes. Each node can simulate in parallel, while sporadically notifying other nodes that they can progress even further. The disadvantage, however, is that the amount of parallelism is explicitly limited by the size of the lookahead. If a node can influence another (almost) instantaneously, no matter how rarely it occurs, the amount of parallelism is severely reduced. It is also up to the user to define the size of the lookahead, depending on how the model is known to behave. Slight changes in model behaviour can cause significant changes to the lookahead, and can therefore also have a significant influence on simulation performance.

2.2 Optimistic Synchronization

A completely different synchronization protocol is *optimistic synchronization* [**?**]. Whereas conservative synchronization would prevent causality errors at all costs, optimistic synchronization will allow them to happen, but correct them afterwards. Each node is allowed to progress in simulated time as much as possible, without taking note of the state of any other node. When an event arrives at a node, which is already further in simulated time, the node will have to roll back its state to right before the event would normally have to be processed. As the simulation time is now rolled back to before the event is processed, the event can simply be processed as if no causality error ever occured.

Rolling back the simulation time requires the node to store past model states, such that they can be restored later on. Furthermore, all incoming and outgoing events need to be stored as well. Incoming events need to be passed to the models again, when the correct simulation time has again been reached, and outgoing events need to be cancelled, as potentially a different series of output events would normally have been 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 could have significant repercussions on the complete simulation.

Further changes are required when the model performs unrecoverable transactions, such as I/O (*e.g.*, tracing, writing to file, printing output), allocating and deallocating memory, and so on. For this, lightweight synchronization algorithms are still required to determine the lower bound of all simulation times, through the computation of a *Global Virtual Time* (GVT).

The advantage of optimistic synchronization is that there is no blocking at all: a node can always progress as fast as possible in simulated time. Additionally, you do not suffer from a small lookahead if the small lookahead situation rarely occurs. If an (almost) instantaneous event occurs rarely, performance will only be impacted if it occurs. The main disadvantage is the unpredictable performance and arbitrary cost of rollbacks. If rollbacks occur rather frequently, simulation quickly becomes slower than sequential, as the overhead of state saving becomes significant. Apart from the overhead in CPU time, there is also an overhead in terms of memory, as all intermediate states will also have to be stored up to a point where it can be considered *irreversible*.

While optimistic synchronization does not explicitly depends on the lookahead, simulation performance is still bound by the lookahead implicitly.

2.3 C++11 Parallelism Features

C++11 offers a wide range of portable synchronization primitives in the Standard Library, whereas in earlier versions one had to resort to non-portable (C) implementations. More importantly, C++11 is the first version of the standard that actually defines a multi-threaded abstract machine memory model in the language. Our kernels use a wide range of threading primitives and atomic operations. As an example, eot/eit/nulltime are not exchanged as messages, but as reads/writes to atomic fields shared by all kernels. This avoids the otherwise unavoidable latency penalty by mixing simulation messages with synchronization messages. For an in-depth study, we refer to [4]. Most modern compilers support the full standard, allowing the kernels to be portable by default on any standard compliant platform.

3. FEATURES

3.1 Based on PyPDEVS

Dxex is based on PyPDEVS, and provides the following features:

1. Direct Connection
2. Dynamic Structured DEVS
3. Termination function: if specified, a termination function is applied to each model every simulation round to test whether the simulation can terminate. This feature is only available in single-threaded simulations.
4. The State/Message objects can have any payload type. Different allows different message types to be used within the same simulation.
5. Tracing: An asynchronous, thread safe and versatile tracing mechanism allows exact verification of the simulation.
6. Optimistic and Conservative synchronization

Furthermore, the implementation tries to adhere to the C++ principle that you don't pay performance-wise for what you don't use. For this reason, the support for a termination function for the multi-threaded kernel was abandoned, as it is non-trivial to implement and had an adverse impact on the

runtime, even when not in use. Another example is the state saving mechanism, which is only used for optimistic parallel simulation and has no performance impact in a conservative parallel simulation.
Our tracing implementation is not comparable to adevs's listener interface. To be usable in an optimistic simulation, the tracing of the simulation has to be reversible and only be committed at GVT points. Furthermore, the framework itself has to be thread safe and deterministic so that a simulation will always produce the exact same output.
The following features from PyPDEVS are not present

1. Activity tracking and relocation
2. Serialization: in this context this is the ability to save/load a complete simulation to disk, not the state saving mechanisms required for TimeWarp.
3. Interactive control
4. Distributed simulation

In dxex, the model allocation is realized by a derivable allocator object which the user can implement to arrange a more ideal (domain-specific) allocation. If this is omitted, a default (non-activity-aware) allocation stripes the models over the simulation kernels.
In addition, debugging tools such as a logger and a graph visualizer are included that can track activity with respect to the allocation for later inspection (but not are not online and as dynamic as is in PyPDEVS).

3.2 Different Synchronization protocols

For parallel executions, synchronization is required for PDEVS to prevent causality violations from happening or to recover from causality violations. Preventing causality violations (conservative) typically requires domain-specific information to compensate for the performance loss, while recovering from causality errors (optimistic) requires the calculation of a state to revert to.

Conservative

In case of conservative synchronization, any kernel will determine which kernels it is influenced by. This information is constructed from the incoming connections on all hosted models. The process is only 1 link deep, since an influenced kernel will in turn be blocked by others deeper in the graph.
A model should provide a lookahead function which returns, relative to the current time, the timespan during which the model cannot change state due to an external event. This information is collected for all models hosted on the kernel and the minimum time is set as the lookahead of that kernel.
The kernel will calculate its earliest output time and write this value in shared memory. The eit of the kernel is then set as the minimal eot of all influencing kernels.
For the garbage collection (of sent messages), the LBTS/GVT is calculated as $\min_{\forall i \in \text{influencors}}(\text{nulltime}[i]) - \epsilon$.

Optimistic

The optimistic kernel requires from the hosted model that copying the state is done carefully (avoid unnecessary copies and make a copies whenever necessary).
The kernels use Mattern's [12] GVT algorithm with a maximum of 2 rounds per iteration to determine a GVT. This process runs asynchronously from the simulation itself. Once found, the controlling thread informs all kernels of the new value, which they can use to execute garbage collection of old states and (anti)messages.
The user must only provide one implementation of a model that can be used for both synchronization protocols. A lookahead function is desired to accelerate the conservative protocol, but is not required. In the absence of a user supplied lookahead, the kernel assumes it cannot predict beyond its current time $t + \epsilon$, creating a lockstep simulation. The user is shielded from the multi-threaded aspect of the kernel.
From the user's perspective, the multi-threaded aspect of the kernel is not exposed.

3.3 Performance Improvements

We now discuss a number of bottlenecks that were discovered during the profiling of several benchmarks.

Heap

In dxex, events are always passed with pointers and thus avoid a possibly expensive copy of the payload caused by allocation and copying overhead. In highly connected models, this allocation cost can become prohibitively expensive, so to reduce that overhead we use thread_local memory pools for states and events, and, optionally, replace the system malloc with calls to tcmalloc [9]. In this way, allocating threads do not block each other. If desired, arena-pools are available for single-threaded simulation. A disadvantage is increased complexity in ownership semantics. The creating kernel is responsible for destruction, but this can only be guaranteed up to the GVT. Experiments with synchronized pools proved to be slower than implementations with the standard malloc/free.
Initially, dxes was using strings as identifiers as was also done in PyPDEVS. Profiling quickly indicated that this caused a real performance bottleneck. In C++, strings are heap allocated variable sized objects with an atomic reference count and not not immutable objects as in Python. Access of that reference count across threads turned out to be quite expensive, as are the calls to malloc/free the string implementation makes to create/destroy new objects or copy existing objects. Strings, however, are more intuitive to work with from a user's perspective, so, as a compromise, we allowed the user to reference models/ports by string name. Once the simulation starts however, all objects use integral identifiers. This also increased the usage of the constexpr feature of C++11 in, amongst others, time stamps and message headers.

Raw pointers

While an important C++11 feature in general, our initial usage of smart pointers for some types of objects was misplaced. Used across threads, the reference counting became very expensive, and the (de)allocation of memory caused significant contention between threads. So for the models and the kernels, dxex is still using smart pointers whereas for the messeages, a raw pointer to compacted memory is being used.

Locking

Locking between kernels uses mostly atomic operations and, occasionally, we can leverage memory orderings to only pay for synchronization when we need it. Messages, on the other hand, are exchanged via a shared set of queues each with a dedicated lock.
On a higher level, we avoid the sending of synchronization messages entirely by writing the time stamp directly into shared memory.
Sending of antimessages is fairly cheap in our implementation, since only the modified pointer to the original message is sent to the receiving kernel.

Schedulers
PyPDEVS has a wide range of schedulers to choose from with varying performance depending on the simulation type. Profiling showed that, the heap implementation used in adevs was faster than any of the schedulers we had tested before (in a C++ environment). Unlike most node based heaps, this scheduler uses a fixed size array where its heap is rebuilt or modified in place depending on the amount of items to update. Items are only updated, never removed.

4. PERFORMANCE

4.1 Sequential Simulation

In our benchmarks, we will consistently compare the performance of dxex with adevs. The implemented models for the each benchmark are functionally identical.
Dxex uses integer timestamps whereas adevs uses floating point time representation. Tests show there is little difference performance-wise (both types fit in a machine word).
The benchmarks have been compiled using the `-O3` option and link-time optimization, with all IO disabled. To compare the models in details, we have run the different benchmarks with different parameters.
The benchmarks have been run on a i7-2670QM, 16GiB RAM, Arch Linux (4.2.5 kernel) and G++ 5.2. Interpretation of the benchmark results is done using call-graphs generated by perf. A selection of those call-graphs is available in the code repository. For the runtime plots, perf is used in combination with R.

Devstone
The Devstone [10] benchmark is highly hierarchical. As can be seen in Figure 1, dxex is outperforming adevs considerably. This can be explained by the fact that, using Direct connection, the hierharchical structure is being flattened whereas, in the adevs case, considerable overhead is caused by traversing the coupled model's structure to pass events.

PHold
PHold [6] is a benchmark that focuses in particular on parallel execution; the sequential runtime is therefore only added as a baseline. A model selects the destination of a message at runtime using a state-saved RNG, which takes up a non-trivial amount of runtime. The results of the benchmark are visualized in Figure 2.

Interconnect
Interconnect [14] is a benchmark where all models broadcast, creating a complete graph in dependencies between models.

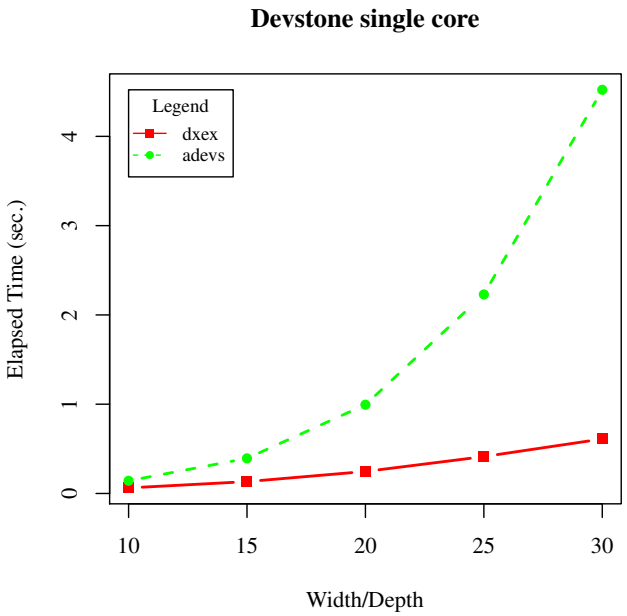

Figure 1. Devstone

As the model count increases, we see (see Figure 3 the expected quadratic increase in runtime for both adevs and dxex, but an increasing penalty for dxex. Profiling shows this is entirely due to the heap allocation of messages, which even though minimized by using memory pools remains significant.

4.2 Parallel Simulation

By default, the benchmarks use 4 kernels for parallel simulation.

Devstone
The flattened models are allocated to kernels by giving each kernel a distinct section of the chain, resulting in a low ratio of inter-kernel to intra-kernel messages. For the optimistic case, this can cause more reverts since the kernels will start to drift faster as the model-count increases. Furthermore, optimistic synchronization is quite sensitive to an increase in kernels, since the delay before a revert propagates, increases. This benchmark requires a specific warm-up time; for $n = d \times w$ models, it takes timeadvance$() * (n-1)$ transitions to activate the last model in the chain. When using parallel processing, this can be reduced to timeadvance$() * (n\frac{kernels-1}{kernels} - 1)$ before the last kernel becomes active.
Figure shows that dxex outperformance adevs (in both optimistic and conservative case).

PHold
In Phold, the allocation is specified in the benchmark itself. Each kernel manages a single node with a constant set of subnodes. The parameter R (which is here set to 10) determines the percentage of remote destination models.
The dynamic dependency graph is a very sparse version of the static dependency graph, penalizing the conservative case.

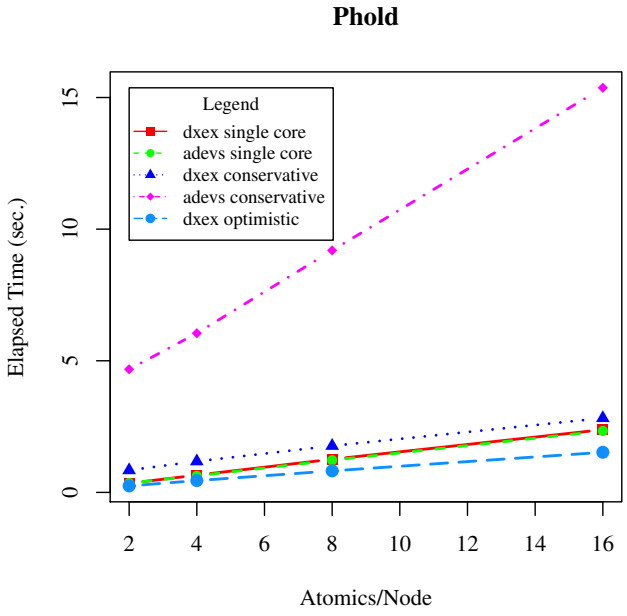

Figure 2. PHold

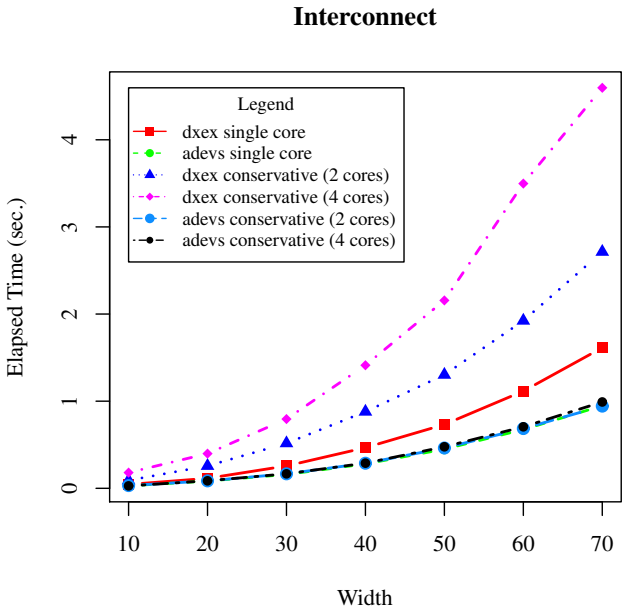

Figure 3. Interconnect

The lookahead is ϵ, so the conservative case spends most of its time crawling in steps of ϵ. Since the dependency graph between kernels is a complete graph, this is not a simulation that scales in our implementation. For N kernels, each kernel has to query the null-time of $N-1$ kernels, resulting in O(N^2) polling behaviour. In a non-cyclic simulation with a non-trivial lookahead (like in, e.g., Devstone), that choice does pay off (see Figure). Adevs's lesser performance is due in part to their lookahead management, which after profiling shows to spend a non-trivial amount of time in exception handling code.

The optimistic case suffers little from the above problems; due to the high interconnectivity, however, a cascading revert is still possible. With the percentage of remotes equal to 100, Phold reflects interconnect in behaviour, which is why the R value is not dimensioned here. A revert is very expensive in PHold due to our usage of C++11's random nr generators. The cost of a revert is dominated by the recalculation of destination models, not in allocating/deallocating states/messages. Once a revert happens the drift between kernels increases fast, increasing the likelihood of more reverts. Despite all this, optimistic can for low R values quickly exploit the uncertainty that slows down conservative in this benchmark.

Interconnect

In Interconnect, the set of atomic models form a complete graph (w.r.t. connections); each model broadcasts messages to the entire set.

Allocation cannot avoid cycles and the resulting dependency graph between kernels remains a complete graph. The runtime dependency graph is almost immediately identical to the static graph.

The conservative case still shows the same issues as in PHold, with the key difference , for a fixed time advance, the lookahead is equal to the timespan between transitions. The scaling issue is identical as with PHold.

In this benchmark, the optimistic case runs very quickly out of memory. With c kernels and N atomic models, a single revert undoing k transitions will lead to $(N-1) \times k$ messages that need to be recreated, plus $(N-1)\frac{c-1}{c} \times k$ anti-messages that need to be sent.

Priority network model

The priority benchmark is composed of a single server generating a stream of $0 \leq m \leq n$ messages at fixed time intervals, interleaved with a probability p for a priority message to n receivers (see Figure 5).

This defaults the lookahead for the receivers to ϵ, but this time there is no scaling effect, nor are there cycles in the dependency graph. This model therefore highlights the basic strengths/weaknesses of both synchronization protocols. Receiving models are allocated on another kernel than the server and have an internal transition so that they will not wait for the incoming messages.

A key difference here with the other benchmarks is that a state (in the Receiver instances) is very cheap to copy/create. The kernel holding the server will never revert since it is a source in the dependency graph. The optimistic case will therefore not suffer the same performance hit in recreating the states as it does in PHold. The overhead in the optimistic case is entirely due to the factor m, which will quickly dominate in increasing buffers of received messages. Interestingly, the parameter p does not clearly favour either synchronization protocol (see Figure). While this removes any possibility

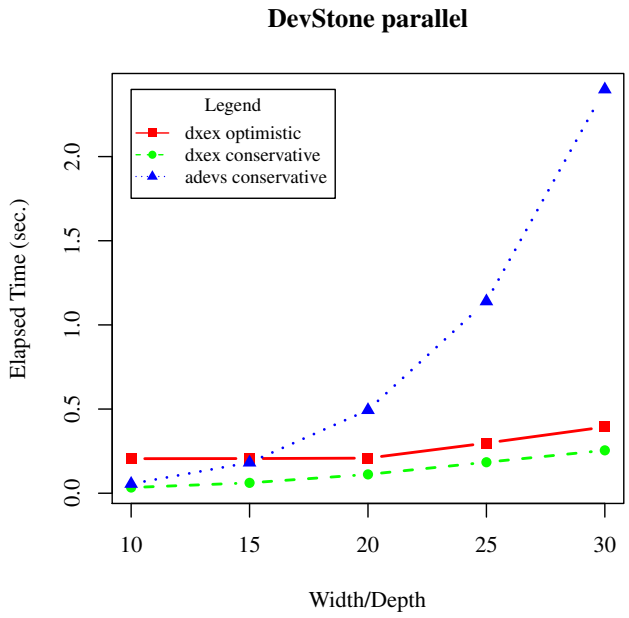

Figure 4. DevStone parallel

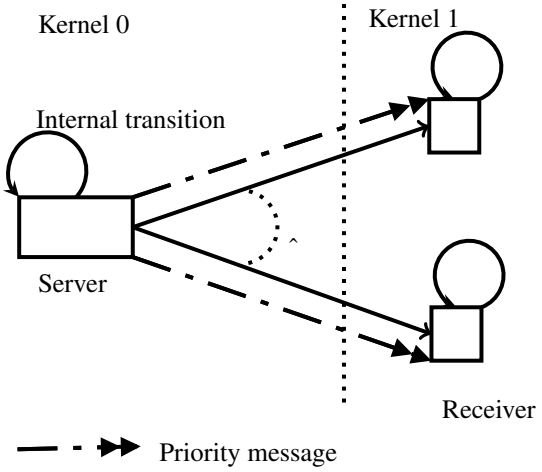

Figure 5. The priority network model

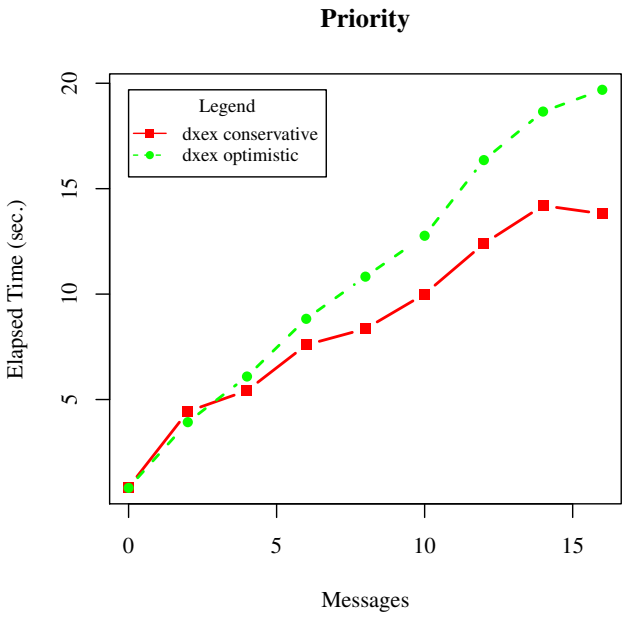

Figure 6. Priority

for a lookahead, the conservative case can quickly bridge the timespan between fixed messages since there is no cyclic dependency. Lightweight states prevent performance loss due to reverts in the optimistic case; only the overhead in event handling in the optimistic case eventually becomes the deciding factor.

4.3 Memory Usage

Platform and tools

Both dxex and adevs use tcmalloc as memory allocator. Additionally, dxex uses memory pools to further reduce the frequency of expensive system calls (malloc/free/sbrk/mmap/...). Tcmalloc will only gradually release memory back to the OS, whereas our pools will not do so at all. If memory has been allocated once, it is from a performance point of view better to keep that memory in the pool. This is one reason why memory utilization is best measured by peak allocation. Profiling is done using Valgrind's massif tool [13]. The platform used for memory profiling has an i5-3317U Intel CPU and 8GiB RAM with a page size of 4,096KiB, running Fedora 22 (kernel 4.2.6).

Measure

Adevs passes messages by value, dxex passes a pointer. The runtime effects of this choice have already been demonstrated in the Interconnect benchmark, so, in this section, we measure memory usage in number of allocated pages combining text, stack and heap memory for the program profiled. For the OS and/or user, this is the actual memory footprint of the application. It is important to note that, especially in the optimistic case, not all this memory is always in use by the kernels. During simulation, the pools will generally not return deallocated memory to the OS, but keep it for later reuse.

Results

Devstone

adevs	adevs con	dxex	dxex con	dxex opt
44	70	42	75	363

Table 1. Devstone 40x40 t5e5, unit MiB, 4 kernels (if parallel)

Since, in the conservative case, messages are passed by pa ointer, a GVT/LBTS implementation is required to organise the garbage collection. This inevitable delay explains the higher memory usage compared to adevs.
Optimistic's TimeWarp requires state/event saving, and its GVT algorithm is more complex (with a resulting higher latency) than the LBTS calculation in the conservative case. Moreover, the differences in LP virtual times are far larger compared to conservative time synchronization. All these factors explain the heavier memory usage. Devstone (flattened) is allocated in a chain. Leafs in the dependency graph will therefore do a lot of unnecessary simulation before having a revert, leading to an increased memory pressure. Unlike conservative and sequential execution, memory usage in the optimistic case varies greatly depending on scheduling of kernel threads and drifting between kernels.

PHold

adevs	adevs con	dxex	dxex con	dxex opt
40	x	37	61	682

Table 2. Phold n 4 s 16 t1e6 r 10, unit MiB, 4 kernels (if parallel)

With only 10% of all messages being inter-kernel, we expect conservative to have memory consumption near that of the single threaded implementation, since intra-kernel messages are reclaimable after each round. The counterintuitive high memory usage can be explained by conservative's stalled round behaviour which occurs whenever a kernel cannot advance (eit == time). In such a round messages are sent out but the kernel does not execute any transitions until it has received all input from all influencing kernels.
With lookahead ϵ this then leads to a high frequency of polling on shared null-times, which are used to determine lbts and thus garbage collection. The lbts calculation will not wait until a new value is found, since this can create unwanted contention with the simulation. The resulting longer time intervals within which no new lbts is found delay memory deallocation. Adevs' conservative fails to complete the benchmark under valgrind even with a significantly reduced load. Optimistic exhibits the expected high memory usage.

Interconnect

In section 4.2 the parallel performance of this benchmark is further explained. Interconnect highlights dxex's lower performance due to message allocation overhead.

Priority Network

The priority network model is detailed in section 4.1.
The high memory usage of the conservative case is largely due to the garbage collection implementation. Kernel 0 is the initiator of the LBTS calculation but since it holds only models independent of any other (server), it will finish simulation very fast leaving the other kernel bereft of an updated LBTS. The kernel holding the server will after simulation wait on the receiving kernel until it can prove all sent messages can be deallocated.

adevs	adevs con	dxex	dxex con	dxex opt
39	39	35	43	259

Table 3. Interconnect w 20 t5e5, unit MiB, 2 kernels (if parallel)

dxex	dxex con	dxex opt
35	450	651

Table 4. Priority model n 128, m 16, p 10, t2e8, unit MiB, 2 kernels

5. RELATED WORK

5.1 PythonPDEVS

Dxex is closely related to PyPDEVS in design and philosophy. PyPDEVS allows anyone who grasps the PDEVS formalisms to immediately simulate his/her model without having to consider the kernel implementation. A Python based implementation offers the advantage of very fast prototype/run/evaluate cycles, this can't be matched by a C++ simulator. It should be noted that, once the kernels have been compiled (in shared libraries), the actual compilation time of the model is small enough to make prototyping possible. Advanced features such as activity based relocation and the performance gains this results in, are still unique to PyPDEVS.

5.2 Adevs

Adevs is still under active development, allowing for an exact comparison in performance and features. It remains a very fast simulation engine for the PDEVS formalism, but it lacks an optimistic synchronization implementation. Due to the fact that adev does not flatten the Coupled Models, it's performance degrades significantly if the hierarchy becomes deeper. Dxex, on the other hand, manages lookahead with more overhead than adevs, leading to a performance difference as the model count increases. Finally, adevs employs the full dependency graph this in contrast with dxex kernels, which only observe 1-edge removed nodes.

5.3 CD++

Different projects on CD++ offer conservative (CCD++) as well as optimistic (PCD++) parallel simulation. In contrast to dxex, neither projects offer both synchronization protocols. CD++ relies on the WARPED kernel. It is a middleware that provides memory, event, file, time and communication scheduling. We did not use the WARPED kernel (nor the underlying MPI) because dxex is designed specifically for a shared memory architecture and, as such, any middleware, however feature-rich, would have lead to an unacceptable overhead.

6. CONCLUSIONS AND FUTURE WORK

6.1 Memory

Our optimistic implementation can benefit from a faster GVT algorithm such as used in [8] or, more recently in [1]. With access to the memory subsystem already in place (pools), the optimistic case could be constrained using per kernel quota, preventing the simulation from exhausting memory. While

less sensitive to memory pressure, the same approach could improve the conservative case.

6.2 Activity

As shown in [15], activity and allocation of models across kernels is a key aspect in achieving high performance in any parallel implementation. Allocating models so that there are no dependency cycles between their containing kernels is a first step, but not always possible. For the optimistic case, one can use re-allocation to break (runtime dependency) cycles and/or perform load-balancing. If kernels are unevenly balanced, they will begin to drift fast, causing increasingly more reverts. In dxex, we implement a limited framework to track model activity for debugging purposes; this could be extended to enable the above strategies.

6.3 Hybrid

The optimistic implementation could use (null/eot/eit) from the conservative case to detect and/or reduce the cost of reverts without completely stalling on influencing kernels.
The eot calculation in the conservative case can be enhanced by using the entire dependency graph between kernels (not models). The implementation could detect dependency cycles between kernels and, instead of waiting on dependent kernels, advance as a synced group with better scaling. LBTS calculation could leverage this as well to reduce memory usage, though this is not that stringent in the conservative case. Ultimately, the simulation could switch at runtime between protocols based on the information provided by activity tracking. This requires the above mentioned framework, and can only be done at timepoints where a GVT/LBTS time is agreed upon between the kernels. From the model's perspective, no changes are needed, although a non-trivial lookahead is obviously desired.

In this paper, we presented dxex, a new C++-based PDEVS implementation supporting both sequential and parallel processing models.
We showed that, in the sequential case, dxex outperforms adevs in hierarchical simulations and that adevs produces better results in broadcast-like simulations.
In the parallel case, the results are more subtle. The sensitivity to broadcast simulations is carried over in the parallel implementation, resulting in performance loss for both synchronization protocols. The optimistic case can still achieve good performance if the volume of inter-kernel messages is low. Both synchronization protocols are fast in non-cyclic simulations, whereas the conservative case is better suited whenever state-saving becomes expensive; the optimistic case, on the other hand, is better suited when a lookahead is not available. Finally, the current optimistic implementation is ill-suited for memory constrained systems, especially in simulations where the frequency of reverts is high.

ACKNOWLEDGMENTS

This work was partly funded with a PhD fellowship grant from the Research Foundation - Flanders (FWO).

REFERENCES

1. Bauer, D., Yaun, G., Carothers, C. D., Yuksel, M., and Kalyanaraman, S. 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, IEEE Computer Society (Washington, DC, USA, 2005), 39–48.
2. Chandy, K. M., and Misra, J. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM 24*, 4 (Apr. 1981), 198–206.
3. Chow, A. C. H., and Zeigler, B. P. Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 26th Winter Simulation Conference*, SCS (1994), 716–722.
4. De Munck, S., Vanmechelen, K., and Broeckhove, J. Revisiting conservative time synchronization protocols in parallel and distributed simulation. *Concurrency and Computation: Practice and Experience 26*, 2 (2014), 468–490.
5. Franceschini, R., Bisgambiglia, P.-A., Touraille, L., Bisgambiglia, P., and Hill, D. 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 (2014), 40–49.
6. Fujimoto, R. M. Performance of Time Warp under synthetic workkloads. In *Proceedings of the SCS Multiconference on Distributed Simulation* (1990).
7. Fujimoto, R. M. *Parallel and Distributed Simulation Systems*, 1st ed. John Wiley & Sons, Inc., New York, NY, USA, 1999.
8. Fujimoto, R. M., and Hybinette, M. Computing global virtual time in shared-memory multiprocessors. *ACM Trans. Model. Comput. Simul. 7*, 4 (Oct. 1997), 425–446.
9. Ghemawat, S., and Menage, P. TCMalloc : Thread-Caching Malloc. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`, Nov. 2005.
10. 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* (2005), 265–272.
11. Jefferson, D. R. Virtual time. *ACM Trans. Program. Lang. Syst. 7*, 3 (July 1985), 404–425.
12. Mattern, F. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing 18*, 4 (1993), 423–434.
13. Nethercote, N., and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not. 42*, 6 (jun 2007), 89–100.

14. Van Tendeloo, Y. Activity-aware DEVS simulation. Master's thesis, University of Antwerp, Antwerp, Belgium, 2014.

15. Van Tendeloo, Y., and Vangheluwe, H. Activity in pythonpdevs. In *Activity-Based Modeling and Simulation* (2014).

16. Van Tendeloo, Y., and Vangheluwe, H. The Modular Architecture of the Python(P)DEVS Simulation Kernel. In *Spring Simulation Multi-Conference*, SCS (2014), 387 – 392.

17. 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.

18. Zeigler, B. P., Praehofer, H., and Kim, T. G. *Theory of Modeling and Simulation*, second ed. Academic Press, 2000.