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

1.1 DEVS

The family of DEVS [23] formalisms serve as a common basis for most other discrete event formalisms. Of interest in this paper are the 3 key formalisms: Classic [24], Dynamic Structured [1] and Parallel [4] and their implementation. This project uses the DirectConnect [3] algorithm, so from a kernel's perspective only Atomic Models exist in the simulation linked to each other by connected ports.

1.2 Parallel computing

Parallel execution of a DEVS simulation can lower runtime and increase the bound on the state space, thereby enabling simulation of more complex systems in the same time-frame. While the shared memory parallelism offered by most modern hardware does not lower the state space bounds, it can reduce the runtime and offers more direct communication and control between entities involved in synchronization compared to distributed simulation.

1.3 Motivation

Adevs [17] offers a very fast conservative synchronized shared memory DEVS simulator, but no optimistic synchronized variant. The latter can be significantly faster, especially in simulations where the runtime behaviour of the simulation is hard to predict.
The matured parallelism features of C++11 were used to in this project with the dual aim of writing standard-compliant (and thus portable) code without losing access to powerful low-level threading primitives.

1.4 Solution

The usage of the DirectConnect[3] algorithm makes reusing the adevs kernel hard. The dxexmachina kernel follows the design of the PythonPDEVS kernel where appropriate, but by the very nature of the implementation languages has to differ in key aspects (e.g. memory allocation strategy). The core aims of the project are to offer a deterministic simulation kernel where the simulation author is shielded as much as possible from the kernel implementation, without sacrificing performance. As in PyPDEVS, a model need be written only once for use in the different simulation kernels (with the exception of a non-trivial lookahead()).
The tracing framework from PyPDEVS was ported to allow optional verification of simulations.

1.5 Time

The DEVS formalism has $\mathbb{R}$ as time base, but any implementation has to decide on a (finite) representation of time. Dxexmachina kernels can operate on IEEE754 floating point time units, or integral time. The latter significantly reduces the possible range of the simulation, but avoids possible approximation errors in floating point. Furthermore, the notion of ϵ as the absolute minimum between time points needs to be established, this is non-trivial for floating point.
The select() function, which arbitrates between concurrent events, is implicit in this project by providing any time representation with a secondary causality field.

2. BACKGROUND

In this section, we provide a brief introduction to two different synchronization protocols for parallel simulation, and the features offered by C++11 that aid in our implementation.

2.1 Conservative Synchronization

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

Conservative synchronization is defined by the invariant that no model will advance in time before it has received all input from any influencing model.
This requires the concepts of eot (earliest output time) and eit (earliest input time) which define the timespan within which a model can safely advance.
The eit of any model is the minimum of all eot values of influencing models. A model can simulate up to (but not including) eit, and then has to wait until that value is increased. An important disadvantage here is that the influenced-by relation is always defined at model(link) creation, not at runtime. A model that can influence another, but never does, can severely slow down the protocol.
Deadlock between models that influence each other can be broken/avoided by a variety of means, in this simulator the CMB [2] null-message protocol is used.
Null message time is the timestamp a model is guaranteed to have past in simulation. More precisely, a null message of time t is a guarantee that any ouput with timestamp t-ϵ is already sent.
In general, the eot/eit/nulltime of a kernel is the mimimum of each of those values for all models in the kernel.
Conservative explicitly relies on information provided by the model creator in the form of lookahead, a relative timespan during which the model is insensitive to outside events. This can be non-trivial to calculate, a simulation writer will in general not be able to predict the exact lookahead of models involved in an experiment without having run the experiment.
The performance is more sensitive to lookahead than it is to cycles in the (static) dependency graph.

2.2 Optimistic Synchronization

Optimistic synchronization allows causality errors to occur but employs a roll-back mechanism to recover from those errors. The most common mechanism is the Timewarp [12] mechanism, whenever a kernel receives an event with timestamp in the kernel's past, the state of the kernel is reverted to that time. The gain in runtime this provides is offset by the increase in memory required to keep saved states and (sent) messages. Optimistic does not rely on any domain specific information, in contrast to conservative. It is only sensitive to runtime use of connections, not the probability that they might occur.
If the (runtime) dependency graph contains a cycle, optimistic can suffer a series of cascading reverts worsened if the timestamps of the events match exactly.
This effect can be lessened by lazy cancellation and/or lazy re-evaluation [6].

2.3 Global Virtual Time

To avoid exhausting memory in state/event saving, optimistic synchronization relies on the concept of global virtual time[12]. In optimistic simulations, GVT is defined as the lowest timestamp of any unprocessed event.
Intuitively this is the timepoint in the simulation that is certain to be preserved, corresponding exactly with the simulation up to that time in a non-parallel implementation.
In conservative the minimum simulation time of all kernels is the GVT, or in terms of null messages: the least timestamp of any null message in transit. In any parallel implementation the GVT calculation is vital to safely commit unrecoverable transactions such as IO (e.g. tracing), releasing memory, ...

2.4 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 exchanged not as messages but reads/writes to atomic fields shared by all kernels. This avoids the otherwise unavoidable latency penalty by mixing simulation messages with synchronization messages. 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 PythonPDEVS

The simulator is based on PythonPDEVS, and provides the following features:

1. Direct Connect
2. Dynamic Structured DEVS
3. Termination function. If specified, a termination function is applied every simulation round to each model to test whether the simulation can terminate. Only available in single-threaded simulation.
4. State/Message can have any payload type. Different message types can be used together within the same simulation.
5. Tracing An asynchronous, thread safe and versatile tracing mechanism allows exact verification of the simulation.

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 a non-negligible impact on the runtime, even when not in use.
The tracing is not comparable with adevs's listener interface. To be usable in 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 threadsafe and deterministic so that a simulation will always produce the exact same output. The following features from PythonPDEVS are not present

1. Activity tracking and relocation
2. Serialization
3. Interaction
4. Distributed simulation
5. Interactive control

Serialization in this context is the ability to save/load a complete simulation to disk, not the state saving mechanisms required for TimeWarp.
State saving has no impact in a single threaded or conservative kernel.
Model allocation is done 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.
Debugging tools such as a logger and a graph vizualizer are included which can track activity with respect to allocation for later study, but not online/dynamic as is possible in PyPDEVS.
While PythonPDEVS can be controlled from within a Python script and adevs has a Java interface, our implementation does not have any bindings to other languages.

3.2 Different Synchronization protocols

Conservative

A conservative 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 from the current time, the timespan during which the model cannot change state due to an external event. Internal transitions are safe. This information is collected for all models hosted on the kernel, and the minimum 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 eot of all influencing kernels.
For garbage collection (of sent messages) the GVT is calculated as $\min_{\forall i \in \text{influencors}}(\text{nulltime}[i]\ ) - \epsilon$.

Optimistic

The optimistic kernel requires from the hosted model only that copying the state is well-defined, which is provided in the base State class for the user.The kernels use Mattern's 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/(anti)messages.

The user need only provide one implementation of a model for use with both synchronization protocols. A lookahead function is desired to accelerate conservative, but is not required. In the absence of a user supplied lookahead, the kernel assumes it cannot predict beyond its current time + ϵ, creating a lockstep simulation.
The implementation details such as defining the copy semantics of a State are provided (but can be overridden).
From the user's perspective, the multi-threaded aspect of the kernel is not exposed.

3.3 Performance Improvements

Continuous profiling of the kernels in several benchmarks highlighted the following key bottlenecks.

1. Heap allocation
2. Strings
3. Scheduling
4. Locking
5. Smart pointers

Heap

A kernel never sends a complete object to another kernel, only a pointer to the object.
The cost of heap allocations was minimized using thread_local memory pools, combined with the (optional) usage of tcmalloc[7]. This changed the ownership semantics of several objects in a non-trivial way, since the thread that creates an object has to destroy it (iff it can prove it is no longer used). Experiments with synchronized pools proved slower than malloc/free itself.
Replacing strings (in c++ : heap allocated variable sized arrays) with integer identifiers substantially decreased the runtime. This is a tradeoff, the user has more affinity with names than numbers as identifiers, so this translation is done mostly internal. A Model can be named using a std::string, but once the model is handed off to the kernel only integer identifiers remain (the same goes for ports, kernels, ...).

Avoiding Smart pointers

While an important feature in C++11 in general, our initial usage of smart pointers for some types of objects was misplaced. Used across threads the reference counting becomes prohibitively expensive. Models will still be held by a smart pointer, as will a kernel, but a message is a raw pointer to compacted memory. Ideally a message object can (depending on payload) fit on a single cacheline.

Locking

Locking between kernels uses mostly atomic operations, where we can occasionally leverage memory orderings to only pay for synchronization when we need it. Messages 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 content (timestamp) directly into shared memory. This alleviates in part the potential delay/latency caused by null/eot messages being mixed with simulation messages in conservative implementations.

Schedulers

PythonPDEVS has a wide range of scheduler for the user to choose from, with performance of each depending on the simulation type. Profiling showed in our case that, for a c++ implementation, the heap implementation used in adevs was faster than any of the schedulers we had tested before. Unlike most node based heaps, this scheduler uses a fixed size array where a heap is rebuilt or modified. Items are only updated, never removed.

4. PERFORMANCE

4.1 Sequential Simulation

CPU Usage

Memory Usage

We measure memory usage as allocated pages at a given time t. Note that our usage of pools will always overestimate actual memory in use, but from the OS' point of view (and that of the user), any allocated page is in use.

Our usage of tcmalloc should be taken into account here as well, tcmalloc will only gradually release memory back to the OS.

To have an fair comparison (our copy of) adevs also uses tcmalloc.

Actual profiling of memory usage is done with Valgrind's [16] massif tool. Our preference of allocating more on the heap w.r.t adevs stack usage is equalized in tracking only the total amount of pages, with stack, text and heap memory all in one measure.

4.2 Parallel Simulation

Devstone

The devstone [8] benchmark is highly hierarchical, as a result adevs suffers a performance hit linear in the depth of the (top) model. The atomic models in dxexmachina's version of devstone are allocated in a single chain cut in equisized subchains. This significantly reduces the nr of messages to pass between kernels which reduces in part memory pressure (local messages can be destroyed immidately whereas remote messages need to wait for GVT). If the nr of models increases, optimistic on fixed timeadvance becomes slower due to drifting.

PHold

In Phold [5] Allocation is specified in the benchmark itself, each kernel manages a single node with varying subnodes.

The dynamic dependency graph is a very sparse version of the static dependency graph, penalizing conservative. Lookahead is ϵ, so conservative 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.

Optimistic suffers little from the above problems, however due to the high interconnectivity a cascading revert is still possible.

Interconnect

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

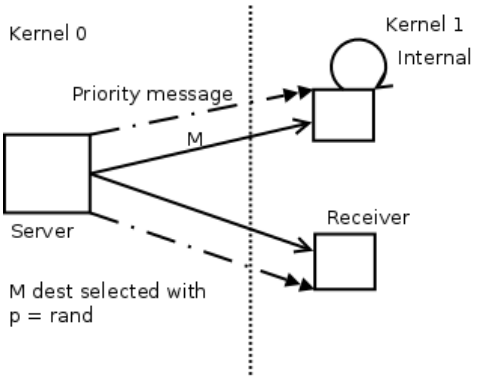
Allocation is irrelevant, the resulting dependency graph between kernels still is a complete graph. The only difference now is that the dynamic graph will converge very fast to the static graph. Conservative still faces the same issues as in PHold, with the key difference that for a fixed time advance lookahead is well defined and is equal to the timespan between transitions. Since all kernels have to wait on each other here, this simulation scales very poorly in nr of kernels. Our optimistic implementation does not complete an instance of this benchmark. The kernels get stuck in an infinite cascade of reverts. If kernel A reverts, it will send antimessages to all others who in turn revert and send antimessages to all others. Support for lazy cancellation could undo this anti-pattern.

Strengths-Weaknesses

Priority network model

The priority benchmark is composed of a single server generating a stream of messages at fixed time intervals, interleaved with a probability p for a priority messages.

The priority messages defaults lookahead 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 a internal transition so will not wait for the incoming messages. The number of addressed receivers m is variable, with addressed receivers randomly selected.



5. RELATED WORK

5.1 PythonPDEVS

Dxexmachina is closely related to PythonPDEVS in design and philosophy. PythonPDEVS allows anyone who grasps the DEVS formalisms to immediately simulate his/her model without having to consider the kernel implementation. C++ implementations cannot hope to match the fast prototype/edit/run cycles provided by PythonPDEVS, although this can be minimized by building the kernels as libraries. Advanced features such as activity based relocation and the performance gains this results in, are still unique to PythonPDEVS.

5.2 Adevs

Adevs's source code is still under active development, allowing for an exact comparison in performance and features. It remains in most aspects the fastest simulation engine for the DEVS formalism, but it lacks an optimistic synchronization implementation. By virtue of not flattening Coupled Models, performance suffers in increasingly hierarchical models.

5.3 CD++

5.4 Warped

The Warped kernel was not used since we operate explicitly for a shared memory system and wanted to design our kernels using the least amount of overhead possible.

6. CONCLUSIONS

ACKNOWLEDGMENTS

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

REFERENCES

1. Barros, F. J. Modeling formalisms for dynamic structure systems. *ACM Transactions on Modeling and Computer Simulation 7* (1997), 501–515.

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. Chen, B., and Vangheluwe, H. Symbolic flattening of DEVS models. In *Summer Simulation Multiconference* (2010), 209–218.

4. 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.

5. Fujimoto, R. M. Performance of Time Warp under synthetic workkloads. In *Proceedings of the SCS Multiconference on Distributed Simulation* (1990).

6. Fujimoto, R. M. *Parallel and Distributed Simulation Systems*, 1st ed. John Wiley & Sons, Inc., New York, NY, USA, 1999.

7. Ghemawat, S., and Menage, P. TCMalloc : Thread-Caching Malloc. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`, Nov. 2005.

8. 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.

9. Glinsky, E., and Wainer, G. New parallel simulation techniques of DEVS and Cell-DEVS in CD++. In *Proceedings of the 39th annual Symposium on Simulation* (2006), 244–251.

10. Himmelspach, J., and Uhrmacher, A. M. Sequential processing of PDEVS models. In *Proceedings of the 3rd European Modeling & Simulation Symposium* (2006), 239–244.

11. Jafer, S., and Wainer, G. Conservative vs. optimistic parallel simulation of devs and cell-devs: A comparative study. In *Proceedings of the 2010 Summer Computer Simulation Conference*, SCSC '10 (2010), 342–349.

12. Jefferson, D. R. Virtual time. *ACM Trans. Program. Lang. Syst. 7*, 3 (July 1985), 404–425.

13. Kim, K. H., Seong, Y. R., Kim, T. G., and Park, K. H. Distributed simulation of hierarchical DEVS models: Hierarchical scheduling locally and time warp globally. *Transactions of the SCS 13*, 3 (1996), 135–154.

14. Muzy, A., and Nutaro, J. J. Algorithms for efficient implementations of the DEVS & DSDEVS abstract simulators. In *1st Open International Conference on Modeling and Simulation (OICMS)* (2005), 273–279.

15. Muzy, A., Varenne, F., Zeigler, B. P., Caux, J., Coquillard, P., Touraille, L., Prunetti, D., Caillou, P., Michel, O., and Hill, D. R. C. Refounding of the activity concept? towards a federative paradigm for modeling and simulation. *Simulation 89*, 2 (2013), 156–177.

16. Nethercote, N., and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not. 42*, 6 (June 2007), 89–100.

17. Nutaro, J. J. ADEVS. `http://www.ornl.gov/~1qn/adevs/`, 2015.

18. OpenMP Architecture Review Board. OpenMP application program interface version 4.5, 2015.

19. Troccoli, A., and Wainer, G. Implementing Parallel Cell-DEVS. In *Annual Simulation Symposium* (2003), 273–280.

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

21. 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.

22. 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 (2015), 844–851.

23. 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.

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