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
Leveraging the shared memory parallelism offered by most modern hardware, the runtime of non-trivial model can be significantly reduced.

1.2
Adevs offers conservative synchronized parallel simulation based on OpenMP, but does not by design support optimistic synchronization which can, depending on the simulation model, be significantly faster. Adevs relies heavily on templates (as do we), but does not hide the types itself.

The usage of the DirectConnect algorithm makes reusing the adevs kernel impossible. 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 (among others memory allocation strategy). Adevs's explicit usage of (payload)typed messages allows for a fast stack-allocated exchange, whereas dxexmachina uses pointers to heap allocated object (with a runtime disadvantage in some simulations). It does give the user the freedom to wrap any type in a message without the type being fixed for all messages.

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
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.
No model can advance beyond its own eit, which is defined as the min of all eot values of influencing models. 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 by a variety of means, in this simulator the null-message protocol derived from (Chandy/Misrah 79) is used to avoid and break deadlock.

2.2 Optimistic Synchronization
Optimistic synchronization allows simulation to advance in time until it receives a message with timestamp in its past. At receipt of such a message, the simulation is rolled back to the time of the received message to undo the causality errors. Optimistic therefore only uses the runtime 'influenced by' relationship, it will only react to actual sent messages between models. The cost for this speedup is memory, saving all states until a kernel can verify that all other kernels have advanced equally far.

2.3 Global Virtual Time
To avoid exhausting memory in state saving, optimistic synchronization relies on the concept of global virtual time, a timepoint that all kernels have simulated past without having to revert before it. While this is critical in optimistic to

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

garbage collect old states (and/or messages), usually a conservative kernel will employ GVT as well for the same purpose (but with less memory pressure). In conservative synchronization, the null messages can be used to determine GVT while in optimistic this is far more complex.

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 earlier versions there was by default no standard way to write a multi-threaded application. Our kernels use a wide range of threading primitives and atomic operations tailored for each use case. As an example, eot/eit/nulltime are exchanged not as messages but as writes to atomic fields shared by all kernels. 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. (Only) In single-threaded simulation a termination function, if specified, is applied every simulation round to each model. If this is not present, there is no overhead.
4. State/Message can have any payload type.
5. Tracing An asynchronous Tracing mechanism allows exact verification of the simulation.

The implementation tries to adhere to the C++ principle that you don't pay for what you don't use. The termination function is non-trivial to implement in a multi-threaded kernel and had a non-negligible impact on the runtime even if not in use, for this reason it was abandoned.
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 cores.
Debugging tools are included which can track activity with respect to allocation for later study, but not online/dynamic as is possible in PyPDEVS.
PythonPDEVS can be controlled from within a Python script, this is not implemented in dxexmachina.

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 a kernel drives, and the minimum is set as the lookahead of the kernel itself.
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(nulltime[i] $\forall i$) $-\epsilon$. This is one of the operations that can benefit from using a relaxed memory ordering.

Optimistic

The optimistic kernel requires from a 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 max 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. 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, but never items removed.

4. PERFORMANCE

4.1 Sequential Simulation

CPU Usage
Memory Usage

4.2 Parallel Simulation

5. RELATED WORK

6. CONCLUSIONS

ACKNOWLEDGMENTS

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