EDX – A COMMERCIAL PARALLEL SIMULATOR PLATFORM
Implementation of Optimistic Parallel Simulation Algorithm

Janis Britals
Incontrol Enterprise Dynamics GmbH, Gustav-Stresemann-Ring 1, 65189 Wiesbaden
janis.britals@incontrolsim.com

Abstract: Theory of optimistic parallel simulation algorithms has been explored for quite some time. Successful implementations in commercial simulation packages, however, have so far been missing. During research into improvement aspects for our discrete event simulation platform Enterprise Dynamics we implemented various features required for such algorithms: a complete state saving and rollback mechanism, support for efficient event queue driven multithreading and bindings to Lua scripting language. These advances have led us to an implementation of generic parallel simulation environment enabling the model builder to designate parts of his/her model to be executed in a separate simulation thread with a click of a mouse. Thanks to usage of a script language the same code can be executed both in parallel as well as sequential settings allowing the model builder the utmost flexibility in exploring the potential parallelism in his/her model.

1 Introduction

Our company has specialized in the field of discrete event simulation and we have developed a general-purpose simulation platform called Enterprise Dynamics. We have been successfully working with this platform ourselves and also sold it to hundreds of demanding customers during the last 10 years. Of course, the field of discrete event simulation as well as general software engineering has not remained still during those years, so some time ago we decided to look at ways how to modernize the platform.

One of the most obvious changes in the IT trends of the last years has been the shift of computer hardware development from single- to multi-processor systems. Still, most of the commercial simulation packages on the market including Enterprise Dynamics are ill prepared to take advantage of this development, since their simulation process remains fundamentally single-threaded. So it seemed quite natural for us to look at possibilities how to change this.

The theory of parallel discrete event simulation – splitting the simulation process among several independent execution threads – has been quite well known for some time, and it has been very well explained in the textbook of Richard M. Fujimoto[1]. It is no secret that the underlying theory is quite complex and could be difficult to implement in real-world computing environments, which explains why it has not found wider adoption among commercial platform vendors. Still it seamed worthwhile to look at possibilities, so we started a research program aimed at producing a new general-purpose parallel discrete event simulation platform called EDX.

At first we started implementing various technologies required for such platform – like state saving (checkpointing), rollbacks, scripting language, modelling environment, visualization. Gradually all these technologies took shape and now we are at the point in our development where we need to integrate them in the final product. We are looking towards the Alpha release of EDX platform that shall be released sometime in the coming months.

Here we are introducing some key features of this platform and our approach to building models for parallel execution. Our aim throughout was to create a platform that would make programming such models as easy as possible – ideally the fact whether the model was running under parallel or sequential simulator should be completely transparent to the model builder. In our opinion we succeeded, and here I try to explain how we did it.

2 EDX Key Features

2.1 Lua as script language

Choice of scripting language (or programming language) for a simulation platform has most profound effect on all aspects of this platform. It is the language in which model builder will express his or her ideas and constructs. Ideally this language would be very versatile – to allow rich ways of expression, easily extendible – to provide for situations when built-in libraries are not sufficient, and fast.

It is possible to build simulation platforms using compiled languages (like C++ or Pascal) for writing model code – there are a couple of examples – but in general it is not such a good idea. It may give the fastest possible code, but compilation step required between coding and running the model makes the modelling process very cumbersome and eventually drags on end-user productivity.

Scripting languages on the other hand are usually interpreted on the fly during run-time and let the end-user make adjustments to the model code even while model is running. Here however the speed becomes an issue, since most of the scripting languages execute much slower (hundreds if not thousands of times) than their compiled alternatives. Our existing platform Enterprise Dynamics solved this dilemma by inventing its own functional scripting language (4DScript) and coding its interpreter directly in assembly code. This achieved acceptable speeds, but came at a cost for flexibility, extendibility and was also quite difficult to support.

When deciding on the scripting language to be used on the new platform we took a careful look at available options and after some tests the choice fell on Lua. It might look like a surprising decision at first, Lua being a relatively unknown niche language, but here are the main distinguishing features that helped to convince us:

1) Lua is very fast. This probably derives from the register-based architecture of the language interpreter (most other scripting languages are stack-based). It also has quite good just-in-time (JIT) compiler that helps to speed massive data manipulations even more, and there is a second-generation JIT compiler in the works, which will be more advanced and could possibly speed certain algorithms even faster than their compiled versions.
2) Lua is quite modern. It provides constructs like closures or co-routines and has a very powerful meta-programming paradigm based on metatables. It also treats functions as first-class values – they can be manipulated in the same way as other data types (strings, numbers, etc.)
3) Lua is extremely flexible. Its meta-programming approach can be used to create customized programming paradigms tailored to any specific application. We, for example, have implemented a complete OO programming paradigm suited for our simulation models. (Lua has no built-in OO paradigm, but it provides tools to build your own)
4) Lua is simple. It has very limited range of data types. It recognizes only one numerical data type (number), and there is only one structured data type – table.

The choice of Lua has been very fortunate – it has helped us to implement several important platform features with relative ease. I will shortly describe some of the most important aspects.

2.1.1 Simple data structures

Lua knows only 8 data types: nil, numbers, strings, booleans, functions (or closures), tables, userdata and threads (co-routines). Of these only one (tables) is used to build more complex data structures.

Lua table is universal – it can be either an array (with numbered indices 1, 2, 3, …), or dictionary with arbitrary typed keys – or both. It knows 2 basic operations: __index (looking up value from a key) and __newindex (storing value with a key). Basis of Lua flexibility is the concept of metatables – they are normal tables that are associated with other values (usually other tables or userdata). These metatables contain certain metamethods that define behaviour of the associated values in specific situations. For example, when you look up value from a key in a table that does not contain this particular key, the __index metamethod is invoked. Similarly, when storing a new value in a table, the __newindex metamethod is invoked.

These 2 metamethods can be used to build the most complex metabehaviour. We used them to implement our own OO programming paradigm complete with classes, object instances, inheritance, polymorphism, signals - slots, etc. A distinguishing feature of our implementation is the possibility to extend inheritance chain across the C++/Lua boundary, i.e. you can define Lua subclasses of C++ superclass.

The underlying simplicity of Lua data structures means that it was relatively easy to isolate any state alterations, a necessary precondition to implement state saving or checkpointing. Basically you need to trace all table “newindex” events (both when the key does not exist and when it does exist). It could be achieved with minor modification of the Lua interpreter.

2.1.2 Solid control over state

Another feature of Lua that helps to simplify state manipulations is the concept of "Lua state". The complete state of the virtual machine is contained in a structure that is always passed along as the first argument to any Lua API function (i.e. a C++ call to Lua virtual machine). This makes it quite straightforward to isolate all state manipulations pertaining to one Lua state.

In practice it turned out to be convenient to associate a Lua state with a single Time Warp logical process (TWLP) so that each Lua state had its own event queue and communicated with other TWLPs (represented by other Lua states) via messages.

That meant also that we could centralize the checkpointing in each Lua state, as we needed one checkpoint stack per TWLP.

2.2 State saving

Central to Time Warp synchronisation mechanism is the ability to capture and store the state of Time Warp logical process (TWLP) so that in case of conflicting incoming messages it is possible to restore this state to an earlier point in local simulation time. This process is called "checkpointing" and the periodically stored states – checkpoints.

Our choice of checkpointing mechanism was driven by architecture of Lua as our modelling language. EDX uses incremental state saving exactly as described by Fujimoto[1]. The implementation was quite straightforward. As explained in previous paragraph what we had to do was simply to put a guard on every __newindex event, and then store the old value under the same key in a separate checkpoint table.

A small problem is presented by userdata values that usually represent external objects (like C++ objects) and could have very complex reference semantics. For simple cases of userdata with value semantics this can still be solved in our binding mechanism used to wrap these objects in Lua. It is achieved by using copy state saving in accordance to Fujimoto[1]. More complex objects with reference semantics require more nuanced approach and maybe some custom programming by end-user. It is always possible to wrap such objects in "proxy" Lua objects that control the checkpointing behaviour.

The bottom line is that every event processing in a TWLP will generate one checkpoint table containing all information necessary to bring the corresponding Lua state back to the state it was before this event.

2.3 Multistate multithreading

Of course, in order to profit from Time Warp parallelization we should run the different TWLPs in separate execution threads. Thus we had to develop a multithreaded executable with all the synchronisation issues that that entails.

Lua virtual machine (LVM) is fundamentally single-threaded, but it is not a problem since every Lua state is associated with a different TWLP and each TWLP always runs in a single thread. We just had to make sure that every call to Lua API is synchronised on Lua state.

To deal with synchronisation issues we developed 2 mechanisms. First, to handle direct communication between different threads, (e.g. to inspect certain values in simulation threads and display them in GUI thread) we used the locking mechanism provided by Qt (QMutex) and locked the Lua state at each entry point into the Lua stack. This provides for very fast mechanism with minimal overhead, but it will block the calling thread while the call is being processed. This is appropriate for ad-hoc exchanges of information between the threads (mostly used to interactively inspect Lua values in simulation threads), but is not usable for message passing between TWLPs.

For simulation messages another synchronisation mechanism was developed, based on signal-slot communication feature in Qt. Since every call from one TWLP to another by definition is a message exchange this inter-thread communication became part of the Parallel simulation algorithm described in the next chapter.

3 Parallel Simulation

3.1 Time Warp Algorithm

According to Fujimoto[1] there are 2 fundamental approaches to event synchronisation algorithms in Parallel discrete event simulation systems: conservative and optimistic. Conservative algorithms are characterized by avoiding any causality violations by stopping any logical process until it is “safe” to simulate it further. This approach avoids unnecessary calculations, but it is only feasible if the simulation model provides a certain degree of “lookahead”, i.e. guarantees that violations shall not occur during a simulation time interval. This lookahead is inherent in the particular model and is not possible to calculate from the model transparently, i.e. it must be provided by the model builder. This makes it not very well suited for a generic simulation platform intended to simplify modelling process.

This leaves us with optimistic synchronisation algorithms like Time Warp. Advantage of this algorithm is that it operates independently of the way the model is split up among different Time Warp logical processes (TWLPs). They allow the parallelization to be implemented independently of the model structure. The model builder only needs to indicate which parts of the model shall reside in which TWLPs, and the rest shall be handled by the simulation engine transparently.

For the first version of EDX we decided to implement the basic version of Time Warp algorithm taken right out of the book[1], with anti-messages, etc.

3.1.1 Atoms and events

We already saw that to structure the simulation in parallel logical processes we associate each TWLP with a separate Lua state and an event queue. Each event from the queue is taken in a computer-generated timer event and processed. This neatly associates the simulation events (residing in the event queue) with system (timer) events generated by computer. In this manner it is relatively easy to time the simulation and run it in synchronisation with wall-clock, if necessary. It also provides convenient entry points for other computer-generated actions, like user- or network-events to be processed between simulation actions.

The remaining question is how to structure the simulation model built by the end-user so that it can take advantage of the many available TWLPs? Here we introduced the concept of atoms.

To users of our current simulation platform Enterprise Dynamics the atom concept should already be familiar. It is the basic building block of a model. You can build large and complex models by taking atoms from a library one by one and linking them together. To facilitate this process the atoms have a parent-child relationship with each other and the resulting model is one big tree of atoms.

The atoms in EDX have a similar role as basic building blocks of a simulation, but they have one more important property:

<u>The atom is the fundamental unit of simulation that is completely simulated in one TWLP.</u>

Atoms are created in the same TWLP as their parent, but they can move from one TWLP to another. In that case all its children move along with their parent. You can also start a new TWLP to simulate a certain atom.

As we saw in previous section the Time Warp algorithm is transparent to the way atoms are distributed among TWLPs, so the end-user can simply click on the various atoms in his or her model and assign them to separate TWLPs as needed. In this way the end-user can play with different distributions of atoms among separate processes to empirically discover the parallelism inherent in his/her model.

3.2 Transparent user interface

Ability to push around atoms among different TWLPs brings us halfway to the goal of creating an easy-to-use parallel simulation platform. However it would not amount to much if end-user would need to change code in order to distinguish between direct calls to atoms residing in the same TWLP and sending messages to atoms in other TWLPs. In other words we had to ensure that calls to other atoms were transparent in regard to the TWLP they were located in.

Here again Lua’s flexibility comes to the rescue. In order to invoke methods on other atoms you have to address the atom object in Lua. We could design it in such a way that it transparently takes care of those calls:

if the called atom lives in the same TWLP then its methods are called directly, otherwise a message is sent to the target TWLP containing this call and its timestamp. This message is inserted in the receiving TWLPs event queue and processed in due course. A record of this message is kept in the sending TWLP, so that if it becomes necessary to roll it back, an appropriate anti-message can also be sent.

```
1. Function atom_object:method(args) :
atom_object.atom is in same Lua state(TWLP)?
Yes: do   atom_object.atom:method(args)
No: send message atom_object.TWLP->call(atom_object, method, args)
2.  Event function call(atom_object, method, args) :
do        atom_object.atom:method(args)
```

Table 1 Atom call control flow

This simple logic becomes more complicated if we must return a value. If the call was to an atom in a different TWLP then returning a value must involve another simulation message in the opposite direction.

3.2.1 Checking return values

To solve the problem of transparently returning values from the calls across TWLP boundaries we turned again to our Lua toolkit and decided to “misuse” the length operator (# - hash sign) for this purpose. A simple call to another atom that is returning a value is split in two: first the actual call processed in accordance to the logic described in previous section, and then the “hash” operator applied to the result of this call that yields the result. To illustrate this, suppose that we want to call a method on another atom like this:

```
result = atom_object:method(args)
```

To achieve this we need to change this call to:

```
result = #atom_object:method(args)
```

What is happening here is that the call to the atom object, after performing the action (sending message if necessary), returns a result object that “remembers” this call. When we apply “hash” operator to this result object it checks whether the actual results from the call have been received (waiting for the message from other TWLP if necessary) and then returns these results.

```
1. Function atom_object:method(args) :
atom_object.atom is in same Lua state(TWLP)?
Yes: do   result_object.results = atom_object.atom:method(args)
          result_object.results_received = true
No: send message atom_object.TWLP->call(atom_object, result_object, method, args)
return result_object
2.  Event function call(atom_object, result_object, method, args) :
do results = atom_object.atom:method(args)
send message result_object.TWLP->return(result_object, results)
3.  Event function return(result_object, results) :
do        result_object.results = results
          result_object.results_received = true
4. Operator #result_object :
block until result_object.results_received == true
return result_object.results
```

Table 2 Atom call control flow returning results

This control flow guarantees that we will only block execution of a TWLP if the results are actually needed, and at the point in the program flow where it is needed. The hash operator can be applied to the result object also later in the program, thus allowing the two TWLPs run in parallel for a while before blocking one of them in expectation of the results from the other, like this:

```
result_object = atom_object:method(args)
…
do calculations
…
result = #result_object
```

The most important result is that the programming code of the model that the end-user writes can follow one very simple rule: whenever calling functions on atoms, if you need to process results, precede the call with a hash (#) sign – the engine will take care of the rest. And it is quite efficient – there will be no unnecessary messages and the execution will block only the necessary minimum of time.

4 Conclusions

The EDX simulation platform, built on the principles illustrated above, is nearing its alpha-release status. We believe that it will change how people look at parallel simulation exercise. Users of the platform will be able to build their models in the same easy way that they have been used to in current object oriented simulation platforms (like Enterprise Dynamics): by combining pre-packaged building blocks (atoms) and extending their behaviour in procedural code blocks, responding to and scheduling simulation events. These code blocks will be straightforward to program: you can simply take references to other atoms and execute methods on them.

The resulting model code will run exactly the same whether all the atoms shall be created and executed in a single execution thread or whether some of the atoms shall be dispersed among several TWLPs running in parallel.

This should allow the model builder an easy experimentation platform to play around with his / her model and try to discover the inherent parallelism in it.

Seeing the direction of computer hardware development in the last years: providing more and more processors per unit, instead of pressing more juice out of a single microprocessor, the EDX platform should provide the user with tools necessary to take advantage of this development.

Besides potentially speeding up larger simulation models there are also several other applications of the parallel simulation technologies of EDX platform that could be of immediate benefit.

4.1 “Undo” function for model builder

A proper “undo” functionality, the likes of which people have come to expect from almost any other modern computer application (office programs, programming environments, etc.) is notoriously difficult to implement in a simulation modelling platform. To undo changes to the model during the modelling process it would require that the previous state of the model be fully saved. It is even more difficult in environments like Enterprise Dynamics, where changes to the model can be applied dynamically during the simulation runs.

EDX has the solution – the checkpointing functionality necessary for implementation of rollbacks can also be applied to restore the model to earlier states during modelling. The modelling process (running in the GUI thread of the EDX application) can be regarded as a separate TWLP running at the current global virtual time. Then the “undo” function is reduced to a simple anti-message recalling the change performed by the last modelling action.

4.2 Speedup of visualization and animation

Another area of application for the parallel simulation technology is visualization. For large models the burden of building their 2D and 3D visualizations and animating those can become quite prohibitive. In a single-process simulation there is no way around this problem: all the visualisation code has to be run in the same process as simulation.

EDX offers a different approach. All 2D and 3D objects used to visualize the model can be put in separate atoms and run in a separate TWLP (as part of the main thread). Since visualizations usually depend on a small set of status variables (like coordinates, orientation, colour, texture), it is easy in the main simulation TWLP to keep track of only the necessary set of these variables and send all their changes in messages to the visualization process. Since this information flow usually is unidirectional (normally the simulation does not depend on the visualized picture) there is zero chance that visualization shall send back messages or anti-messages, so the progress of the simulation process does not depend on the fact whether there is an active animation or not (provided there is a spare processor available in the system).

4.3 Time control for emulation and human-in-the-loop simulation

Third application area directly benefitting from parallel simulation capabilities of EDX is emulation (including human-in-the-loop simulation as special case). The challenge for emulation applications is the correct time-stamping of incoming events and outgoing messages. Enterprise Dynamics for example take the opportunistic approach: the incoming events are processed directly when they are received by simulation process at whatever logical simulation time it represents. As long as simulation is relatively idle it is quite ok, but if computer is busy doing some complex calculations at the time the external event processing can be considerably delayed. Then the incoming event will get a wrong time-stamp that could lead to causality conflicts.

The solution again is parallel processing. The external interface (visualization for human-in-the-loop or TCP/IP sockets for emulation) is put in a separate TWLP that does nothing else, so that external events get time-stamped with exactly right simulation time, and the Time Warp mechanism takes care of the right processing and correct causal consequences of these events.

REFERENCES

1 Fujimoto, Richard M., 2000. *Parallel and Distributed Simulation Systems*, John Wiley & Sons.