SimVars: A Simulation Software Primitive to Facilitate Parallel Simulation Software Development

(Poster Abstract)

Nilo Ney Coutinho Menezes
Multitel ASBL
Rue Pierre et Marie Curie, 2
Mons, 7000, BELGIUM
menezes@multitel.be

Roberto Melo Cavalvante
Multitel ASBL
Rue Pierre et Marie Curie, 2
Mons, 7000, BELGIUM
cavalcante@multitel.be

ABSTRACT

SimVars are software primitives used to avoid lock coordination and to simplify parallel simulation software. We propose a simple technique to represent the change of value of a variable in time. SimVars are represented as sparse arrays, where a value is valid until the next entry, allowing a compact representation of changes in memory, avoiding lock coordination, providing a set of operations used in common parallel discrete event simulation software and a software abstraction that mimics some functional programming techniques in common imperative languages.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel programming

General Terms

Theory

Keywords

parallel simulation, multi-core simulation

1. INTRODUCTION

During the development of a new simulation software, we were faced with the problem of parallel software development. The previous version of the commercial simulator we used required many computers in a network to run simple simulation scenarios. Most of the time, processor utilization levels were extremely low, fostering the idea to run the same simulation using a single computer.

We started to evaluate the most interesting properties of functional languages, like immutability and some features of Erlang, like message passing. Our goal was to design a simple structure able to shield normal programmers from the intricacies of parallel software programming, even if they decide to use a commercial imperative language like C#.

SIMUtools '14 Lisbon, Portugal

2. RELATED WORK

The idea of virtual time is not new to simulation[8, 4, 7], but it is very important regarding distributed and parallel systems. The virtual time combined with time parallel simulation allows the construction of large systems without the constraints of real time. It also enables distributed simulations to coordinate their times using algorithms like Time Warp.

Parallel discrete event simulators have been implemented using Erlang as their programming language to facilitate program construction and to reduce the pitfalls of parallel software development[11, 5, 1]. Functional programming languages have been pointed as a better way to write parallel software, but that a multi-paradigm (functional and imperative) approach can provide better results[9, 3].

We developed a time parallel simulator that combines event driven and time stepped simulation with a synchronization aware state queue, using a sparse vector primitive that we call SimVar.

3. SIMCORE & SIMENTITIES

The development of our new simulator was split in three parts: SimCore, SimEntities and SimVars.

SimCore is the simulation engine, responsible for keeping lists of active simulation entities, variables and events. Its main task is keeping together the simulation and controlling the simulation time. SimCore implements a mixture of time stepped and event based discrete simulation. It uses discrete time to compute and update entities states, but it is also capable of storing and forwarding events. These events are sorted by simulation time and by destination. Events are used to enable message passing between entities, no other dependencies between entities are allowed.

Each SimEntity is a Logical Process (LP), as in [6]. It is the basic simulation unit inside SimCore and it is used to coordinate SimVars. At every simulation step, each SimEntity is called by SimCore to update its internal state and optionally to generate events. A SimEntity never communicates directly with other entities, but through an event exchange service provided by SimCore.

4. SIMVARS

SimVars were designed to provide a light weight software abstraction to a simulation entity state. In an imperative programming language, variables can have their state changed any time and do not keep record of these changes.

Simutools 2014, March 17-19, Lisbon, Portugal

DOI 10.4108/icst.simutools.2014.254789

When normal variables are used in parallel programs, the programmer is responsible for coordinating access to each variable state, usually by using locks or other synchronization primitives. In a functional language, due to immutability, when a function f is called with parameters a, b and c, returning d, the programmer can rely that these values will not be changed inside f, as f receives only a copy of these parameters. This property does not exist with reference types in object oriented languages. Thus, in a standard object oriented language, if a, b and c are objects passed as arguments to method $o.f$, the method implementation can change its values internally. More over, in a multi-threaded environment, a, b and c can be accessed by other threads and have their values changed during the execution of $o.f$. Whether $o.f$ changes any of these values or not is implementation dependent and requires programmer knowledge of the operations executed by that method, violating the object orientation principle of implementation encapsulation[10].

To solve this problem, we developed SimVars. A SimVar is a class that controls access to its values, taking care of concurrency and immutability. We defined a SimVar as a generic class, SimVar<T> with two main methods T Get(double time) and Set(double time, T value). The main task accomplished by SimVar is to keep a matrix with the simulation variable values. This matrix has two columns, one for the time and another one for the value at that time.

$$SimVar < int > X = \left| \begin{array}{cc} 0.0 & 1 \\ 1.5 & 3 \\ 2.0 & 9 \end{array} \right|$$

Where X is the SimVar<int> object with three values. We can also see the state of X as a list of tuples $(0.0, 1)$, $(1.5, 3)$, $(2.0, 9)$. For example, we could implement a model with a driver and a car using two SimEntities. The Driver SimEntity would reference the Car SimEntity to request its position, speed and acceleration. Each of these variables would be implemented as a SimVar. An external reference has no access to the *Set* method of a SimVar. To access the Car current values, the Driver has two options: send events to Car and wait for the reply, or create references to Car's SimVars. As a reference or event has an associated time parameter, the values of position, speed and acceleration are all relative to the same instant, avoiding the need to coordinate the locks of these two SimEntities. An example of Get method definition for a reference to SimVar<int> would be int Reference.Get<int>(double t), where t is the time.

5. MODEL

To simulate a train, we developed a model composed by a physical train with engine, a set of brakes (electrical and pneumatic), weight, length, and aerodynamic resistance. This model was developed to provide an accurate train movement simulation in different track conditions like curves and elevation (gradient).

This train is driven by a simulated driver, that observes the train movement and follows a speed profile. The speed profile is a set of speed goals along the voyage. The driver increases and decreases the train engine traction throttle and is also responsible for braking when necessary. The speed profile imposes a hard speed limit on the train speed, so the speed should not exceed the maximum allowed speed more than +1%. The driver should predict a slower speed zone before arriving on it.

The simulation was set to run with 50 ms interval between its cycles, allowing a fine grained control of train speed and driver interaction.

6. EXPERIMENTS

To test the applicability of SimVars, the model was implemented as a set of SimEntities and internal parameters were developed as SimVars. Most of these variables are double precision floating numbers, but others are integers and even segment tables. The objective of the experiment was to test simulation deterministic results with an increasing number of trains and different numbers of processors. As SimVars keep track of all changes during the simulation run, we also observed the amount of memory used in each execution.

The number of processors and trains to be used in each simulation run was changed and runs were executed with 50, 100, 500, 1000 trains with 1, 2, 4 and 8 processors. All testes were run on a standard notebook equipped with an Intel i7 8 core processor (i7-2670QM 2.2GHz). As this processor provides 8 logical processors in 4 cores, we also tested the speed difference with a different combination, avoiding allocation of 2 processors in the same core. The results are presented in Figure 1 and Figure 2.

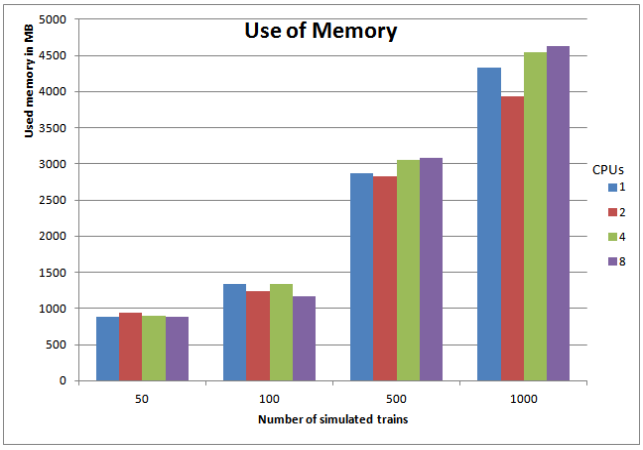

Figure 1: Use of memory x number of trains

These figures show a performance increase (around 260%, for 1000 trains, for 1 and 8 cores) between executions with different number of cores. The code we used to develop the model is standard C# classes, implementing SimEntities and using SimVars. No special code regarding concurrency was written and SimEntities don't have to care about locking. SimVars can be shared between different SimEntities, but only one Entity should be in charge to update its values. This constraint can be relaxed, but then a list of concurrent updates would be received, making the implementer responsible to decide how to handle their simultaneity. We can also see different amounts of used memory, depending on the number of cores used, probably due to how the garbage collector reacts to memory pressure and decides when to free unused memory.

Memory usage is also a big concern, as it increases very fast with the number of trains. Depending on the applica-

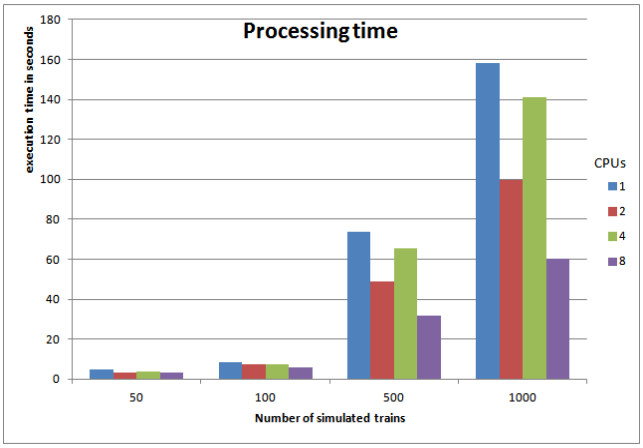

Figure 2: Processing time

tion, the number of previous values kept by SimVars can be reduced, allowing better performance. The main advantage of keeping these states is the ability to pause the simulation or to roll it back to a different simulation time. These scenarios are particularly interesting regarding driving simulation optimization, for example, to save energy, where the same simulation has to be run many times from the same point. The difference in memory usage on multi-core runs is due to a different garbage collection policy, were multi-core runs can also release memory in parallel.

7. CONCLUSION

SimVars were used to implement a simulation model dedicated to railway equipment test and certification, where a set of norms and regulations have to be implemented. Most of these norms have timeouts, events by time and also by displacement. We successfully implemented all these simulation entities without any locking coordination. All entities are computed in a single Simulate call, processed by a parallel for structure. We could implement this new simulator in seven months and we think that the fact of using SimVar as a simulation primitive in our simulator let us focus on physical problems related to the simulation itself at the same time allowing the use of a standard programming language like C#.

Although a performance increase of 260% was obtained, it was far off the expected 800%, theoretical linear increase (8 times the performance of one processor). We can explain this difference by the fact that 8 cores are not completely independent processors (neither full cores, due to Hyper Threading), sharing many structures and optimized to process local data [2]. As the performance increase was obtained without any change on the source code, we can say that SimEntities are a good solution to automatically scale small to medium simulations.

8. REFERENCES

[1] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, KTH, Microelectronics and Information Technology, IMIT, 2003.

[2] A. Binstock. Multi-core processor architecture explained. http://software.intel.com/en-us/articles/multi-core-processor-architecture-explained, Oct. 2010. [Online; accessed 3-February-2014].

[3] A. Bloss. A functional approach to simulation programming. In *Winter Simulation Conference, 1990. Proceedings.*, pages 214 –219, dec 1990.

[4] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985.

[5] A. Ermedahl, S. R. Virding, and E. Ab. Discrete event simulation in Erlang, 1995.

[6] R. M. Fujimoto. *Parallel and Distribution Simulation Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1999.

[7] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7:404–425, 1985.

[8] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[9] V. Pankratius, F. Schmidt, and G. Garreton. Combining functional and imperative programming for multicore software: An empirical study evaluating scala and java. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 123 –133, june 2012.

[10] M. L. Scott. *Programming Language Pragmatics, Second Edition*. Morgan Kaufmann, Nov. 2005.

[11] L. Toscano, G. D'Angelo, and M. Marzolla. Parallel discrete event simulation with Erlang. *CoRR*, abs/1206.2775, 2012.