Distributed Simulation of Large Computer Systems

Moreno Marzolla
INFN Padova and Dept. of Computer Science, Univ. of Venice, Italy
Email: moreno.marzolla@pd.infn.it

Abstract

Sequential simulation of large complex physical systems is often regarded as a computationally expensive task. In order to speed-up complex discrete-event simulations, the paradigm of Parallel and Distributed Discrete Event Simulation (PDES) has been introduced since the late 70s.

In this paper we'll analyze the applicability of PDES to the modeling and analysis of large computer system; such systems are increasingly common in the area of High Energy and Nuclear Physics, because many modern experiments make use of large "compute farms". Some feasibility tests have been performed on a prototype distributed simulator.

Keywords: Parallel and Distributed Discrete Event Simulation, Compute Farms, Performance Evaluation

1 Introduction

Modeling and analysis of complex physical systems is of wide interest in various fields of science and engineering. Large computer systems, such as the "Compute Farms" used for High Energy Physics applications, are an example of such complex systems [7]. An analytic modeling can be performed usually on simple cases only, or making some restrictions or assumptions on the underlying physical systems [1]. In many cases, simulation is the only tractable approach; conducting simulation experiments, however, can be very time consuming, especially when the number of physical processes to simulate is large. Compute farms often consist of hundreds of computing nodes, each one being composed of smaller units (one or more CPU, usually several disks and other adapters); each node interacts with the others in several ways, and also accesses shared resources (disks, Mass Storage Systems, Database Servers...). A deep understanding of the behavior of the complete system under different load conditions or assuming different hardware components can be extremely useful in order to maximize the throughput of the compute farm. Sequential simulations of such systems are computationally very expensive, since a typical simulation run may require several hours to complete, even on powerful computers. Sometimes the simulation has to be repeated several times, each time with slightly different parameters whose values depend on the previous runs, so it's not possible to simply run in parallel different sequential simulations. A possible approach is to run the simulation on a parallel machine, or on a cluster of workstation. In this cases we are talking about *Parallel and Distributed Discrete Event Simulations* (in short, PDES) [3]. The main general protocols for distributed simulations will be described briefly in the following sections.

2 Simulation using Logical Processes

Let us assume that the system under study can be modeled as a collection of *Logical Processes* (LPs), which evolve by exchanging timestamped messages. Each LP contains a portion of the state of the global simulated system, and also contains a local clock telling how far it has advanced in simulated time. It is supposed that events happen at *discrete* points in simulated time. In order for the simulation to be correct, it is necessary to verify that *causality violations* do not occur. This means that the state of the system at simulated time t cannot be influenced by any message having timestamp greater than t. In other words, the current state of the system

at each point in simulated time cannot be influenced by future events. It is possible to ensure that no causality error arises in distributed simulations by forcing each LP to process incoming messages in nondecreasing timestamp order. This constraint is a sufficient but not necessary condition for avoiding causality violations, because if events e_i and e_j are not causally related (they don't influence each other), it is safe to process them in any order. Synchronization protocols used for PDES fall in two categories: *conservative* [2, 6] and *optimistic* [5]. Conservative approaches strictly avoid causality violations, usually allowing to process only events such that every other event that could potentially affect them has been processed. Optimistic approaches use a detection and recovery strategy: causality errors are detected, and a rollback procedure is invoked to return to a consistent state. There is no clear consensus on which policy works better in practice; performances are usually application-dependent. However, conservative protocols are generally simpler to implement, so we decided to concentrate on conservative synchronization models, which we will illustrate in more detail in the next section.

3 Conservative simulation in detail

A conservative distributed simulator is composed by a number of interacting processes, each one simulating a portion (called *region*) of the whole simulation space. Each process has an incoming and outgoing channel for each other process from which it respectively receives and to which sends messages. Messages are enqueued in nondecreasing timestamp order. The core of the process is an almost sequential simulation engine, composed of status variables, a nonnegative real variable containing the *Logical Virtual Time* (LVT), and a list of scheduled events, the *Future Event List* (FEL). At each iteration the process blocks until there is at least one message on each input channel; then the event having smaller timestamp is chosen among those in the FEL and in the input queues. Such event is removed and processed, possibly altering the LVT, the FEL and causing one or more messages to be put in the output queues.

It should be observed that the algorithm, as described, is prone to deadlock, because a loop of processes may exist such that each process is blocked waiting messages from its predecessor in the loop. This is a well-known problem in conservative distributed simulators, and many solutions have been proposed [9]. One of the simplest is a policy of *deadlock avoidance* in which care is taken so that deadlock never forms. Each LP is required to send *null messages* to every outgoing channel in order to signal other LPs that it will *not* send any real message until the timestamp of the null message. In other words, a null message is a promise that the originator will not send any message until the specified timestamp. Null messages require that the system exhibits a certain degree of *lookahead*, that is, the ability of predict future actions, which is actually the case for many real-world systems.

4 Implementation details

A prototype of a conservative, distributed discrete event simulator has been developed. The simulator is written in C++, to ensure portability, and makes use of the MPICH library version 1.2.1 [8] for communication and process allocation. MPICH is a freely available, portable implementation of MPI, the standard for message-passing libraries. The simulator itself is composed as a hierarchy of C++ classes. The root of the hierarchy is the abstract class `lp`, which represents a generic Logical Process. Its interface has some public methods, the most important ones being:

- `initialize( )` (pure virtual), used to initialize the LP object immediately after it is created
- `simulate( simTime_t theTime )` (pure virtual) which every subclass must define, and which is used to advance the state of that LP to simulated time `theTime`; `simTime_t`

represents an instant in simulated time.

- `simTime_t lookAhead( )` (virtual), by default returns zero (no lookahead). This should be redefined for those classes implementing logical processes to reflect how far in the future the current LP can predict *not* to send any message, starting from its current simulation time.

LPs can have a hierarchical structure, that is, a single "composite" LP can be made of a number of simpler LPs. This can be easily achieved by using a *composite pattern* [4] to describe the LPs' structures. Composite LPs are implemented by the `compositeLp` subclass, while "simple" LPs (the leaves of the hierarchy) are implemented as specializations of the `simpleLp` subclass.

The experiments were conducted on a small cluster composed of 3 dual-CPU machines, each CPU being a Pentium II running at 233MHz. Each machine was equipped with 128MB of RAM, and was connected to the other two with a 100Mbps Ethernet controller. The Operating System was Linux 2.2.19-SMP.

5 Experimental Results

Two sample systems have been simulated. The first one is a *pipeline* system of N elements (see figure 1(a)), where there are $N-2$ service nodes connected as a list, plus a source and sink respectively at the start and end of the list. The simulated system has a total of $N = 480$

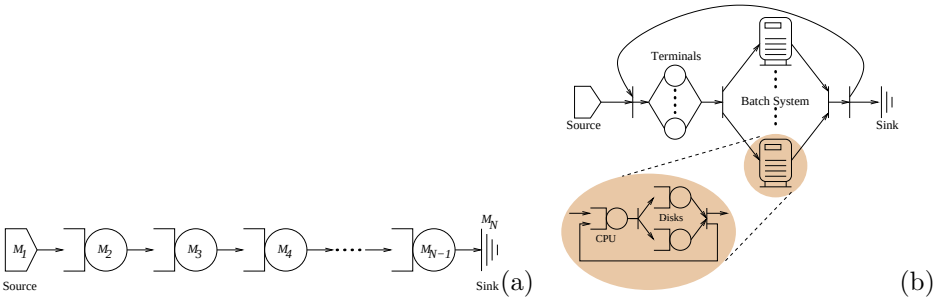
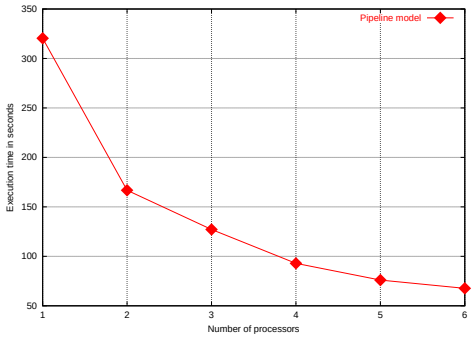


Figure 1: (a) Pipeline test model, (b) Batch system test model. We use the standard pictorial notation for queueing networks, see [1]

nodes. Messages were generated at the source every one time unit, and the processing at each node required one time unit. The system was simulated up to time $T = 10^4$ in such a way that, on k processors, N/k adjacent nodes were assigned to each processor. The execution times of this simulation, using one to six processors, are plotted in figure 2 The second experiment is the simulation of a system having interactive users sending jobs to a pool of batch machines (see figure 1(b)). The system is modeled as a *central server*, where the users are in this case the server, and they send jobs to a pool of N workstations after some *think time*. Each workstation is in turn represented as a central server, with a CPU and two disks. The system has $N = 60$ workstations, and was simulated up to time $T = 2000$. The system has been simulated on $k \geq 2$ processors by dividing the simulation of the N workstations on $k-1$ processors. The remaining processor simulates the terminal, source and sink nodes. The results are depicted in figure 3.

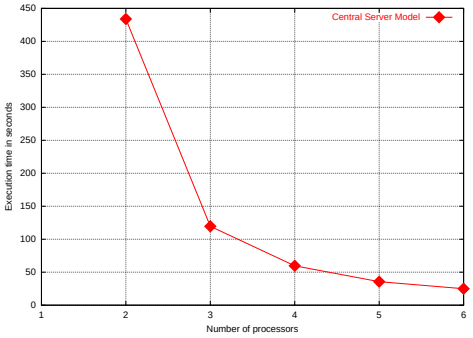
6 Conclusions

In this paper we introduced the main ideas behind Distributed Discrete Event Simulation. This technique can be used to efficiently perform complex simulations, by subdividing the simulation task between multiple processors. Given the encouraging results from our early



CPUs	Time (s)	Speedup
1	320.44	1
2	166.81	1.92
3	127.15	2.52
4	92.92	3.44
5	76.04	4.21
6	67.66	4.73

Figure 2: Pipeline simulation results. In the table, the *Speedup* column is calculated as $T(1)/T(n)$, where $T(j)$ is the time required to perform the simulation on j processors, $1 \leq j \leq 6$



CPUs	Time (s)
2	433.92
3	119.53
4	59.69
5	35.64
6	24.95

Figure 3: Batch system simulation results

tests, our goal is to continue the development of the prototype used for the tests presented here into a fully functional and effective tool, which will be used for the simulation of complex real-world computer systems.

References

[1] G. Bolch, S. Greiner, H. de Meer and K. S. Trivedi, Queueing Networks and Markov Chains : Modeling and Performance Evaluation With Computer Science Applications, John Wiley & Sons, 1998

[2] K. M. Chandy and J. Misra, Asynchronous Distributed Simulation via a Sequence of Parallel Computations, Comm. ACM, 24, 11(1981)

[3] R. M. Fujimoto, Parallel Discrete Event Simulation, Comm. ACM, 33, 10(1990)

[4] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, Addison-Wesley, 1995

[5] D. R. Jefferson, Virtual Time, ACM Trans. on Programming Languages and Systems, 7, 3(1985)

[6] J. Misra, Distributed Discrete-Event Simulation, ACM Comp. Surv., 18, 1(1986)

[7] The MONARC project, `http://monarc.web.cern.ch/MONARC/`

[8] MPICH Home Page, `http://www-unix.mcs.anl.gov/mpi/mpich/`.

[9] D. M. Nicol, Principles of Conservative Parallel Simulation, Proc. of the 1996 Winter Simulation Conference