Distributed Systems Project Report

Distributed Simulation

Members:
Arpit Kumar
Sachin Kumar
Siddharth Rakesh
Manav Sethi

Supervisor:
Prof. Arobinda Gupta

April 14, 2015

Contents

1 Introduction

Distributed simulation involves the execution of a single simulation program on a collection of loosely coupled processors (e.g., PCs interconnected by a LAN or WAN). Distributed simulation is used for a variety of reasons, including:

- Enabling the execution of time consuming simulations, that could not otherwise be performed (e.g., simulation of the Internet), which helps in reducing the model execution time (proportional to the number of processors) and provides the ability to run larger models (with more memory).
- Enabling the simulation to be used as a forecasting tool in time critical decision making processes (e.g., air traffic control), wherein the simulation can be initialized to the current system state, and faster than real-time execution can be achieved for what-if experimentation, as the simulation results may be needed in seconds.
- Creating distributed virtual environments, possibly including users at distant geographical locations (e.g., training, entertainment), providing real-time execution capability, scalable performance for many users and simulated entities.

2 Objectives

The objective of this project is to develop a distributed simulation framework for simulating problems such as random walks in a graph. It includes multiple components, which can be represented by the following points:

- Ensuring a near equal workload distribution among multiple nodes, and distributing the graph in a manner, so as to minimize inter-node communications.
- Developing a reliable, FIFO communication framework among the nodes.
- Developing a well-coordinated framework for message handling and processing using multi-threading.
- To design methods for assessing how well the entire system performs.

3 Literature survey

Simulation is the imitation of the operation of a real-world process or system over time. The act of simulating something first requires that a model be developed; this model represents the key characteristics or behaviours/functions of the selected physical or abstract system or process. The model represents the system itself, whereas the simulation represents the operation of the system over time.

Computer Simulations of real world process provides a means to test out different strategies in order to determine which will be the most effective. Modelling complex real world systems has become very essential in many fields especially in engineering, health, math, military, social and telecommunication sciences. It is very low cost method of information gathering before decision making.

Computer Simulation are primarily of two types,

Continuous Simulation: Continuous Simulation refers to a computer model of a physical system that continuously tracks system response according to a set of equations typically involving differential equations. Typical use cases include rocket trajectory mapping, electrical circuits and robotics.

Discrete Event Simulation: A discrete-event simulation (DES), models the operation of a system as a discrete sequence of events in time. Each event occurs at a particular instant in time and marks a change of state in the system. Between consecutive events, no change in the system is assumed to occur; thus the simulation can directly jump in time from one event to the next.

DES consists of a collection of techniques which, when applied to the study of a discrete event dynamical system, generates sequences called sample paths that characterize its behaviour.

Computer simulations are very attractive as they can compress time simulating years of activity in a matter of minutes or even seconds.

They also help us to replicate experiments. To replicate means to rerun an experiment with selected changes in parameter values or operating conditions made by the investigator.

Despite the sophisticated techniques used, some simulation may take hours to complete in real time due to limited availability of resources (processor or memory) in a single computer. This leads to slow simulations while the decisions may be required to be made in minutes in some cases.

To deal with this parallel and distributed simulations came into being.

3.1 Parallel/Distributed Simulations

This is a technology that enables a simulation program to be executed on parallel/distributed systems, namely systems composed of multiple interconnected computers. As the definition suggests, there are two key components to this technology (1) simulation and 2) execution on parallel or distributed computers. Although parallel and distributed simulations differ from each other in many regards they offer similar benefits. Parallel simulation executes on a set of computers confined to a single cabinet or shelf whereas distributed simulation executes on systems that are geographically distributed across a building, a university campus or even the world.

Following are the benefits of executing a program on multiple computers:

- **Faster Execution**: By subdividing a large simulation computation into many sub-computations, and executing the sub-computations concurrently across, say, ten different processors, one can reduce the execution time up to a factor of ten.

In computer simulations it may be necessary to reduce execution time so that an engineer will not have to wait long periods of time to receive results produced by the simulation. Alternatively, when used to create a virtual world into which humans will be immersed, multiple processors may be needed to complete the simulation computation fast enough so that the simulated world evolves as rapidly as real life. This is essential to make the computer-generated world "look and feel" to the user just like the real thing

- **Geographical Distribution**: Executing the simulation program on a set of geographically distributed computers enables one to create virtual worlds with multiple participants that are physically located at different sites. Participants in such a simulation can interact with each other as if they were located together at a facility at a single site, but without the time, expense, and inconvenience of travelling to that site.

- **Integrating simulators that execute on machines from different manufacturers**: If we have multiple simulators designed by different manufacturers, rather than porting these programs to a single computer, it may be more cost effective to "hook together" the existing simulators, each executing on a different computer, to create a new virtual environment. Again, this requires the simulation computation to be distributed across multiple computers.

- **Fault tolerance**: Another potential benefit of utilizing multiple processors is increased tolerance to failures. If one processor goes down, it may be possible for other processors to pick up the work of the failed machine, allowing the simulation computation to proceed despite the failure. By contrast, if the simulation is mapped to a single processor, failure of that processor means the entire simulation must stop.

3.2 Underlying technologies

- The first key ingredient is an inexpensive computer, thereby making systems composed often of hundreds, or even thousands of computers economically feasible.

- *Inter-computer communications.* There are two flavours of technology at work here. On the one hand, high-speed switches enable one to construct systems containing tens to hundreds or even thousands of processors that reside within a single cabinet or computer room. On the other hand, advances in fibre optics technology is fuelling a revolution in the telecommunications industry, making possible computing systems distributed across continents. These advances enable one to consider developing computer applications utilizing many geographically distributed machines.

- *Modelling and simulation.* The final ingredient are technologies to enable construction of models of actual or envisioned real-world systems that can (1)

be represented in the internal storage of a computer, and (2) be manipulated by computer programs to emulate the evolution of the actual system over time.

Distributed Simulation over Parallel Simulation

Distributed computers cover a much broader geographic area. Their extent may be confined to a single building, or may be as broad as across an entire nation or even the world. Unlike parallel computers, each node of a distributed computer is usually a stand-alone machine that includes its own memory and I/O devices. Commercial off-the-shelf personal computers or engineering workstations, often from different manufacturers, are usually used.

3.3 Notions of time in simulation

There are several different notions of time that are important when discussing a simulation.

- *Physical time* refers to time in the physical system.
- *Simulation time* is an abstraction used by the simulation to model physical time.

  It is defined as a totally ordered set of values where each value represents an instant of time in the physical system being modelled. Further, for any two values of simulation time T_1 representing physical time P_1 and T_2 representing P_2, if $T_1 < T_2$, then P_1 occurs before P_2, and $(T_2 - T_1)$ is equal to $(P_2 - P_1) * K$ for some constant K. If $T_1 < T_2$, then T_1 is said to occur before T_2, and if $T_1 > T_2$, then T_1 is said to occur after T_2.

  The linear relationship between intervals of simulation time and intervals of physical time ensures durations of simulation time have a proper correspondence to durations in physical time
- *Wall-clock time* refers to time during the execution of the simulation program.

  A simulation program can usually obtain the current value of wall-clock time (accurate to some specifiable amount of error) by reading a hardware clock maintained by the operating system.

3.4 Conservative Synchronization Algorithms

The physical system being modelled is viewed as being composed of some number of physical processes that interact in some fashion. Each physical process is modelled by a logical process (LP), and interactions between physical processes are by exchanging time-stamped messages between the corresponding logical processes. The computation performed by each LP is a sequence of event computations, where each computation may modify state variables and/or schedule new events for itself or other LPs.

But, to ensure proper execution, Each logical process must process all of its events, both those generated locally and those generated by other LPs, in time stamp order. Failure to process the events in time stamp order could cause the computation for one event to affect another event in its past, clearly an unacceptable situation.

Errors resulting from out-of-order event processing are referred to as causality errors, and the general problem of ensuring that events are processed in a time stamp order is referred to as the *synchronization problem.*

Conservative synchronization protocols ensure that each LP strictly avoids processing events out of time stamp order. In the next section, we describe an alternative approach called *optimistic synchronization* where errors are detected during the execution, and some mechanism is used to recover from them.

Synchronization algorithms do not need to actually guarantee that events in different processors are processed in time stamp order but only that the end result is the same as if this had been the case.

The state of the entire simulator must be partitioned into state vectors, with one state vector per LP. Each logical process contains a portion of the state corresponding to the physical process it models, as well as a local clock that denotes how far the process has progressed in simulation time.

A conservative algorithm solves the problem of when it is "safe" to execute an event. More precisely, If a process contains an unprocessed event E with time stamp T (and no other with smaller time stamp), and that process can determine that it is impossible for it to later receive another event with time stamp smaller than T , then E is said to be safe because one can guarantee that processing the event now will not later result in a violation of the local causality constraint.

3.5 A Deadlock Avoidance Conservative algorithm

Let us assume that one statically specifies the links that indicate which logical processes may communicate with which other logical processes. Further assume that (l) the sequence of time stamps on messages sent over a link is non decreasing, (2) the communications facility guarantees that messages are received in the same order that they were sent (software to re-order messages is necessary if the network does not guarantee this property), and that (3) communications are reliable (i.e., every message that is sent is eventually received). This implies that the stream of messages arriving on a given link will have non-decreasing time stamp values. It also guarantees that the time stamp of the last message received on an incoming link is a lower bound on the time stamp of any subsequent message that will later be received on that link.

Because messages in each FIFO queue are sorted by time stamp, the LP can guarantee adherence to the local causality constraint by repeatedly processing the message containing the smallest time stamp, so long as each queue contains at least one message If one of the FIFO queues becomes empty, the LP must wait until a new message is added to this queue because a message could later arrive that contains a time stamp as small as the message it just removed and processed from

that queue. This will never break causality constraint but may lead to deadlock as some process may keep on waiting for each other forever to send a message but neither will send any message.

We can either avoid the deadlock or may have a separate routine which detects the deadlock and breaks it. A deadlock avoidance method will require additional messages called null messages. It is required that an LP indicates to other LPs a lower bound on the time stamp of messages it will send that LP in the future. Null messages can be used for this purpose. Null messages are used only for synchronization, and do not correspond to any activity in the physical system.

Algorithm 1 Conservative Null Message Deadlock Avoidance Pseudo Code

```
1: procedure SimulateSystem(d1, . . . , dN)
2: while simulation is not over
3:     wait until each FIFO contains atleast one message
4:     remove smallest time stamped message M from its FIFO Queue
5:     clock ← time stamp of M
6:     process M
7:     send NULL message to all LPs with time stamp equal to lower bound on
   time stamp of future messages (current time stamp + look-ahead)
```

An alternative approach to sending a null message after processing each event is a demand-driven approach. Whenever a process is about to become blocked because the incoming link with the smallest link clock value has no messages waiting to be processed, it requests the next message (null or otherwise) from the process on the sending side of the link. The process resumes execution when the response to this request is received. (**We have implemented this method in our framework for simulating a network as described in later sections**, where it is described in more detail)

3.6 Optimistic Synchronization Algorithms

Conservative synchronization algorithms avoid violating the local causality constraint, whereas optimistic algorithms allow violations to occur but provide a mechanism to recover. The term optimistic execution refers to the fact that logical processes process events, "optimistically" assuming there are no causality errors.

4 Dataset

- A small synthetic dataset
- A random graph of 100 nodes
- Co-authorship network with 1000 nodes
- Co-authorship network with 100000 nodes

5 Approach

5.1 Distributing the graph

Algorithms that find a good partitioning of highly unstructured graphs are critical for developing efficient solutions for a wide range of problems in many application areas on both serial and parallel computers.

For example, large-scale numerical simulations on parallel computers, such as those based on finite element methods, require the distribution of the finite element mesh to the processors. This distribution must be done so that the number of elements assigned to each processor is the same, and the number of adjacent elements assigned to different processors is minimized.

The goal of the first condition is to balance the computations among the processors. The goal of the second condition is to minimize the communication resulting from the placement of adjacent elements to different processors. Graph partitioning can be used to successfully satisfy these conditions by first modelling the finite element mesh by a graph, and then partitioning it into equal parts.

We have used gpmetis, an open source application which uses multilevel k-way partitioning algorithms to compute a partitioning solution in which each partition is contiguous. The partitioning and ordering routines compute multiple different solutions and select the best as the final solution. Our objective function is Edge-cut minimization. The command-line programs provide full access to the entire set of capabilities provided by gpmetis' API.

5.2 Establishing communication between the nodes

Our framework requires reliable communication because receiving messages correctly is imperative for the proper functioning of the system. FIFO channels are required because each processor maintains a set of event queues for every other machine and a local queue, and we assume that a new message in an event queue has a time stamp, not smaller than the last event.

In order to ensure such reliable, FIFO communication, we use TCP connections between the nodes. Each node in the system acts as a server and client. While acting as the client, it sends connection requests to all other nodes, using their IP addresses stored in a configuration file before the application begins execution. Simultaneously, it acts as a server and accepts connections from the other nodes, which try to connect to it.

Thus, we obtain a fully connected topology wherein each node has a bidirectional connection to every other node via TCP connections.

5.3 Developing the message handling and processing framework

For n nodes in the system, a total of $n + 3$ threads were used for the basic processing in our framework. After one main thread, there were $n - 1$ threads for

receiving messages(events) from other systems in our simulation framework, one thread for sending messages(events) to other systems, one thread for the processing of events and generating new events, and one thread for managing all the receive threads.

We have followed conservative algorithm for distributed simulation sending $NULL$ messages via a demand driven approach. Whenever a system needs a safe time from another system it sends a $DEMAND$ message to the system and expects a $NULL$ message from the other system.

Each system maintains n event queues one local queue and $n-1$ queues for each incoming connection and one send queue for sending events to other systems. They also maintain a safe time for all other systems. The safe time T maintained by system i is a vector of $n-1$ timestamps, t_1, t_2,.. t_{i-1}, t_{i+1}, ..., t_n. System k generates new events greater than t_k maintained by other systems.

Essentially system k sends $NULL$ messages to other system with this time as a guarantee that it will not generate events with a previous timestamp. Since we have used TCP we connection we have a reliable FIFO channel for message transfer so before a $NULL$ message reaches a system all the messages sent before it reaches the system.

Main thread: Initializes the data structures needed and manages other threads.

Process event thread: Processes the events in logical order from the event queues. Newly generated event from this thread are either placed in local event queue or send queue(if the event generated is for other system).

Receive manager thread: Spawns the other receive threads and manages them (e.g. socket time-out, reconnection etc.)

Receive threads: Received events from the TCP channel are placed in appropriate event queue by this thread.

Send thread: Takes the events from send queue and sends it to appropriate system.

5.4 Designing system assessment methods

To test the system and its capabilities, we have simulated the random walk problem. A random walk is a mathematical formalization of a path that consists of a succession of random steps. For example, the path traced by a molecule as it travels in a liquid or a gas, the search path of a foraging animal, the price of a fluctuating stock and the financial status of a gambler can all be modelled as random walks, although they may not be truly random in reality.

For testing our system, we began by seeding approximately 10 percent of the nodes with random walkers. If any node achieves 1000 visits, we terminated the system. At the end, we computed the steady state probability that a node is visited in a random walk using the number of times it was actually visited during the walk.

1. **Sending/Receiving messages**: Different text messages were injected in each processor which were meant to be sent to every other processor. We

asserted that these messages reached the correct processors. This also verifies the correct working of send and receive queues.

2. **Synchronization algorithm**: The conservative algorithm involving the use of $DEMAND$ and $NULL$ messages was verified using the Random walk simulation. All the processors kept simulating without any deadlock. They kept on generating new events, both internal as well as external. $DEMAND$ and $NULL$ messages were generated at correct times as verified by the Message log.

3. **Timestamp Ordering**: Each processor maintains a logical clock with non-decreasing timestamps. It increases with local, send and receive events. Each processor also maintains safe times for event queues corresponding to other processors which is also non-decreasing. It increases with $NULL$ messages. The correct working of these timestamps was verified from the log of the Random Walk Simulation.

6 Results Obtained

Based on the tests performed, we computed the Normalized Root-Mean-Square error between the observed and theoretical steady state probabilities. The error obtained is 8.18 percent.

The simulation lasted for approximately 5 minutes. No deadlocks were observed during the entire simulation.

Although we have restricted the number of iterations, we observe that the actual results closely match the theoretical steady state probabilities. This is a clear indicator of the correctness of the entire distributed framework.

7 Conclusion and Future work

We have developed a distributed simulation framework which generates a near-equal workload distribution among the nodes; establishes reliable, FIFO communication links between them and coordinates the message protocols between the nodes and internal processing using multi-threading.

Currently, we rely only on local termination conditions for the termination process to initiate. We wish to introduce global consensus protocols for termination to our system. This will ensure that local conditions being met in some part of the system do not force the application termination for other parts as well.

We also plan to introduce multiple message protocols to our system, as opposed to the current single message protocols. This will allow an even greater level of functionality to be achieved through our system.

References

[1] Parallel and Distributed Discrete Event Simulation: Algorithms And Applications by Richard M. Fujimoto, Proceedings of the 1993 Winter Simulation Conference
http://dl.acm.org/citation.cfm?id=256596

[2] Distributed Discrete-Event Simulation by Jayadev Misra
http://dl.acm.org/citation.cfm?id=6485

[3] Distributed simulation of networks by K. M. Chandy, Victor Holmes and Jayadev Misra
http://www.cs.utexas.edu/users/misra/scannedPdf.dir/DistributedSimulationNetworks.pdf

[4] A fast and high quality multilevel scheme for partitioning irregular graphs by G. Karypis and V. Kumar, SIAM J. Sci. Comput., 20:359392, December 1998.
http://libra.msra.cn/Publication/1318048/a-fast-and-high-quality-multilevel-scheme-for-partitioning-irregulargraphs

[5] Integrated Fluid and Packet Network Simulations by George F. Riley, Talal M. Jaafar and Richard M. Fujimoto
http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1167114

[6] Parallel Simulation of Telecommunication Networks http://titania.ctie.monash.edu.au/pnetsim.html

[7] Introduction to Discrete-Event Simulation and the SimPy Language by Norm Matloff
http://heather.cs.ucdavis.edu/~matloff/156/PLN/DESimIntro.pdf

[8] The OMNET++ Discrete Event Simulation System by Andrs Varga
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.331.1728&rep=rep1&type=pdf

[9] EpiSimdemics: an Efficient Algorithm for Simulating the Spread of Infectious Disease over Large Realistic Social Networks by Christopher L. Barrett, Keith R. Bisset, Stephen G. Eubank, Xizhou Feng, Madhav V. Marathe, Network Dynamics and Simulation Science Laboratory, Virginia Tech, Blacksburg
http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5214892