2012 International Conference on Innovation and Information Management (ICIIM 2012)
IPCSIT vol. 36 (2012) © (2012) IACSIT Press, Singapore

Optimum Dynamic Load-Balancing Algorithm for Synchronous Parallel Simulation

Xue Hui Wang [1,a], and Lei Zhang [1,b +]

[1] 601 Staff room, School of Computer, National University of Defense Technology, Changsha, P.R. China

Abstract. Parallel simulation is used to study the behaviors of complex systems. These systems often exhibit irregular load characteristics that change over time and are difficult to predict accurately. Most parallel simulation systems employ user-defined static partitioning. However, static partitioning requires in-depth domain knowledge of the specific simulation model in study. Contrarily, dynamic load-balancing allows the simulation system automatically to balance the load of different simulation models without user input. This paper presents an optimum dynamic load-balancing algorithm for synchronous parallel simulation. This algorithm dynamically balances the load on processors in order to reduce the time of simulation, and thus increase the total simulation speed. Also, the simulation processes are allowed to migrate according to the load on the processors.

Keywords: Synchronous Parallel Simulation(SPS), Optimistic Synchronization Algorithm, Load-Balancing, Dynamic Algorithm.

1. Introduction

Parallel simulation is used to study the behaviors of complex systems. These systems often exhibit irregular load characteristics that change over time and are difficult to predict accurately. Partitioning plays an important role in achieving good performance when simulating these systems. It enables the simulation load to be evenly distributed among the processors.

As is well known, the performance of a parallel simulation depends on partitioning the simulation load evenly among the set of processors to ensure load-balance between processors. Most parallel simulation systems rely on users to define the required partition. However, defining a good partition for a complex simulation model often requires in-depth domain knowledge. The dynamic behaviors of the model also mean that a good static partition at the start of a simulation run may become less effective over time.

So static partitioning is not effective if the load of a simulation model cannot be quantified accurately or changes over time during a simulation run. Therefore we would rather adopt a more flexible means of balancing the load. A dynamic load-balancing algorithm (DLBA) offers the ability to balance the load of the simulation system as the simulation progresses. This can achieve consistent performance given any arbitrary simulation model with unpredictable load characteristics.

2. Synchronization Mechanisms

Time management is concerned with ensuring that the execution of the PDS is properly synchronized, which is particularly important for developer and user. This is because the synchronization not only ensures that events are processed in a correct order in logic, but also helps to ensure that repeated executions of a simulation with the same inputs produce exactly the same results. In other words, the goal of the

+ Corresponding author
E-mail address: [a]findyanzi@126.com, [b]zlmailbox2000@163.com

synchronization mechanism is to make all Logical Process (LP) processed events accord with the local causality constraint (LCC); to do otherwise doubtless results in causality violations.

A parallel simulation consists of a number of Logical Processes that communicate by exchanging time-stamped messages or events, typically each one running on a separate processor. The simulation progress is ensured by the processor scheduling new events to be executed in the future and executing these events in the time-stamp order. A process can schedule an event for itself locally (self-initiation), or remotely for another process. In the latter case, a message is sent via the network to the remote process. Each process maintains a separate time clock, called the Local Virtual Time. Synchronization algorithms can be classified as being either conservative or optimistic. In brief, conservative algorithms take precautions to avoid the possibility of processing events out of time stamp order, i.e., the execution mechanism avoids synchronization errors. On the other hand, optimistic algorithms use a detection and recovery approach. Events are allowed to be processed out of time stamp order, but a rollback mechanism is provided to recover from such errors.

2.1. Conservative Protocol

The principal task of any conservative protocol is to determine when it is "safe" to process an event. An event is said to be safe when there is an indication that no event containing an earlier time stamp will be later received by this LP. In other words, conservative approaches do not allow an LP to process an event until it has been guaranteed to be safe. At the heart of most conservative synchronization algorithms is for each LP to compute a Lower Bound of the Time Stamp (LBTS) of future messages that may later be received by that LP. This allows the mechanism to determine which events are safe to process.

Conservative protocols require each logical process to broadcast to its neighbors, in the form of null messages, a LBTS of events it will send to other logical processes, or Earliest Output Time (EOT). By listening to the null messages from all neighbors, each logical process can determine the lowest timestamp of any events it will receive, or called Earliest Input Time (EIT). If the EIT is larger than the timestamp of the earliest event in its local event list, the logical process is certain that this earliest event can be processed without violating the causality constraint. Otherwise, the logical process has to block execution of events until the earliest local event is safe to be processed.

2.2. Optimistic Protocol

In contrast to conservative approaches that avoid violations of the local causality constraint, optimistic methods allow violations to occur, but are able to detect and recover from them. In any optimistic protocol, a logical process is allowed aggressively to process local events and to send to other logical processes new messages generated by the event execution. However, when an event arrives with a timestamp smaller than the local simulation time, which is called a straggler, a causality error is triggered.

For example, the Time Warp mechanism is well known optimistic method. When an LP receives an event with timestamp smaller than one or more events has already processed, it rolls back and reprocesses events in timestamp order. Rolling back an event involves restoring the state of the LP to that which existed prior processing the event. Time Warp are sent optimistically, i.e., aggressively, or with risk. All processed local events later than the straggler must be undone, and anti-messages must be sent to other logical processes to cancel messages sent during the execution of these events.

3. Synchronization Algorithm on SPS

Synchronous Parallel Simulation (SPS) model has features such as simple programming interfaces, scalable performance and a simplified cost model for performance prediction. The SPS model allows the separation of concerns between the computation cost, synchronization cost and communication cost when designing a parallel algorithm. A SPS programming model consists of P processors linked by an interconnecting network and each with its own pool of memory. The SPS processors communicate with one another by exchanging messages using the inter-connect network.

3.1. The Event Horizon

The event horizon is a concept that can first be understood without referring to parallel processing. Imagine first that all pending events are mapped into a single event queue. As each event is processed, it may generate zero or more new events with arbitrary time stamps.

To maintain complete generality, it is assumed that the simulation engine has no way to predict what each event will do until it is processed. All newly generated events are maintained in a special auxiliary event queue. At some point in time, the next event to be processed will be in the auxiliary event queue. This point in time is called the event horizon.

One of the key features of the event horizon is that, by definition, events in a given cycle are not disturbed by other events generated in the same cycle. If the events were distributed to multiple processors and if the event horizon were known before hand, it would be possible to process events in parallel without ever receiving a straggler message. The event horizon is not determined, however, until the events in the current cycle are actually processed.

Each cycle processes its pending events while collecting newly generated events in an auxiliary event queue. When the next event to be processed is in the auxiliary event queue, the auxiliary event queue is sorted, merged into the primary event queue, and then the next cycle begins. Thus, there is no way for a deadlock situation to occur when processing events in cycles defined by the event horizon.

3.2. Optimistic Synchronization Algorithm

The optimistic synchronization algorithm for the SPS time management is simply shown as Figure 1. Each processor manages a group of logical processes (LPs) in the system. In this algorithm, LPs in the same processor share a common event-list. When the algorithm comes into the outer while loop a series of integrated steps is executed by each processor and the SPS_sync() statement at the end of the loop is carried out.

```
Initial()
While  (GVT < SimEndTime)  do
ReceiveExternalEvents();
RB_DEFINE_TREE();   // ProcessRollback;
 ComputeGVT();        // Compute new GVT
 FossilCollection(); // perform fossil collection
        Ne=ComputeEventHorizon();
//compute new event limit Ne every N integratedstep;
                ExecuteNeEvents();
SPS_sync();
endwhile
```

Fig. 1 Optimistic Synchronization Algorithm for the SPS

The algorithm provides an automatic means of limiting the number of events by event horizon, Ne being simulated per integrated-step based on statistics from fossil collected events. The aim of the algorithm is to complete the simulation in the least number of integrated steps possible.

3.3. Global Virtual Time

For all the high performance demonstrated on shared memory architecture, the cost of state saving and restoration still excessive. During forward execution the logging cost is the greatest part of the overhead. It can explore the parallelism in a system by allowing out of order event execution. Some of these event executions may not be correct and need to be cancelled. When an event with time stamp t arrives in a process and the process already executed an event with a time stamp greater than or equal to t, the current logical process needs to be recovered to the state before time stamp t to be able to correctly execute the new event.

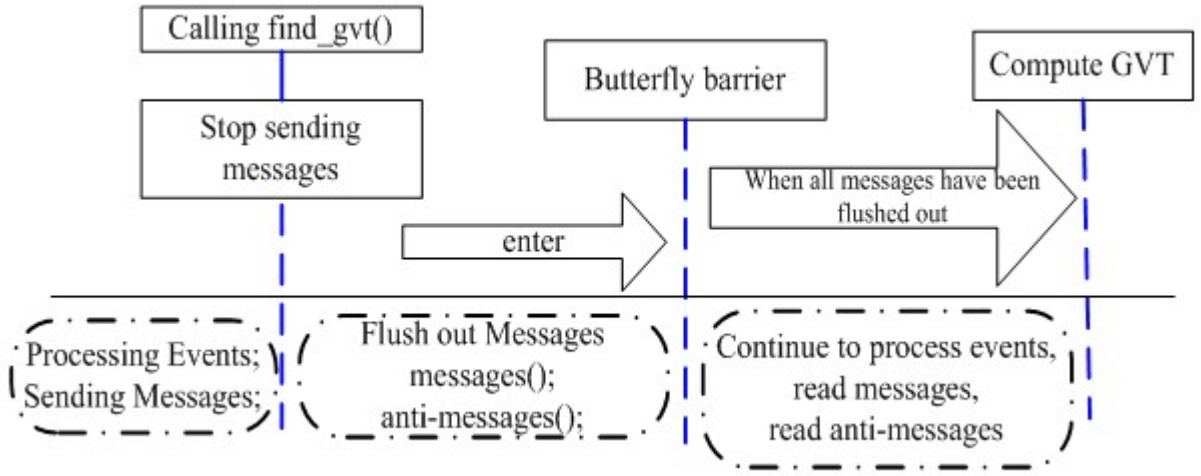

Fig. 2 The Computation of the GVT.

So there are still two problems remaining to be solved. For one thing, certain computations, e.g., I/O operations, cannot be rolled back. For another thing, the computation will continually consume more and more memory resources because a history must be retained, even if no rollbacks occur; some mechanism is required to reclaim the memory used for this history information. Both problems are solved by global virtual time (GVT).

GVT is defined as the minimum time stamp of all events in transit or in the input queues of all logical processes. No new event generated can have a smaller time stamp than GVT. Therefore all states that have smaller time stamp than GVT can be discarded safely. GVT is computed by observing that rollbacks are caused by messages arriving "in the past." Therefore, the smallest timestamp among unprocessed and partially processed messages gives a value for GVT. Once GVT has been computed, I/O operations occurring at simulated times older than GVT can be committed, and storage older than GVT (except one state vector for each LP) can be reclaimed.

The computation of the GVT is shown in Figure 2. The Global Virtual Time (GVT) gives a lower bound on the timestamp of the earliest unprocessed event in the simulation. Any event processed with simulation time earlier than the GVT is committed, in the sense that it will never be rolled back. The global virtual time (GVT) shows the advance of a parallel simulation run. The GVT computation is performed after every n integrated steps; n is also known as the GVT update interval. Memories for events or states in an LP with time-stamps smaller than GVT are reclaimed after each GVT computation. The body of the loop is executed till the processor's GVT value is greater than the simulation end time.

4. Dynamic Load-Bbalancing Algorithm

In this section, a new dynamic load-balancing algorithm that balances both computation and communication workload among the processors is presented.

4.1. Load-Balancing for Computation

The parameter determines if the simulation system is suffering from computation or communication load-imbalance. Computation load-balancing is only activated when the computation load-imbalance, $F(i)$ exceeds the threshold value. Where (P_i , work) is the total computation workload for processor P_i since the last migration point, the function gives the maximum computation workload across all processors. The function gives the average computation workload across all processors. Then $F(i)$ is defined as follows:

$$F(i) = \frac{\max(P_i, work) - \overline{(P_i, work)}}{\overline{(P_i, work)}}$$

The pseudo code for the procedure balance computation is presented as below:

```
pid := sps_pid();
while F(i) > Δ do
AverageW:=(Pmax.work+Pmin.work)*1/2;
   loadMigrate := Pmax.work - AverageW;
   if pid = max then
      send workLoad(loadMigrate,Pmin);
   endif
      Pmax. work := AverageW;
      Pmin. work := AverageW;
Compute F(i);
Endwhile
```

Fig.3 Code for Balance Computation

where Pmax is the processor with the greatest computation workload and Pmin is the processor with the least computation workload. The procedure send_workload() migrates the required amount of computation workload to the destination processor. The selection of simulation objects to migrate gives preference to those with higher computation workload so that fewer simulation objects are migrated. The procedure terminates when $F(i)$ falls below the threshold set by Δ.

Load-Balancing for Communication. Communication load-balancing is only activated when the communication load-imbalance, *M(i)=S(i)+R(i)*exceeds the threshold value. (P_i , com) is the total communication for processor P_i since the last migration point. The function gives the maximum communication across all processors. The function gives the average communication across all processors. Then M(i) is defined as follows:

$$M(i) = \frac{\max(P_i, com)}{\overline{(P_i, com)}} - 1$$

The pseudo code for the procedure balance communication is presented as below:

```
pid=sps_pid();
while M(i) > Δ do
a := ((Pmax.com − Pmin.com)*Pmax.work*Pmin.work) / (2*(Pmax.com*Pmin.work − Pmin.com*Pmax.work));
if pid = max then
        send workLoad(a,Pmin)
    endif
if pid = min then
    send workLoad(a,Pmax)
    endif
  AverageC:=(Pmax.com+Pmin.com)*1/2;
  Pmax:com := AverageC;
  Pmin:com := AverageC;
  compute M(i);
endwhile
```

Fig.4: Code for Balance Communication

The procedure uses exchange of computation workload to balance the communication workload between two processors so as to preserve computation balance. The estimated amount of workload to be exchanged between the processors having the maximum and minimum communication workload is computed using the

equation shown in the algorithm. Both processors then proceed to send the same amount of computation workload to each other. The procedure terminates when M(i) falls below the threshold Δ.

5. Summary

In this paper, the metrics used are the computation and communication workload. The rate of progress in simulation time between integrated-steps for each processor in SPS is automatically controlled by the adaptive event limit set for each integrated-step.

In order to speed up the synchronous parallel simulation on a heterogeneous non-dedicated multiprocessor system, using the cost model for SPS, a new dynamical load balancing algorithm is proposed. This algorithm dynamically balances the load on processors in order to reduce the total execution time, which is shown to be able to achieve consistently better performance over the original SPS optimistic protocol by balancing both computation and communication workload, Our simulation experiment results indicate that this algorithm provides better performance when the difference between processor speeds and/or the variation of workload is large. Further work is needed to improve the computation and communication load-balancing modules to take into account lookaheads between processors when selecting simulation objects for migration.

6. Acknowledgements

Supported by NSFC No. 60803100 and No. 61103190

7. References

[1] K. Iskra: Parallel Discrete Event Simulation Issues with Wide Area Distribution, ASCIcourse a9, March 7, 2003

[2] C. D. Carothers, and R. M. Fujimoto: Efficient Execution of Time Warp Programs on Heterogeneous, NOW Platforms, *IEEE Transactions on Parallel and Distributed Systems*, vol.11, no. 3, pp. 299–317, March 2000.

[3] Metron, Inc., SPEEDES User's Guide, http://www.speedes.com, 2004

[4] X. F. Wu, V. Taylor, C. Lively, *et al.* Performance an Alysis and Optimization of Parallel Scientific Applications on CMP clusters [J] .*Scalab le Computing: Practice and Experience*, 2009, 10(1) : 61-74.

[5] The Top 500 Supercomputer Sites, 2009. Processor G eneration share for 11/2008 [EB/OL]. http: / /www. top500. org/charts/list/30/archtype, 2008-11-1 /2009-05-4.