Parallel discrete event simulation Basics

https://www.eg.bucknell.edu/~xmeng/Course/CS6337/Note/master/node20.html

* Parallel discrete event simulation (PDES) refers to the execution of a single discrete event simulation program on a parallel computer.
* A discrete event simulation model assumes the system being simulated only changes state at a discrete points in simulated time.
* The simulation model jumps from one state to another upon the occurrence of an *event*.
* In a real world system model, many things (events) can occur at the about same time, yet few take place at the exact same moment. Also, they don't have a regular interval in between occurrances.
* *Asynchronous* systems where events are not synchronized by a global clock.
* A possible mechanism of PDES is to use *lock-step* execution using a global simulation clock among many processors. At each step of simulated time, event lists on different processors are checked and the events due in time are executed.
* This approach performs very poorly because very few events would have the exact same time.
* *Concurrent* execution of events at *different* points in simulated time is required! This introduces interesting synchronization problems that are at the heart of the PDES problem.
* Sequential simulations typically utilize three data structures:
  1. the *state variables* that describe the state of the system,
  2. an *event list* containing all pending events that have been scheduled, but have not yet taken effect, and
  3. a *global clock* variable to denote how far the simulation has progressed.
* In this execution paradigm, it is crucial that one always select the smallest timestamped event from the event list as the one to be processed next.
* If an event with larger timestamp were executed before a smaller one that would schedule this larger timestamped event, an logic error would occur. We call this type of errors *causality* errors.

Example: if a customer's departure event is processed before its arrival event, a causality error occured.

* The greatest opportunity in parallel simulation is to process events concurrently on different processors.
* A typical strategy is to map each physical process to a logical process (LP) and each LP proceeds simulation on its own pace.

Example: if we are to simulate a gas station with two independent attendants, they form naturally two LPs, each of which can execute simulation of its own.

* One can ensure that no causality errors occur if one adheres to the following constraint:

***Local Causality Constraint***

A discrete event simulation, consisting of logical processes (LPs) that interact exclusively by exchanging timestamped messages, obeys the local causality constraint if and only if each LP processes events in non-decreasing timestamp order.

The above LCC essentially says if events are processed in non-decreasing timestamp order, then we say it obeys the causality constraint; if events are not processed in non-decreasing timestamp order, then we say it does not obey the causality constraint.

* Adherence to this constraint is sufficient, though not always necessary, to guarantee that no causality errors occur. In another words, violating causality constraint may not always result in simulation error. This is because two events within a single LP may be independent of each other, in which case processing them out of timestamp sequence does not lead to causality error.

Example: a supermarket has a service desk and a number of check-out lines. The customers who go through service desk can be considered independent of those who go through check-out lines. If we process them out of timestamp order, it will not lead to causality error.

On the other hand, if in the same check-out line for a single customer, if we process the *bagging* event before the *check-out* event, a causality error has occured.

* The challenge in PDES is to execute LPs concurrently and lead to correct simulation results.
* PDES mechanisms can be divided into two categories: *conservative* and *optimistic*.

Conservative approaches strictly *avoid* the possibility of any causality error. Typically these approaches reply on some strategy to determine if it safe to process an event.

Optimistic approaches use a *detect and recover* strategy: causality errors are allowed, but detected, and a *rollback* mechanism is invoked to recover the errors.

**Conservative Approach**

* The key to conservative PDES is to make sure no causality error will occur before processing an event. Different strategies exist.
* Statically specify the links that indicate which processes may communicate with which other processes. An LP can send messages only to specified LPs.

E.g.: when simulating the operation of a number of airports with airplanes taking-off and landing, we may speicify that airplanes take off from A can only land on B and C. This way we have a fixed link.

* Messages arriving on each incoming link are stored in FIFO order for the LP to process. Each of the incoming links maintains a clock that is equal to either the timestamp of the message at the head of the queue if the queue is not empty, or the timestamp of the last received message if the queue is empty.
* The LP repeatedly selects the link with the samllest clock and, if there is a message in that link's queue, processes it.
* If the selected queue is empty, the process blocks.
* The above protocol guarantees that each process will only process events in non-decreasing timestamp order, thereby ensuring adherence to the local causality constraint.
* Problems: if a cycle of empty queues arises that all have small clock value, each process in the cycle must block, and the simulation deadlocks. (See Figure 2 on page 34 of Fujimoto's 1990 CACM paper).
* To solve this problem, a *null* message is utilized. An LP sends messages to all of its out-going links in a round-robin fashion. If at a time the LP doesn't have a message for on particular out-going link, a null message with timestamp $T_{null}$ is sent to that link. Doing so guarantees that the receiving LP will not have to block, thus deadlock is avoided.
* The null message from $LP_A$ with timestamp $T_{null}$ essentially tells the reciving process that $LP_A$ will not send events in the future to other LPs with timestamp smaller than $T_{null}$.
* How to compute $T_{null}$? $T_{null}$ is the minimum of all incoming links timestamp and first event's timestamp on LP's own event list.
* This protocol works correctly. However, it generates large number of null messages, thus waste of processing time.
* Large amount of work has been done to improve the performance of conservative approach to PDES.

**Optimistic Mechanisms**

* Optimistic mechanism allow all LPs proceed because it is possible that causality error might not occur.
* If at a later stage an causality error is detected, the recover process takes place, undoing the effects of the events that have been processed (rollback).
* The event causing rollback is called a *straggler*.
* An event may do two things that have to be rolled back:
  1. it may modify the state of the logical process;
  2. it may send event messages to other processes
* Rolling back the state is accomplished by periodically saving the process's state, and restoring an old state vector on rollback. The state has to be rolled back to a simulation time that is equal or smaller than the time of the *straggler*.
* "Unsending" a previously sent message is accomplished by sending a negative message or an *anti-message* that annihilates the original message when it reaches its destination.
* If a process receives an anti-message that corresponds to a positive message that has already been processed, this anti-message becomes a straggler, causing rollback on this LP.
* Recursively repeating this procedure allows all the effect of erroneous computation to eventually be canceled. It can be proved that this process converges, and it always makes progress under certain conditions.
* The fundation of the optimistic approach is *virtual time*, which is one of the most important concepts in distributed computing.
  1. A *virtual time system* is a distributed system executing in coordination with an imaginary *virtual clock* that ticks *virtual time*.
  2. Virtual time itself is a global, one-dimentional, temporary coordinate system imposed on a distributed computation.
     + It is used to measure computational progress and to define synchronization.
     + It may or may not have a connection with real time.
     + It is a real positive value, totally ordered by $<$.
  3. Virtual time systems are subject to two fundamental rules.
     + The virtual send time of each message must be less than its virtual receive time.
     + The virtual time of each event in a process must be less than the virtual time of the next event at that process.

These rules are exactly what Lamport's Clock Conditions (the well-known Happens-Before relation).

* 1. The major constraint on the implementation of virtual time can be stated: *If an event A causes event B, then the exeuction of A and B must be scheduled in real time so that A is completed before B starts.*

This implies that if A does not cause B, B could be executed before A in real time, even if the logical time of B is after that of A.

* 1. One implementation of virtual time is the Time Warp system.
     + For correct implementation of virtual time, it is necessary and sufficient that *at each process* messages are handled in timestamp order.
     + Structure of the run time representation.
       - A unique *process name*.
       - A *local virtual clock* (LVT) which has to be compatiable with the *global virtual time* (GVT), but it doesn't have to have the same value as GVT.
       - A *state* which is a collection of variables.
       - A *state queue* containing saved copies of the process's recent states.
       - An *input queue* containting all recently arrived messages sorted in order of virtual receive time.
       - An *output queue* containing all negative copies of the messages this process has recently sent, kept in virtual send time order. They will be used in case of rollback.
     + See Figure 1 and Figure 2 in Jefferson's 1985 paper for illustration.
     + Global virtual time: GVT at real time *r* is the minimum of (1) all local virtual times in all local virtual clock at time r, and (2) of the virtual send times of all messages that are in transition.
       - GVT never decreases.
       - GVT can serve as a floor for the virtual times to which any process can ever again roll back.
       - GVT can be viewed as a moving *commitment horizon*: any event with virtual time less than GVT cannot be rolled back and may be removed from the system.

Thus the name *time warp*.

* + - * If (1) every event completes normally, and (2) messages are delivered reliably, and (3) the scheduler does not indefinitely postpone execution of the farthest-behind process, and (4) there is sufficient memory, then *GVT must eventually increase*.
  1. Examples of virtual time system include distributed discrete event simulation, distrbuted database concurrency control, virtual circuit communicaiton.
  2. Virtual time has an anlogy to virtual memory in memory management.
     + The virtual address space of a page is its spatial coordinate; the virtual time of an event is its temporal coordinate.
     + A page resident in main memory at time *t* is analogous to an event with a virtual time in the future of process *x*, where the page may be accessed in the future, the event will be processed in the future.
     + A page out of memory at time *t* is analogous to an event in the present or past of process *x*.
     + Accessing a page in memory is relatively inexpensive, but accessing a page out of memory at time *t* is very expensive (*page fault*); similarily sending a message that arrives in the virtual future of the receiving process is relatively inexpensive, while sending a message into its virtual past causes a very expensive *time fault*, that is, rollback.
     + Under a virtual memory system, it is only cost-effective in time to execute programs that obey the spatial locality principle, so that most memory accesses are to pages already resident in memory, and page faults are relatively rare. Likewise, under a virtual time system, it is only cost-effective to run programs that obey temporal locality principle, that is, most messages arrive in the virtual future of the destination processes so that time faults are relatively rare.
     + The term ``memory mapping'' refers to the translation of virtual addresses to real address. We could use *time mapping* to refer to the mapping of virtual times to real times. The same virtual address may be mapped to different physical address in memory at different time, and similarily, the same virtual time may be mapped to (scheduled at) different real times at different places.
     + The only acceptable memory maps are the one-to-one functions because they preserve *distinctness*, mapping distinct virtual addresses (pages) to distinct real addresses (frames). At any given moment, some virtual addresses may be unmapped because they refer to pages not in memory. Similarily the only acceptable time maps are the strictly increasing function because they preserve *order*, mapping distinct virtual times into disctinct real times. At any given place, some events may be unmapped (not yet scheduled) because they are in the local future.
     + For a process running under virtual memory, the page fault rate can usually be reduced by increasing the number of pages it is permitted to have in main memory. Similarily the time fault rate of a process running under a virtual time can usually be reduced by increasing the number of events that are in its near future. Essentially, more interaction among LPs reduces the chances of time fault. But it may slow down the simulation progress.
     + If a process can have enough pages in memory, its page fault rate can be reduced to zero; in general, this is undesirable because it leads to inefficient use of main memory. Similarily, if a process has sufficiently large number of events to process in a short period of time, its time fault rate can be reduced to zero; this too is undesirable because this process then becomes bottleneck, holding back the progress of GVT, leading to inefficient use of real time (other processes have to wait).
* GVT calculation. There are many different ways to compute GVT.
  1. Centralized calculation: Each LP sends its LVT to a centralized manager, the minimum of these is the calculated GVT.
  2. Distributed calculation: LPs broadcast their LVT periodically. Each LP calculates GVT to be the minimum of all LVTs (need non-blocking read).
* Rollback distance vs. forwarding distance (a model of random walk)
  1. Rollback converges (partial ordering of the events, cases for events that have the same time stamp, and different time stamp)
* Memory management, all issues are memory vs. speed
  1. Interleaved state saving (How often?) Instead of saving every single input/output events, interleaved state saving strategy only saves a snap shot of the state at certain time. When a rollback is needed, the simulation is rolled back to the point where the state is saved.
  2. Fossil collection: events that have a clock value smaller than GVT will never be used again, removing them can recover some memory
     + lazy: don't collect until running our memory
     + aggressive: collect at a fixed interval or upon a certain event
  3. Return un-processed messages

When memory is running low, one can return un-processed input events back to the sender to free up some memory. The ones with highest time stamps should be returned first because they are less likely to have effects on the system.

* 1. Gafni's protocol: instead of always returning un-processed message, reclaiming memory according to the nature of the activity.
     + If an output event is the cause, a corresponding anti-event is sent. The local simulation is restored to a previous state.
     + If an input event is the cause, do conventional return-message.
     + If a state saving is the cause, discard that set of state variables. They can be re-computed later if needed.
     + Artificial rollback: with the absence of straggler, rollback is called artificially to reclaim memory.
* Communications in PDES
  1. Communication overhead
  2. Difference between parallel simulation (shared physical memory) and distributed simulation (message passing)
  3. In distributed simulation: less communication would reduce total delay, may cause more rollback
  4. The relative value counts: i.e. the ratio of computing time vs. communication time
* Performance models:
  1. Chen's work: unify the performance model for memory effect on message distribution, rollback, GVT advances in shared memory environment using a Markov's process model
  2. Gupta, Akyildiz, and Fujimoto: performance model for homogeneous and heterogeneous multiprocessor, shared memory environment
  3. Kleinrock and Nicol have separately developed some bounds for the performance of Time Warp system
  4. Lubachevsky, Weiss and Shwartz's work: the average time $\tau$ to complete the simulation of a system with *N* nodes and *R* events on a *p*-processor PRAM satisfies