Mark Randles
2005-09-27
CS 680 – Dist. Sim.

Issues in Conservative Synchronization

Within there are a variety of topic to concern ourselves with. At the very upper levels these topics take on an abstract form, and begin to split down into various areas, such as synchronization, communication, and partitioning. Each of these are interrelated with one another, and in may ways, the choice in one area will influence the choice in others. However, few of the choices are of interest, as usually from one choice flows the others. However, what is of interest to us is the influence and interrelationship between these topics. In this paper we will explore the interrelationship of communication and a specific form of synchronization.

Conservative synchronization is a type of synchronization used in PADS (Parallel and Distributed Simulation). We deem this type of synchronization as conservative since it allows no LP (logical process, a thread of simulation) to advance without making sure that it is safe to do so, hence being conservative. The reason this is beneficial is because you can maintain a global causality order, in which all the events are executed in precisely the correct order for the given simulation. However, with this approach there are some drawbacks, such as a LP deadlock and the high-cost of the communication needed to ensure that causality is enforced.

To maintain the local causality order, a LP must find a way to make sure it's next message is the correct message at that time. In a conservative algorithm, an LP can't proceed till it absolutely knows that it is safe to. To accomplish this task, LPs must use a value called lookahead, which defines the minimum time past the current time that a remote LP can expect a message from it's LP. LPs exchange timestamps with the lookahead using null messages. By letting the connected LPs know when to expect it's next message, they can process their own local events along with any remote events until it reaches the minimum lookahead time at which point it must stop since it doesn't know if it can go on, because a new message could be scheduled before the next event happens. However, this leads to a dangerous condition, deadlock.

Deadlocking is perhaps one of the most dangerous things that can happen in a conservative synchronization based simulation. Deadlocking is a situation where no LP knows if it's safe to proceed as it's waiting for the “go-ahead” from other LPs in the same situation.

Since no LP knows if it's safe to proceed, they stall and so does the simulation.

To recover from a deadlock, the LPs and their controlling thread, need to know something about the current and future state of the simulation. In Parallel and Distributed Simulation Systems, Fujimoto talks about a few methods of resolving deadlocks. One method, based on a tree, uses the way the simulation communicates to tell if a deadlock has occurred.

In the tree based deadlock detection algorithm, each new communication link between LPs is added to a virtual tree where the sending node is the parent and the receiving node is the leaf. When a child node becomes stuck or blocked because it's unsure if it's safe to proceed, it signals the parent that it's no longer a leaf, and there for the parent is now a leaf. If all the nodes disconnect, signaling a dead lock, the controller LP is a leaf and needs to resolve the deadlock.

To resolve the deadlock, the controller LP needs to find a safe event for at least one of the LPs, to make the controller LP a parent and the safe LP the child and start the process all over again.

There are a few ways of finding the correct LP, but generally the LP that the controller node wants to select is the LP which has the lowest event time of any events outstanding in all LPs. This requires that the LPs have a mechanism built in that can report the minimum next event time to an external source.

All of the deadlock detection and resolution algorithms are communication intensive. Not only must LPs exchange information between parent and leaf, but they also need to exchange information with the controller LP. All of this communication is expensive, so much so that every attempt must be made to avoid a deadlock situation.

To avoid the deadlock situation and there by improve the overall speedup of the simulation. There are a few ways we can do this. First and foremost we can fine tune the lookahead and find an optimal value given the simulation circumstances. Secondly, it is important to partition the simulation correctly, getting the correct parts of the model mapped to the correct LP. Lastly, it's important to make sure that the event granularity is correct, so, if possible, no event will consume so much time as to unbalance the simulation. If one event were to do this, it would improve the chances of a deadlock, since it would tie up it's LP for a large amount of time.

Once there is a good way to resolve the deadlock issue, it

becomes clear what the true bottlenecks to a conservative algorithm are. First and foremost a conservative algorithms total wall clock time is highly depended on what sort of computer it is run. Since there is an inordinate amount of communication needed to insure both the local causality order and to resolve deadlocks, a system must be chose that reduces the total amount of time used on the communication itself.

There are only, however, two different types of parallel computer paradigms, shared-memory and distributed memory (cluster). Of the two shared-memory is the fastest, as communication is handled through a shared memory space on the local machine, rather then as messages across a network or some other medium as in distributed memory machines. However, with share memory you introduce a whole another level of possible deadlocks as processes vie for access to global memory spaces. This, however, is a mute point if the program is written properly.

The actual performance of the simulation is ultimately limited by the overhead associated with synchronization and deadlock detection. As with all programs, eventually the speed gained by adding more LPs will be overwhelmed by the increasing cost of the communication between threads. Even without the upper barrier, the ultimate speed lies in the ability of the simulation to avoid deadlock and keep the remote event queues in the LPs full. If these conditions aren't met the simulation will not run at peak performance, and the speedup will degrade.

One last issue that is important to all PADS is load-balancing LPs to processors. With a conservative algorithm, it may be beneficial to have more then one LP mapped to a single processor, especially if deadlocks are common on the system. Since the deadlocks already limit the total speedup, increasing the LP to processor ration could increase the processor usage up until the point that not enough active (non-deadlocked) LPs are available to keep all of the processors busy. This approach would be useful on both a shared-memory or distributed memory machine, depending on the simulation and load-balance within an individual LP.

Communication and synchronization are integral parts of the same puzzle. Both are interdependent in a PADS program, and by weighing the various options available, one can make a educated decision as to which will give the user the most gain over another approach.