A Network Version of the Time Warp Operating System

Steven Bellenot
Mathematics Department B-154
Florida State University
Tallahassee, FL 32306-3027
bellenot@math.fsu.edu

ABSTRACT

TWOS (Time Warp Operating System) is a special purpose operating system designed to run parallel discrete event simulations. TWOS runs these parallel simulations using an optimistic synchronization method based on rollback and message cancellation. TWOS was developed by JPL (the Jet Propulsion Laboratory), originally for the JPL/Caltech Hypercube and later on the BBN Butterfly. Although TWOS has run on a network of Sun's for many years, this port has been primarily a debugging version and not a performance platform for TWOS. Experiences in the on-going upgrade of the debugging Sun network version to a performance version running on a heterogeneous network are given.

Introduction

This cluster conference (Tallahassee, 1992) included many papers about adapting sequential codes into parallel network codes to run on a cluster of workstations. However we are reporting on porting parallel code from parallel machines to a network of Sun workstations. A good general reference to the subject of parallel discrete event simulations is [F].

The Time Warp Operating System (TWOS) [JBW] is a special purpose operating system designed to run discrete event simulations in parallel with the primary goal of maximum speedup. (TWOS runs simulations; it is an operating system and not a simulation itself.) TWOS uses a well established synchronization method which at first exposure seems unusual. The synchronization method is based on the theory of virtual time [J]. Each event in the simulation is assigned a virtual time by the user. TWOS guarantees that the resulting parallel execution will produce results identical to running every event in increasing virtual time order. TWOS actually runs events at many different virtual times in parallel in order to provide good speedup. Instead of conservatively executing only those events which can safely be run in parallel without compromising the sequentiality constraint, TWOS permits each node of a parallel machine to run the earliest event it has. As a result, some events may be performed out of order, in which case TWOS rolls back the disordered computation and cancels any of its effects, before going on to process the events in the correct order.

TWOS Simulation

A simulation to be run on TWOS must be decomposed into objects, which run events and send time stamped messages to other objects. In TWOS each object is like a process in a operating system; it has its own stack, data space and thread of control. The message is in effect scheduling an event for the receiving object. The virtual time of the event is the receive time of the message. Thus objects communicate solely by message passing with no shared memory whatsoever. Any object may send a message to any other object at any time, without setting up any kind of channel between the objects. Except for initialization and termination code, all user code runs as part of an event.

TWOS runs one simulation at a time, with the goal of completing that simulation as quickly as possible. Each node of the parallel execution hosts several simulation objects which are scheduled independently of all other nodes. The TWOS scheduler always chooses the local object with the unprocessed message at the earliest simulation time to run next. Objects are only preempted when another object receives a message of earlier time than that of the event currently running.

Since each processor schedules without waiting for or consulting with, other processors, at any given instance of real time the system processors may be executing objects at a wide range of simulation times. An object at a low simulation time may schedule an event for an object with a high simulation time. If this event is earlier then what the object is currently working on, the receiver must rollback its computation to the time of the newly arrived message. Any erroneous work done by the out of order computation must be totally undone. Undoing erroneous work requires throwing away local results and sending message cancellations to other objects. TWOS is able to correctly undo any work done prematurely and any side effects it may have had. Rollback and message cancellation are transparent to the user's application program.

Every object has a set of private variables called its state which cannot be directly examined by any other object. Every event causes the creation of a new version of the state which is time stamped with the simulation time of the event. TWOS typically keeps multiple copies of each object's state in order to support rollback.

To support message cancellation, copies of output messages are kept in TWOS. A negative copy of an event is sent to cancel a wrong event. When the negative and positive messages (events) meet, they annihilate each other and the receiving object is rolled back to the sim-

ulation time of these messages. Thus each object has three queues kept in time stamp order: input messages, output messages and a state queue.

TWSIM

Also part of TWOS is a sequential simulator called TWSIM that runs exactly the same simulations as TWOS. TWSIM is a conventional sequential program which uses a splay tree for its central event queue. Although TWSIM is useful in debugging application code, it is also used to determine the fastest one node times for the speedup curves. Speedup is measured relative to a TWSIM run time and not relative to a one node TWOS run time.

Dynamic Load Management

TWOS versions 2.5 and later support dynamic load management. Simulation objects are migrated from heavily loaded nodes to less loaded nodes. As mentioned above, objects are like operating system processes and so object migration is similar to process transportation. Since each object uses a fair amount of memory, rather than transport all of the queues, the object is split and only the most recent part of the object is moved. This split is at some simulation time, say T. Events for time before T are done on the original node and events after T are done on the new node.

TWOS also includes static load balancing tools which use a TWSIM sequential run to obtain the CPU requirements for each object during the simulation. A simple bin packing is used to assign objects to nodes by trying to level the CPU requirements on each node. Besides the lengthy sequential runtime, static load balancing does not take into account objects whose CPU requirements change during the course of the simulation (See [RJB]).

Benchmarks

We used three benchmarks, developed at JPL, to measure performance. The speedup numbers below are for TWOS 2.5 running on a BBN butterfly GP1000. These maximal speedups require using 60 to 72 processor nodes. The Pucks benchmark[HBD] models colliding pool balls and has 304 objects, 405802 events and a speedup of near 14. The STB88 benchmark [WHF] is a theater level war game with 380 objects, 603472 events and a speedup above 26. The Warpnet benchmark [PEW] models a communications network and has 169 objects, 45319 events and a speedup of almost 30. Figure 1 shows these speedups. These speedup are for the static binding of objects to nodes. The speedups for runs with dynamic load management are only slightly lower. These benchmarks were truncated for our testing of the Sun network version. And it is these truncated benchmarks whose preformance is graphed in the later figures. The truncated Pucks had only 41746 events (10%), the truncated STB88 had 29593 events (5%) and the truncated Warpnet had 8923 events (20%).

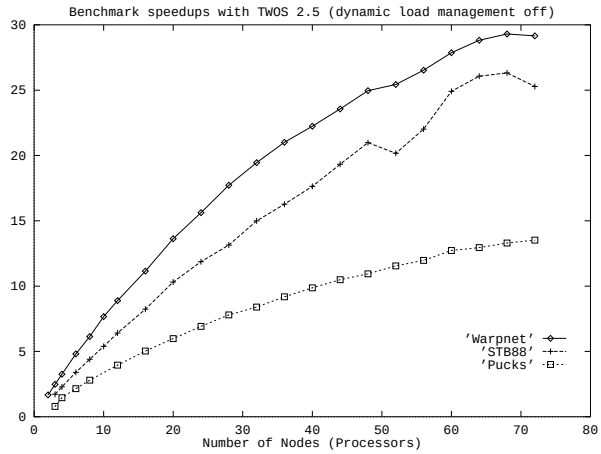

Figure 1. Benchmark speedup curves for Butterfly.

Platform History

TWOS was designed to be portable. Only the bottom layer is machine dependent. TWOS has been ported to a number of platforms over the years. The first target was the JPL Mark 2 Hypercube, an 8086 based 32 node machine with a memory of only a quarter of a megabyte per node. The second port, mainly to help with debugging, was targeted to a network of Sun 3/50's on top of UNIX. However, the two main platforms over the course of the project were the JPL Mark 3 Hypercube and later the BBN butterfly GP1000. Both of these machines are 68020 based and each had 4 Megabytes of memory per node. It is on the these latter two machines that performance of TWOS was tuned and new features were tested. TWOS has also been ported to a SPARC network, a transputer network, a BBN Butterfly TC2000 and the Intel Delta machine.

Real Time Clock

From the start, TWOS required a real time clock to measure CPU usage of the objects. Originally this information was used to determine how often to save an object's state, a function which is no longer needed. This same information is now used in load management. Also the real time clock is used for the performance tools mplot and fplot (See [BD]). However, the clock resolution on the Sun 3/50 was only 10 – 20 milliseconds and the features based on the existence of a real time clock were omitted in the Sun network debugging version of TWOS. Since most TWOS objects only execute for a few milliseconds, the real time clock should have a shorter resolution, for example the Butterfly's 1/40 of a millisecond (62.5 microseconds). A memory read is all that is needed to access the real time clock on the Butterfly; on the Sun it is a system call.

Message Types

Besides the positive and negative simulation event messages, TWOS has a variety of message types. Reverse event messages are used for a high level flow control. I/0 is done with Standard Out messages. All other message types are system messages. There are system messages for GVT which are used to measure the progress of the simulation and for garbage collection to reclaim memory. Since objects move around there are object location messages. System messages have priority over simulation messages in the message queues.

Migrating objects in TWOS takes more than one system message. The object to be moved is first split by virtual time and then migrated. The Butterfly message sending has a feature which allows a message to be pushed to the front of queue, which was being used to rush system messages. Sometimes the two messages to split and move got reversed when the second message was pushed onto the queue before the first one was read. As a result the Butterfly version of TWOS has two message queues, one just for system messages and the other for simulation messages.

Network TWOS

In the summer of 1991 it was decided to upgrade the Sun 3 network version to support features of TWOS which had been added to TWOS since the original port and to improve its run time performance. The Butterfly and Sun versions of TWOS shared most of the non-communications code, thus message passing was the obvious place to look for improvement. Unfortunately only a few Suns were available for making tests. TWOS and SunOS both used more memory than when the original Sun network port was done. Because the 4 Megabytes of memory on the Sun 3/50 was too small, each of the benchmarks had a significant number of page faults, ruining the performance. There were only four Sun 3/60 machines available for network testing, and these were workstations constantly in use during the day. A real time clock was added for the performance tools and for dynamic load balancing. The code was profiled and tuned for better performance. However the major improvement was in the setup method of the network version.

The network version was made before NeWS or X-windows became available. The setup consisted of opening several windows on the work station. In each window, the user remotely logged onto another machine on the network. Then the user changed the directory to the directory containing the executable. Finally the executables were hand started in increasing node order, pausing a few seconds so they were fully initialized in node order. Clearly this method could not be used for batch processing.

Startup Programs

A Bourne shell script called 'xtw' was written to automate the setup of a network TWOS execution. The xtw program read the existing NODES file which listed which node number was running on which machine and remotely started up an xterm program on the remote host which in turn ran the TWOS executable. The machine executing the xtw script must be running an X-windows display server and the remote machines must have permission to draw on the display. A convenience script, 'twxhost', could be used to find the names of the remote hosts, from the NODES file, and use xhost to give them permission to draw on the display.

The network version was changed to allow out-of-order node initializations and the xtw script became a way to do batch processing of TWOS simulations. This allowed performance runs to be run all night long while the workstation owners were away. As an aside, one of the workstation owners started using lockscreen, a busy screensaver, midway through this project. It took a while to figure out why the performance had dropped.

Code Profile

To understand the profile information we first give an introduction to the communication code in the Sun version. The network communication topology is that of a fully connected network. Each node directly sent and received messages from every other node (and a host program). These communication channels use stream sockets, one for each direction between any two nodes. The advantages of stream sockets are that messages are reliable and are received in the order sent. Also having a reliable channel between any two nodes made debugging easier, since one could look to see which messages were in transit with Tester, our home grown debugger (see [RBJ]). The old code polled every incoming socket (four sockets for a four node run) for messages each time around the main loop of the TWOS code.

Profiling the old network code showed that most of the time (twenty to fifty percent) was being spent doing reads of sockets. An average of 250 microseconds per read (roughly), but since the code was polling for I/O, most of these reads were to empty sockets. The TWOS code was changed to use signals, in particular SIGIO, instead of polling. The signal handler merely set a flag to indicate the presence of I/O. Thus the new TWOS would only poll all incoming sockets when a SIGIO had occurred and hence one of the sockets had a message. There is a select system call which will tell you which of the incoming sockets have input. As the use of the select call slowed down TWOS, at least on the four nodes it was tested on, it was faster to just to poll each socket. The select call was left in the code as an unexercised compile time option.

Another big saving came on the sending side of the sockets. A message used to take two write system calls to send one message. The first write was for a message header and the second for the message itself. The writev system call allows the writing of two (or more) non-contiguous pieces of memory with one system call. Replacing the two socket writes for each message with one writev reduced the number of write system calls in half. Generally speaking the calls to the UNIX kernel were expensive and frequent. We will have more of these kernel call problems when we put in the real time clock.

Alarms in TWOS

The old Sun version used an alarm call to periodically run a GVT calculation. GVT results are used for memory management, I/O commitment and termination detection. The Butterfly version had a separate alarm for dynamic load management [RJ]. In addition each object execution (there could be several per simulation event) was timed with the real time clock. Thus the new Sun version was going to frequently use yet another system call. Figures 3 and 4 below show the effect of three different clocking schemes. The first is the old method with no timing of the object execution, which is not acceptable but does show what these system calls cost. The second is call process clock, which uses the UNIX itimer_virtual clock (and NOT the itimer_prof clock). This UNIX clock only measures the time the TWOS code (or system calls made by the TWOS code) is running. (The itimer_prof clock doesn't include the system calls.) This process clock would be acceptable for dynamic load management. However, there are a number of diagnostic tools for TWOS which require wall clock time. The wall clock option in the new TWOS uses the itimer_real clock from UNIX. This clock is also used to generate the signal SIGALRM used in the alarm calls. A certain amount of masking and unmasking signals made the wall clock more expensive than the process clock (see figures below).

Performance of the New Network Version

Figures 2, 3 and 4 compare the performance of the new network version of TWOS to the performance of TWOS on the Butterfly. The first base point is the run time of the sequential versions of the simulations using TWSIM. Figure 2 shows a surprise, great variations in the Sun run time of simulations depending on which operating system version the code was compiled on. In each case the compiler used was the bundled one in /bin/cc and not gcc nor the unbundled Sun compiler. The difference we think is that the -O flag (optimize) to the compiler changed its meaning going from 4.0.x to 4.1.x. It use to mean -O1 but now it meant -O2. The Butterfly compiler was from Green Hills. Figure 2 shows that a Butterfly node and the Sun 3/60 are roughly similar

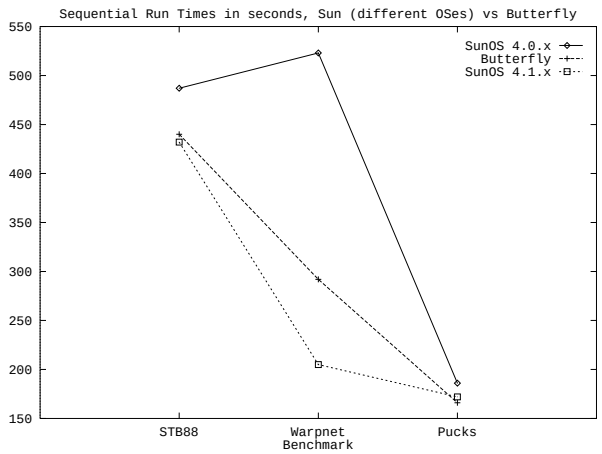

Figure 2. Sequential run times.

machines for these simulations.

Figure 3 shows the four node run times for both the Sun and the Butterfly. Note that Pucks and Warpnet have switched locations between Figures 2 and 3. Warpnet has a much higher computation to communication ratio than STB88, which in turn has a higher computation to communication ratio than Pucks. The cost in run time of using the process clock is two to three percent over using no clock. And the cost of using the wall clock is another two to three percent in run time over using the process clock. Figure 3 shows that the new network version successfully competes with the Butterfly version, at least on four nodes.

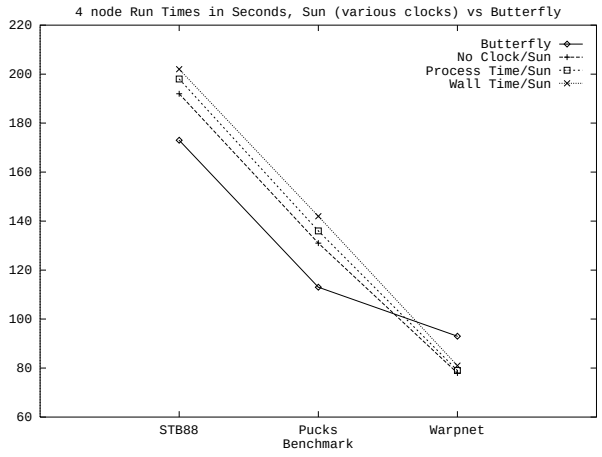

Figure 3. Four node run times.

Figure 4 shows the four node speedups for both the Sun and the Butterfly. The SunOS 4.1.x sequential times are used for these speedups. Note that Warpnet and STB88 have switched locations between Figures 3 and 4 so that each benchmark is in a different location between Figures 2 and 4. Speedup results are worse (for the Sun)

than the run time results because the Sun sequential run times are mostly faster.

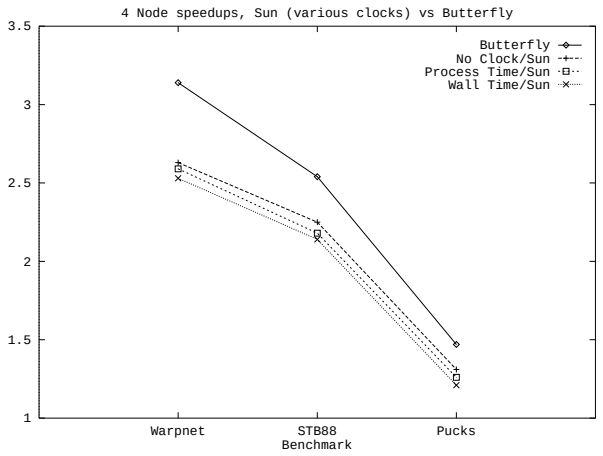

Figure 4. Four node speedups.

Cautions

The current fully connected socket system does not scale well. With more nodes the select call should become faster. Also there is a limit on the number of sockets a UNIX process can have. At four nodes is unlikely that the ethernet bandwidth was limiting communication, but as we add more nodes this likelihood increases.

There is a better way of implementing the wall clock in the Sun version which should make it no worse than the process clock in terms of run time.

At least in SunOS 4.x.y, stream sockets, while they have reasonable throughput, seem to be slow in response time. A simple pair of application objects playing ping pong with a single message can only send about ten messages per second. The benchmark applications are all saturated at four nodes, that is, they always have bunches of good simulation work to do.

Dynamic load balancing is not robust on the Sun version of TWOS. There seems to be a race condition which is unseen on the butterfly because of its separate system message channel between each pair of nodes.

More Recent Work

Recently work on porting the network TWOS to a network of Sun 4 workstations has began. Communications speed is even more critical on a SPARC version since the computation speed has increased so much over the Sun 3. Placing TWOS atop PVM increases the ping pong messages to over 60 messages per second. The use of stream sockets must be abandoned for the datagram style of PVM in order to get better communication times. (See [GS] for a description of PVM.)

How to obtain TWOS

TWOS 2.5.1 is available through COSMIC (NASA's Computer Software Management and Information Center) University of Georgia, 382 East Broad Street, Athens, GA 30602-4272, phone (706) 542-3265.

References

[BD] S. Bellenot and M. Di Loreto, *Tools for measuring the performance and diagnosing the behavior of distributed simulations,* Distributed simulation, SCS simulation series 21 (1989), 145-150.

[F] R. Fujimoto, *Parallel Discrete Event Simulation,* Communications of the ACM, 33 no 10 (1990), 30-53.

[GS] G. Geist and V. Sunderam, *Network based Concurrent Computing on the PVM System*, Concurrency: Prac. and Exp. 4 (1992), 299-311.

[J] D. Jefferson, *Virtual Time,* ACM Transactions on Programming Languages and Systems, 7 (1985) 404-425.

[JBW] D. Jefferson, B. Beckman, F. Wieland, L. Blume, M Di Loreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren, J Wedel, H. Younger, and S. Bellenot, Distributed Simulation and the Time Warp Operating System, Proc. 12th SIGOPS - Symposium of Operating Systems Principles, 1987, 77-93.

[HBD] P. Hontalas, B. Beckman M. Di Loreto, L. Blume, P. Reiher, K. Sturdevant, V. Warren, J. Wedel, F. Wieland, D. Jefferson, *Performance of the colliding pucks simulation on the time warp operating systems (Part 1: Asynchronous behavior and sectoring),* Distributed Simulation, SCS Simulation Series 21 (1989), 3-7.

[PEW] M. Presley, M. Ebling, F. Wieland and D. Jefferson, *Benchmarking the Time Warp operating system with a computer network simulation,* Distributed Simulation, SCS simulation series 21 (1989), 8-13.

[RBJ] P. Reiher, S. Bellenot and D. Jefferson, *Debugging the Time Warp Operating System and its Application Programs,* Proc. Symposium on Experiences with Distributed and Multiprocessor Systems 2, (1991), 203-220.

[RJ] P. Reiher and D. Jefferson, *Virtual Time Based Dynamic Load Management in the Time Warp Operating System* Trans. Soc. Comp. Simulation, 7 (1990).

[RJB] P. Reiher, D. Jefferson and S. Bellenot, *Temperal Decompostion of Simulations under the Time Warp Operating System,* Distributed Simulation, SCS Simulation Seris 23 (1991), 47-54.

[WHF] F. Wieland, L. Hawley, A. Feinberg, M. Di Loreto, L. Blume, P. Reiher, B. Beckman, P. Hontalas, S. Bellenot, and D. Jefferson, *Distributed combat simulation and time warp: The model and its performance,* Distributed Simulation, SCS simulation series 21 (1989) 14-20.