JET PROPULSION LABORATORY

INTEROFFICE MEMORANDUM
SFB: 363-88-005
August 12, 1988

To: The Time Warp Team

Re: Following paper

From: Steven Bellenot

The following paper, which is about graphics but which contains no graphics, might be considered my final report for this summer. My e-mail address is bellenot@nu.cs.fsu.edu.

Tools for Measuring the Performance and Diagnosing the Behavior of Distributed Simulations using Time Warp

Steven Bellenot
Departments of Mathematics and Computer Science,
The Florida State University,
Tallahassee, FL 32306
and the Jet Propulsion Laboratory

Michael Di Loreto
Jet Propulsion Laboratory,
California Institute of Technology,
4800 Oak Grove Drive,
Pasadena, CA 91109

it is Time Warp as implemented by JPL

Abstract

Like any new design, Time Warp (an operating system for distributed simulation), needs tuning in order to obtain maximum performance. We do not claim to have tuned Time Warp to this peak, but we offer a collection of tools and observations we have found useful in improving Time Warp's performance. We caution the reader that we do not claim to present the latest in graphics nor the last word in performance measuring tools. We hope that this documentation of our experience will benefit others.

Time Warp has constantly proved to be a source of counter-intuitive results. Many so called optimizations have actually caused Time Warp to run slower or to even crash. We have changed an application to send fewer messages only to have it run slower. We have done queueing chores earlier and simulations run slower. (We then tried doing them later, but it ran slower too.) Indeed, we can still be surprized by the consequence of a change to Time Warp. Clearly, some tools are required to determine what is happening.

Our tools are either graphical or statistical. In both cases we make a log file of either all object executions or of all messages sent. We can do this for short simulations (around 20 seconds on 32 nodes). With this complete log file, we can analyze the data in several ways. Graphically presented, the large amount of data (perhaps 70,000 messages or 180,000 execution pieces) can have a

chance of being understood. The graphic limitations naturally lead us to some statistical tools.

The Time Warp Environment:

For this paper, *Time Warp* refers to a special purpose operating system for running discrete event simulations on multi-processors. Time Warp was built and is maintained by the Jet Propulsion Laboratory for the Army Model Improvement Program (see [Jefferson 87]). To run on top of Time Warp, a simulation must be broken into *objects* which schedule events for each other via messages. We will call such a collection of objects an *application*. The Application layer (in theory) is transparent to the number or kinds of computers it runs on. Currently, the application objects are statically assigned to nodes at run time, and hence knowledge of the application is used to load balance the nodes. The simulation time, that is the time that simulation events are scheduled is called *virtual time* in Time Warp.

By the Time Warp Layer, we mean the layer of abstraction between the objects on the application level and the lower machine dependent level. In practice, this lower level can be viewed as message passing kernel which logically connects any two nodes. Time Warp has been ported to a number of machines, a network of Suns and a Butterfly, for examples, but results of this paper are based on Time Warp running atop the Mercury message passing kernel on one of the Jet Propulsion Laboratory's Mark III hypercubes with 32 nodes.

For the purpose of this paper, we may view the application as sending *positive* messages, which may or may not be correct. Time Warp will *rollback* an object which has gone down an incorrect simulation path and Time Warp will send *negative* messages (often called *anti-messages*) to cancel the effects of the incorrect positive messages. Time Warp also has system messages of which the most important for our purposes are those used to compute *global virtual time* (also denoted by GVT).

We will talk about two applications. The important application is *ctls87*, which is called STB87 in [Weiland 88]. Ctls87 is a distributed combat simulation with 3 distinct phases and a reasonable run time of around 20 seconds. The other application is an artificial one called *slooow*. Slooow is a fully connected model designed to give wrong answers when it is out of sync. Slooow stresses the message passing system and slooow violates every known principle of what a good Time Warp object should not do. Slooow has large fan in and large fan out as well as a very high communication to computation ratio. Moreover, slooow is time driven and (at least in theory) completely parallel.

Graphic Tools

Since there are two kinds of time in a simulation, real time and virtual time, it seems natural to plot graphs with the two times on the different axes. The virtual time of an event is easy to determine, however to obtain the absolute real time of events on several nodes required some care (see synchronizing the clocks below). The first graphic (fplot) was of execution (real) times of an objects vs the virtual time at which they were running. The real execution time of an object was graphed as an horizontal interval at height given by the virtual time of the execution.

The second graphic (mplot and later m3plot) was for messages. Each message was represented as a line from (real-send-time, virtual-sent-time) to (real-receive-time, virtual-receive-time). A Silicon Graphic's Iris Workstation was the target machine for the graphics. We could use a wide range of colors for the different objects, or just a couple of colors which show the difference between messages and anti-messages or between good positive messages from those which were later cancelled.

Both of the plotting programs had acquired lots of features. We could zoom-in and zoom-out recursively. We could graph just what was happening on a single node. (Or view the nodes one at time like a movie.) We could view what just one object was doing. The graphic could be spread across a number of pages. In any of these views we could identify any line by pointing the mouse and clicking on the line.

M3plot would allow you to stack two or three simulations on the same graph. The later simulation graphs would be shifted-up in the virtual time direction to prevent overlap. M3plot also added a third dimension to message plots. The third dimension was object "number". It became possible to view the graph in perspective and walk around in it or even walk inside the graph. Using the object "number" in the third dimension worked for slooow, but was less successful for ctls87. The ordering of the objects was first come, first served. Perhaps an ordering by node number (which is how slooow was ordered) would have been more successful for ctls87.

The third graphic (hc - for hypercircle) was designed to show the inner workings of Mercury on the hypercube. The nodes of a hypercube were positioned on a circle (in a gray code order). Each communication channel between two nodes was represented by a line connecting the corresponding two points on the circle. As a message was sent, its path in the hypercube was translated in to a path on the hypercircle which was highlighted (in green for positive messages and blue for anti-messages). While a message was in transit, the other messages which were also in transit at that time were shown. Apparent channel conflicts were displayed in white.

The log file had to be created while the simulation was running. The log entries were stored in RAM and put into a single file after the execution of the application was over. These files are quite large, a typical simulation which runs about 20 seconds could produce about 70,000 messages or 180,000 execution pieces. Each execution log entry requires at least one object name, one virtual time and two real times. The message log entries required twice that space. (Messages were given four real-time stamps: 1. When the message was created by the application. 2. When the message was given to Mercury to send to another node. 3. When Mercury on the receiving node got the message. 4. When the Time Warp layer enqueued the message.) The time needed to read the output file was long. A program that encoded this information into binary file was used to decrease the set-up time of the graphics programs.

Time intervals

The graphic programs greatly increased our interest in delta time intervals as opposed to total time. At first, the largest delays were produced by a race condition in the Mercury kernel. This race condition was known by the people maintaining Mercury. They were using spin locks to determine the direction of the bidirectional channels. Two spinning nodes would backup traffic on the whole cube. Since the nodes are all on different clocks, eventually the spins would stop being in sync. The code now carefully spins at different speeds on different nodes.

We made a number of tests to check this Mercury fix. To decide if gaps were Mercury gaps, we had to identify the operations in the

Time Warp layer which required large time investments each time they were called. All of these Time Warp operations used a small total time investment compared to the total run time of the application. The graphics plots were still showing time periods where nothing was being done. Two time periods were in Time Warp and they were on the order of 50 - 100 milliseconds. One was garbage collection and the other was a printf from node 0 to the host computer. (Communication with the outside world from a hypercube is much slower than inner-node communication.) Garbage collection is done more or less at the same time by all the nodes.

Getting the clocks in sync (see below) and trying to measure the performance on Mercury became important. We designed hc (hypercircle) to show the channel use between nodes and the amount of conflict for the use of these channels. And at first glance, it looked like there was a lot of conflict. However, Mercury wasn't the problem.

Mercury Performance

Using the statistical tools below we found that most messages were being transferred by Mercury in less than than two milliseconds. Moreover, the distance or number of hops the message had to travel was much less important than message size in determining message transit time while in Mercury. There were a handful of messages taking up to a dozen milliseconds to travel through Mercury. However, it is possible that all of these were behind a GVT collection.

Time Warp does not use any special knowledge of the topology of the hypercube. While computing a new GVT, node 0 requests a message from all other nodes at once. This influx of messages from all nodes to node 0 is what we call a *GVT collection.* Mercury, because of the hypercube topology, will eventually send half these messages through the bottleneck channel connecting node 0 and node 1. Sixteen messages through the same channel in a dozen milliseconds doesn't seem out of line since the average message time is slightly over a millisecond. We do know better algorithms for these GVT collections on the hypercube, but GVT calculations only happen once each three to five seconds.

Communication and Queueing

Mercury does have a limitation. It runs in constant space. Too many messages (around 500) in Mercury's receive queue will cause Mercury to fail. To prevent overflowing the receive queues, the Time Warp layer most restrain itself. For reasons used in computing GVT, Time Warp on the receiving node sends back an acknowledgement to the transmitting node for each message it receives. Time Warp limits the number of transmitted messages for which have not been acknowledged. If this limit (max_acks) is x, and there are y nodes, then at most (y - 1) * x messages can be waiting on any node. Common values of x range from 2 to 20 and on the current hypercubes y is 32. Interestingly enough this limit is more important after the simulation has completed. After the simulation all the nodes dump their statistics to node 0.

If max_acks is high enough (near twenty) then almost all of the message transit time is spent between time stamps 3 and 4 above. Most messages spent most of their time in Mercury's receive queue. When max_acks is two, many messages spent some time between time stamps 1 and 2, but most of the time is still spent in the Mercury receive queues. For both ctls87 and slooow the receive queues never exceed 25 messages. But there are other applications which would overload this queues if not restrained.

Statistical Tools

We made use of histograms or frequency distributions to analyze the message and execution logs. Our data showed exponential distributions as one might expect with all the queueing of the application messages. Hence our histogram maker was designed to keep track of the messages with the largest delays. We found the longest delay times were much larger than expected. Time Warp was never designed to run in real time, however, some of these delays were excessive. The longest delay for a message (call it "M") in one run of ctls87 was 385 milliseconds. When message "M" arrived, there were no other messages waiting to be consumed on the receiving node. The Time Warp layer was "zipping ahead" while the message "M" waited 383 milliseconds. An object, red_div22, had sent lots of messages (64 of them) to itself, and at this point all these messages seemed incorrect. However, "M" was a message for red_div22, and had message "M" arrived a couple of milliseconds earlier, the Time Warp layer would not have "zipped ahead" more than five of the 64 messages. Identifying and reducing these delays has become one of our preformance goals.

Slooow

The application slooow is usually run with one slooow object per node. At each virtual time, each slooow object counts the number of messages it has and then sends that number to every other slooow object. However, each slooow object has the heart of a civil servant, and it sends multiple copies to each other slooow object. If a slooow object starts early, it will have too few messages and send the wrong number to everyone. If the slooow object is the last to finish, then it will cause the others to rollback. Thus these objects see-saw back and forth between being ahead and being behind.

Slooow is interesting in that its run time is not constant. In Time Warp version 1.09, run times of slooow were wildly erratic. It could even run slooower on 32 nodes than it did on one node. (The range on 32 node runs varied from 5 to 70 seconds.) Time Warp version 1.10 tamed slooow to be only mildly erratic. For example, certain run times under Time Warp 1.10 could range from 18 to 25 seconds. (Main diff between 1.09 and 1.10: packet size, buffer pools

Graphic displays of good run of slooow compared to bad runs of slooow didn't show any difference other than the run time. It was like the bad run was just a bit unlucky. Somehow critical measures arrived a bit late in the bad runs as compared to the good runs.

The m3plot which colored the positive messages green and the negative messages red provided the key to completely taming slooow. The max_ack parameter was slowing the anti-messages so much that green lines were vertical but the red lines were in the form of a cone. By using a different and much higher limit on the number of unacknowledged negative messages, the run time of slooow dropped down to a consistent 11 seconds.

Synchronizing the Clocks

Each node of the Mark III hypercube has a two microsecond countdown counter. These could be all started at once by a global interrupt. There were two problems about keeping time with these counters. Each was driven by a different clock cycle which could differ by as much as 50 tics per second. Although the counters could be read on the fly, borrows could still be in progress if the counter wasn't stopped. Trying to adjust for the time stopped (about 2 tics per reading) was not accurate enough. The solution was to read the counters twice in succession, if borrowing was in progress than the second time would be larger than the first, otherwise we had the correct time. To adjust for the different clock cycles, we used a timing run to determine the number of ticks per second on each node. Thus we easily obtained better than millisecond accuracy on runs of twenty to thirty seconds.

Conclusions

It is difficult to overrate the impact of graphics. Time Warp allows the simulation to go down wrong paths. We can get a idea of how many and how far these incorrect paths are executed. The graphs show the bottlenecks and the objects that have run off in the future. Sometimes the graphs provide a sort of negative information. For example, the slooow graphs showed that the slooow simulation never run away with itself, all the objects were near the same time. A better example of the negative information was that the difference between a good run and a bad run of slooow was only the length of the graph. This was another reason to make a study of the delay times.

Graphs easily show things which would be difficult to discribe with words. Indeed, the histogram program was designed so that its output could be graphed with Microsoft's Excel. The histogram program attempted to give near a 100 lines of output. We reduced 50K messages to a histogram of 100 lines and then graphed those lines.

Finally, and perhaps we should have known from the beginning, the performance of distributed simulations depends on the queueing ? delays. Long delays can make the whole simulation wait. However, there are still things about Time Warp behavior we do not understand.

Acknowledgements

We would like to acknowledge the help and encouragement of the Time Warp team and others, including Brian Beckman, Leo Blume, Maria Ebling, Phil Hontalas, David Jefferson, Roger Lee, Garry Paine,

Matthew Presley, Justin Magaram, Peter Reiher, Joe Ruffles, Jack Tupman, John Wedel, Fred Wieland and Herb Younger.

References

[Jefferson 87] Jefferson, David, *et al.*, "Distributed Simulation and the Time Warp Operating System." *ACM Proceedings of the Symposium on Operating System Principles*, (November 1987). 77–93.

[Wieland 88] Wieland, Frederick, Lawrence Hawley and Abe Feinberg, "Implementing a Distributed Combat Simulation on the Time Warp Operating System." *Proceedings of the Third Hypercube Conference,* (In Press).

P of Third Conference on Hypercube Concurrent Computer and Application
Vol 2 1269—1276
ACM Publication

D Jefferson ~~et al~~ The Status of the Time Warp operating system
Vol 1 738 – ~~744~~

~~86~~ 87 dis comp & TW and

28 36