Distributed Simulation and the Time Warp Operating System

David Jefferson (UCLA)
and
Brian Beckman, Fred Wieland, Leo Blume, Mike DiLoreto,
Phil Hontalas, Pierre Laroche, Kathy Sturdevant, Jack Tupman,
Van Warren, John Wedel, Herb Younger (Jet Propulsion Laboratory),
and
Steve Bellenot (The Florida State University)

Abstract

This paper describes the Time Warp Operating System, under development for three years at the Jet Propulsion Laboratory for the Caltech Mark III Hypercube multiprocessor. Its primary goal is concurrent execution of large, irregular discrete event simulations at maximum speed. It also supports any other distributed applications that are synchronized by virtual time.

The Time Warp Operating System includes a complete implementation of the Time Warp mechanism, and is a substantial departure from conventional operating systems in that it performs synchronization by a general distributed process rollback mechanism. The use of general rollback forces a rethinking of many aspects of operating system design, including programming interface, scheduling, message routing and queueing, storage management, flow control, and commitment.

In this paper we review the mechanics of Time Warp, describe the TWOS operating system, show how to construct simulations in object-oriented form to run under TWOS, and offer a qualitative comparison of Time Warp to the Chandy-Misra method of distributed simulation. We also include details of two benchmark simulations and preliminary measurements of time-to-completion, speedup, rollback rate, and antimessage rate, all as functions of the number of processors used.

1. Introduction

Discrete event simulations are among the most expensive of all computational tasks. One sequential execution of a large simulation may take hours or days of processor time, and if the model is probabilistic, many executions will be necessary to determine the output distributions. Nevertheless, many scientific, engineering and military projects depend heavily on simulation because it is too expensive or too unsafe to experiment on real systems. Any technique for speeding up simulations is therefore of great economic importance.

One obvious approach is to execute different parts of the same simulation in parallel. Most large systems that people want to simulate are composed of many interacting subsystems, and the physical concurrency in these systems translates into computational concurrency in the simulation. When the system to be simulated is extremely regular in its causal/temporal behavior, i.e. at each simulation time most objects in the simulation change state, and the real time needed to compute that change of state is approximately constant, then a time-stepped approach is reasonable. Cellular automata and clocked logic circuits fall into this category. Such systems can often be easily parallelized by executing different parts of the model *synchronously* in simulation time, so that all subsystems are simulated in parallel at simulation time 1, and then all in parallel at time 2, etc. However, it is a much greater challenge to extract concurrency from systems that are highly irregular in their temporal behavior. For them the event-driven paradigm (as opposed to the time-stepped) is appropriate.

In this paper we discuss the design and performance of the Time Warp Operating System (TWOS) a multiprocessor operating system directed toward parallel discrete event simulation. TWOS is a prototype system running on the 32-node Caltech/JPL Mark III Hypercube. It is not intended as a general-purpose operating system, but rather as an environment for any single concurrent application (especially simulations) in which synchronization is specified using virtual time [Jefferson 84]. Besides simulations, potential applications include large distributed databases, real time systems, and animation systems.

The main innovation that distinguishes TWOS from other operating systems is its complete commitment to an optimistic style of execution and to *process rollback* for almost all synchronization. Most distributed operating systems either cannot handle process rollback at all, or implement it in a limited way as a rarely-used mechanism for special purposes such as exception handling, deadlock breaking, transaction abortion, or fault

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specfic permission.

© 1987 ACM 089791-242-X/87/0011/0077 $1.50

recovery. But the Time Warp Operating System embraces rollback as the *normal mechanism for process synchronization*, and uses it as often as process blocking is used in other systems. TWOS contains a simple, completely general distributed rollback mechanism capable of undoing or preventing absolutely *any* side-effect, direct or indirect, of an incorrect action. In particular, it is able to control or undo such troublesome side effects as errors, infinite loops, I/O, creation and destruction of processes, asynchronous message communication, and termination.

The basic Time Warp mechanism [Jefferson 82] has been implemented or simulated several times before, but always on top of other systems, e.g. Lisp [Jefferson 82], Jade [Joyce 87], [Li 87], [West 87], [Xiao 86], or Simula 67 [Berry 86]. However, there are good reasons to believe that Time Warp should not run on top of another operating system, but should be the operating system. Rollback forces a rethinking of almost all operating system issues, including scheduling, synchronization, message queueing, flow control, memory management, error handling, I/O, and commitment. Since all of these are handled in some way by every operating system, building the Time Warp mechanism on top of another operating system would require having two levels of scheduling, two levels of process synchronization, two levels of message queueing, and so on. Ours is the first implementation where the Time Warp mechanism is the primary level of operating system on a true multiprocessor.

TWOS is written in C (with some assembly language in the lowest layer) and has been under development for three years at the Jet Propulsion Laboratory. It was originally designed for the Caltech Mark II hypercube, but now runs instead on the newer Mark III hypercube [Fox 85], [Peterson 85], and also on a network of seven Sun workstations. The older Mark II hypercube, a forerunner of the Intel iPSC, was constructed of 32 nodes, each of which contained an Intel 8086 processor and an 8087 floating point coprocessor with 256K bytes of RAM. The nodes were connected by bidirectional channels in the topology of a 5-dimensional Boolean hypercube. The newer Mark III hypercube nodes consist of one 16 MHz Motorola 68020 processor for computation, a 68881 floating point coprocessor, a second 68020 processor dedicated to internode communication, 4 megabytes of dynamic RAM, and internode communication channels with a 64M bit/sec peak transfer rate. The measurements given later in this paper were all made on the 32-node Mark III at JPL. When a larger 128-node machine is completed later this year, we will extend our measurements for larger simulations to that scale.

TWOS is a single-user system that supports distributed applications composed of processes communicating by message. It can use any number of processors, not just a power of two, since the hypercube topology is rendered invisible above the lowest level of software. Each node is multiplexed so that as many processes can share a node as can fit in its memory.

TWOS is not intended to support general time sharing among independent processes. Furthermore, since it is still a prototype, it has some significant limitations. It does not yet permit dynamic creation of processes at runtime, nor dynamic migration of processes for load management. Because of architectural limitations there is only low bandwidth output from the application, and no interactive input. TWOS applications today operate in a simple download-and-go manner.

TWOS retains the same general modular decomposition as an ordinary distributed operating system; it differs only in that different algorithms are used inside those modules. Although it has highly unusual processor scheduling, memory management, process synchronization, message queueing, and commitment protocols, they each play the same familiar roles as they do in other distributed operating systems.

In the remainder of this paper we will describe the general issues of Time Warp and virtual time. Then in Section 3, 4, and 5 we describe the programming model imposed on users by TWOS, the TWOS calls used to program a simulation, and give an small example simulation intended for execution under TWOS. In Section 6 we give a qualitative comparison between the Chandy-Misra approach to distributed simulation and the approach taken by Time Warp. In Sections 7 and 8 we talk specifically about the TWOS implementation, first its structure and then its performance. Section 9 offers some conclusions and future directions.

2. Time Warp and Virtual Time

2.1 Background

The basic Time Warp mechanism, which is at the heart of TWOS, was invented by Henry Sowizral and David Jefferson (then at the Rand Corporation and the University of Southern California respectively) as a method for speeding up discrete event simulations [Jefferson 82]. The major contribution of that work was the idea that *process rollback* should be considered a fundamental synchronization tool for distributed simulation. Before Time Warp was described most researchers probably believed that general rollback in an asynchronous environment was either fundamentally impossible to implement, or prohibitively expensive. Time Warp offered a simple and elegant implementation based on the notions of *antimessages* and *annihilation*.

Later the theory of *virtual time* was introduced as a paradigm for organizing and synchronizing certain kinds of distributed systems [Jefferson 85]. Virtual time is a global temporal coordinate axis defined by the application as a measure of its *progress* and as a scale against which to specify synchronization. The Time Warp mechanism was then reinterpreted as being not just a distributed simulation mechanism, but as the primary implementation for the broader abstraction of virtual time. There is a strong space-time symmetry

between the theories of virtual memory and virtual time, and between their respective implementations, demand paging and the Time Warp mechanism.

2.2 When is Time Warp needed?

Time Warp may not be appropriate for every distributed application. But those applications whose behavior can be specified using an artificial time scale (e.g. logical time, simulation time) are candidates. Even then there are protocols simpler than Time Warp that may perform better under certain conditions. For example, when a simulation can be described as a static network of interacting processes such that most of the arcs in the network have an approximately equal amount of message traffic, then the Chandy-Misra distributed simulation mechanism [Chandy 81] may perform better than Time Warp. (See the cautionary study [Reed 87].) Whenever "time slip" is not important to the analysis of a simulation model, the SRADS mechanism [Reynolds 82] is simpler and may perform better. However, Time Warp seems to have the widest applicability with the fewest restrictions, and seems to be the only choice for applications that contain instances of the following *virtual time synchronization problem.*

2.3 The Virtual Time Synchronization Problem

Assume that an application is composed of processes that communicate by timestamped messages. One such process, together with incoming messages from many different senders, is shown in Figure 1. The figure shows several messages that have already arrived and are queued in increasing timestamp order. All incoming messages are funnelled into a single input queue.

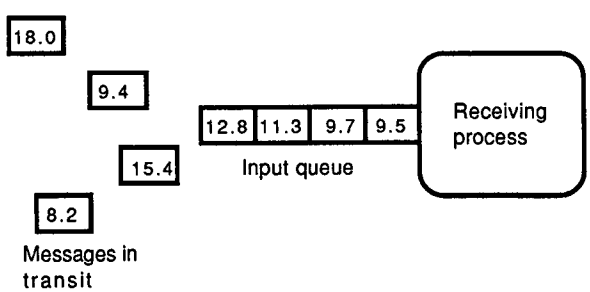

Figure 1: Virtual Time synchronization problem

The message timestamps are not real times, but virtual times, and are assigned by the senders to specify the order in which the messages must be processed. We do not assume that messages will arrive nicely in increasing timestamp order. Although all timestamp-driven synchronization mechanisms perform better when messages arrive in approximately the correct order, in general we must assume that they might arrive in *any* order. Furthermore, we do not know anything about the subset of the possible timestamps that will actually appear on arriving messages. Timestamps may be real numbers, and it is not the case that successive timestamps must be separated by some minimal difference, so we cannot even bound the number of messages that might arrive bearing timestamps between t_1 and t_2.

The virtual time synchronization problem then is this: How can the operating system control the execution of a process so that it receives its messages in nondecreasing timestamp order *and is guaranteed to make progress?* We might try examining the next unprocessed message in the input queue. If it is the 'true next' message, i.e. the message with the next highest timestamp from among all those that have arrived *or will ever arrive*, then we should execute it; but if it is not then we should block the process until the 'true next' message does arrive. Unfortunately this strategy cannot work because, since timestamped messages can arrive in arbitrary order and we cannot know what timestamps will appear, there is no way to *recognize* the 'true next' message when it does arrive.

In general, it is impossible to solve the virtual time synchronization problem using local information if the only synchronization tool allowed is process blocking. But with a stronger synchronization primitive, namely process rollback, we can solve it.

2.4 Sketch of the Time Warp mechanism

The Time Warp mechanism [Jefferson 82, 84] takes an *optimistic* approach, and *assumes* at each moment that the messages already in the input queue are the 'true next' ones and proceeds accordingly to execute them in timestamp order. Of course, new messages can arrive asynchronously during this execution, and as long as they have timestamps higher than the highest timestamp processed so far, the arriving messages are simply enqueued in their proper order. But whenever a message arrives with a timestamp t less than some that have already been executed, then the optimism was unjustified and Time Warp must

(a) roll back the process to a time just before virtual time t;

(b) execute the new message at virtual time t; and

(c) start re-executing messages with timestamps greater than t, again in timestamp order, cancelling all of the effects of any output messages that were sent after t during the last forward execution but were not re-sent in this one.

In order to support rollback TWOS regularly takes a snapshot of the state of each process. These states are stored in a queue associated with the process and are reinstated whenever it is necessary to roll back. The difficult part of rollback is the implementation of step (c), the cancellation of the effects of messages that should never have been sent. To accomplish this Time Warp introduces the concept of *antimessages.*

Every event-, query-, and reply-message (the three kinds of messages that are exchanged among processes) is considered to have a *sign*, either + or -. Two messages that are identical in all fields but of opposite signs are said to be antimessages of one another. Whenever a process *P* requests a message to be sent, TWOS actually creates a message-antimessage pair. The positive message is delivered to the intended receiver's *input queue*, while the negative one is retained by *P* in its *output queue*. As long as *P* does not roll back because of a message arriving with a timestamp in its past, the negative messages remain in the output queue and are eventually garbage-collected as part of commitment. However, when *P* rolls back to simulation time *t* and executes forward again, it will usually take a different execution path and send a different sequence of output messages this time as it executes past simulation time *t* than it did last time it executed past simulation time *t*.

As a process executes forward after time *t*, TWOS compares every message-send request from *P* with the old (negative) messages in *P*'s output queue. If a new message is already represented in the output queue both it and its antimessage are discarded, since the receiver already has a copy. For any new message not represented in the output queue TWOS transmits its positive copy and saves its negative copy in the output queue. Finally, any (negative) message in the output queue that is not re-requested for transmission during the new forward execution of *P* must be incorrect, and TWOS must *cancel* the corresponding positive message, meaning that all of its side-effects, direct and indirect, must be undone.

Positive and negative messages are treated exactly symmetrically by TWOS in all respects. The only significance of the signs is this: whenever a message is inserted into a queue that contains its own antimessage, the two messages annihilate and the queue gets shorter. Thus, the queueing discipline in TWOS, which is used universally for timestamped messages, satisfies the following algebraic laws for any queue *Q* and any positive or negative message *m*:

$$-(-m) = m$$
$$Insert(Insert(Q,m), -m) = Q.$$

With this understanding of antimessages, the rest of the Time Warp cancellation mechanism is simple: to undo the side-effects of a positive message *m* from *P* to *Q*, it suffices to remove the antimessage *-m* from *P*'s output queue and transmit to *Q*'s input queue. There are basically two cases to consider:

(1) If *-m* arrives in *Q*'s future, then it will annihilate with the *m* in P's input queue and the cancellation is finished;.

(2) If *-m* arrives in *Q*'s past, it will cause *Q* to roll back, but it will also annihilate with *-m*, so that when *Q* executes forward again neither *+m* nor *-m* exist *Q* will not see either of them.

Although we do not have space to demonstrate it here, this cancellation mechanism (called lazy cancellation [Gafni 85]) works under any circumstances and guarantees progress of the simulation as a whole. If the messages tend to arrive at a process in *almost* correct order, as they do in actual practice, then there will be comparatively little rolling back necessary. In fact, it is essential that messages arrive in almost correct order on the average. "Almost correct order" means that the number of inversions in a long sequence be only linear in the length of the sequence, rather than quadratic (which is the worst case). Essentially all simulations of real physical systems can be expected to have this behavior if run long enough.

3. The TWOS Programming Model

The Time Warp Operating System supports a simple object-oriented programming model with a global process name space. Each process has a 20-character name that is globally unique. Any process can send a message to any other process at any time simply by referring to the name of the receiver. There is no notion of a 'channel', 'pipe', or 'connection' between two processes, and there is no need to 'open' a connection before sending messages. This model was chosen to provide maximum flexibility in the design of complex simulations, so that it is not necessary to declare statically which processes will communicate with each other.

A process is logically composed of four parts, shown in Figure 2 in a Pascal-like syntax, although in fact we write them in C according to a discipline that approximates this structure.

The `StateVariables` have scope global to all four entry sections and retain their values between incoming messages.

The **`Initialization Section`** is a code segment that is executed once-only at initialization time (when virtual time is -∞) and whose main purpose is to initialize the StateVariables. It may send event messages with finite timestamps, but they will not be received until all initialization sections are complete. An **`Initialization Section`** may not send query messages, however, since they have the effect of requesting information from earlier in virtual time, and there is no time earlier than -∞.

The **`EventMessage Section`** is invoked whenever a set of event messages is to be processed. It usually modifies the StateVariables and sends one or more query or event messages.

```
begin
   var StateVariables;
       { Variables whose value is retained
         across events }

   Initialization Section:
       begin
           { Code to be executed during
             initialization at time -∞; }
       end;

   EventMessage Section:
       begin
           { Code to be executed when an
             event message is processed;

             Can have side-effects; can send
             Event Messages and Query
             Messages; }
       end;

   QueryMessage Section:
       begin
           { Code to be executed when a
             query message arrives;

             Must be side-effect free and
             can send only Query Messages;

             Must send exactly one Reply; }
       end;

   Termination Section:
       begin
           { Code to be executed at
             termination at time +∞; }
       end;
end.
```

Figure 2: Structure of a TWOS process

The **QueryMessage Section** is invoked to process a query message. It must be side-effect free, and thus cannot modify the state variables or send any event messages (because they would cause side-effects). It may, however, send additional query messages. The **QueryMessage Section** is required to generate exactly one reply message to the query message that invoked it.

The **Termination Section** is invoked when the simulation is ended, at virtual time $+\infty$. Its main purpose is to allow final statistics to be output before terminating execution. It may send query messages, but not event messages since the latter would have to be processed later in virtual time and there is no time later than $+\infty$.

Any of the four entries may declare local stack variables, but the values of those variables are not preserved across invocations. Only `StateVariables` retain their values across invocations.

Except at initialization and termination the only time a process executes is to handle an incoming message. Processes are thus *message driven*, and do not execute between incoming messages. Of course, a process may send itself an event message. The processing of an event message is called an *event*, using terminology drawn from simulation.

There are two significant restrictions imposed on the behavior of processes. First, a process must be rigidly *deterministic* in its input-output behavior. In order to prevent a domino effect during rollback it is vital that a process, when rolled back and restarted in an earlier state with the *same* input messages as before, should generate exactly the *same* output messages. This restriction is a theoretical necessity, but it should not be exaggerated. For example, there is no problem with the use of pseudorandom number generators; they can be used freely as long as all random seeds are among the `StateVariables` so that their values can also be rolled back when necessary.

The second restriction is that processes should not use heap storage (e.g. `new()` in Pascal or `malloc()` in C). To support rollback the entire state of a process must be saved from time to time, and heap storage makes state-saving difficult and/or slow. This restriction is just a performance issue, and although we have not found it to be too burdensome yet, it is a potential liability.

The programming restrictions in this model are not enforced by TWOS. They are the kind of restrictions that should be enforced instead by linguistic mechanisms in an object-oriented simulation language. For now we rely on the discipline of our application programmers.

Processes request output by sending event messages to special operating system processes whose type is `stdout`, not by making operating system calls. This convention is convenient, but it is also necessary, because in an environment where rollback can happen at any time it is possible that an output request will have to be *unrequested*. Time Warp must buffer output requests, and not execute them until they can be *committed*. Discussion of commitment is deferred until Section 7.

4. TWOS interface

Here we present the system calls available to simulation programmers wishing to run under TWOS. These descriptions have been slightly simplified, primarily by leaving out error parameters. We will discuss their implementation in the next section. In what follows we will refer to the current virtual time, i.e. the virtual time at which the call is made, as `Now`. The underlined parameters are modified by the call.

Time Warp Operating System calls:

Me (MyName)
Sets the `MyName` parameter to the 20-character name of the calling process.

VirtualTime (VTime)
Sets the *VTime* parameter to *Now*, i.e. the current simulation time.

SendEventMessage (ReceiveTime, Receiver, Text)
This call transmits an event message containing *Text* to the process named *Receiver*, and schedules it to be received at virtual time *ReceiveTime*. It can only be invoked from the **EventMessage Section** of the sending process, and then only if *ReceiveTime* is greater than or equal to *Now*.

At virtual time *ReceiveTime* the operating system will invoke the **EventMessage Section** of the receiving process, giving it access to this message and all other messages arriving at the process *Receiver* with the same receive time. Although *ReceiveTime* can equal *Now*, there must not be a cycle of processes each of which sends a message to the next with *ReceiveTime* equal to *Now*. Semantically the behavior of such a cycle is analogous to deadlock, though under TWOS it will cause infinitely repeated rollback instead. A process may send a message to itself, but if it does so it must be with a *ReceiveTime* strictly greater than *Now* so as not to violate the rule about cycles.

SendQueryMessage (Receiver, Text, Reply)
This primitive transmits a query message containing *Text* to the process named *Receiver*. It acts much as a remote, side-effect free function call to another object to obtain information about its state at time *Now*. The query message is scheduled to be received *Now*, i.e. at the current virtual time. It then blocks the calling process to await the reply, which also comes back at virtual time *Now*, and whose content is delivered into the buffer *Reply*.

At any given virtual time, query messages are processed before event messages; hence the reply to a query message sent at time 100 is based on information at the receiver *before* any event at time 100 is executed. In particular, if a process sends a query message to itself from part-way through the execution of its own **EventMessage Section**, the reply will be based on the state variables as they were just *before* the **EventMessage Section** *started* execution.

It is permitted to have a cycle of query messages (all within the same virtual time). The behavior is analogous to recursive invocations of the *QueryMessage* sections of the processes involved in the cycle.

SendReplyMessage (Text)
This call *must* be invoked once and only once for each invocation of the **QueryMessage Section** of a process. It sends a reply message containing *Text* back to the sender of the query, to be received at virtual time *Now*. The reply is uniquely associated with the query message that caused it to be generated, and is analogous to the return of a remote function call. When it arrives, the reply will restart the receiver (i.e. the sender of the query) at the point in the *EventMessage* section or **QueryMessage Section** where it was suspended.

MCount (n)
Several event messages may arrive at a process at the same virtual time, and *MCount* sets *n* to the number of such messages, typically one. It can only be invoked from the **EventMessage Section** of a process.

ReadEventMessage (k, Text)
This call reads the text of the *k*'th event message that arrived with timestamp of *Now* into buffer *Text*. It can only be invoked from the *EventMessage* section.

ReadQueryMessage (Text)
This call reads the text of the current query message, and can be invoked only from the *QueryMessage* section of a process. Since the **QueryMessage Section** of a process must be side-effect free, only one query message at a time is processed even if several queries arrive at the same process with the same virtual time.

5. How to write a simulation under TWOS

In this section we illustrate a simulation designed to run under TWOS. We will write a very simple simulation of one of the servers in a queueing network shown in Figure 3. There is one customer source, A, and three servers B, C, and D. Upon leaving station B 90% of the customers (randomly selected) go to station D, and only 10% to station C. We will assume that all sources and servers are exponential with parameter λ, and that queueing is FIFO.

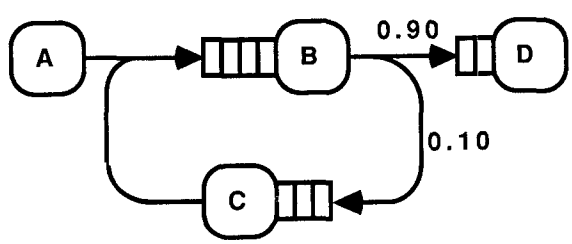

Figure 3: Simple queueing network

The natural decomposition of this network is as four processes, one for each of the sources and servers. The following code fragment will implement server process B. Bear in mind that this pseudocode is presented only for illustration of the synchronization and message handling features in TWOS.

```
begin { Logical process B }
{ Arrival and service rate }
const λ = 1.0;
{ State variables }
var  Q_Len : integer;
        { Current queue length }
    Seed : integer;
        { Random seed }
    Cum_Q_Len : real;
        { Cumulative, time-weighted
          queue length; for calculating
          mean queue length at end }
    Last_Ev_Time : VirtualTime;
        { Simulation time of event
          preceding this one }

Initialization Section:
begin
  { Code to be executed during initial-
    ization; }
  Q_Len := 0;
  Seed := 1234567;
  Cum_Q_Len := 0.0;
  Last_Ev_Time := 0.0;
end;

EventMessage Section:
begin
  var i : integer; { Loop counter }
      n : integer;
          { Number of event msgs arriving
            at same virtual time }
      Type : string;
          { Type of event, either 'End-
            Service' or 'CustomerArrival' }
      Current : VirtualTime;

  { Read current simulation time }
  VirtualTime(Current);

  { More than one event may be scheduled
    at this simulation time--up to two
    customer arrivals and one service
    end. Do all, in arbitrary order. }
  MCount(n);
  for i := 1 to n do
    begin
      { Find out what kind of event }
      ReadEventMessage(i,Type);
      case Type of
        'EndService':
          begin
            { Send customer onward }
            if random(Seed) < 0.9
              then
                SendEventMessage(Current,
                  'D','CustomerArrival')
              else
                SendEventMessage(Current,
                  'C','CustomerArrival');
                Cum_Q_Len :=
                  Cum_Q_Len + Q_Len *
                  (Current-Last_Ev_Time);
            Last_Ev_Time := Current;
            Q_Len:= Q_Len - 1;
            if Q_Len > 0 {Start service}
              then { Message to self }
                SendEventMessage(Current
                  + ExpRandom(Seed,λ),
                      'B','EndService');
          end;
        'CustomerArrival':
          begin
            if Q_Len = 0 {Start service}
              then
                SendEventMessage(Current
                    +ExpRandom(Seed,λ),
                      'B','EndService');
                Cum_Q_Len :=
                  Cum_Q_Len + Q_Len *
                  (Current-Last_Ev_Time);
            Last_Ev_Time := Current;
            Q_Len:= Q_Len + 1
          end
      end {of case stmt }
    end { of for stmt }
end; { of EventMessage Section }

QueryMessage Section:
begin
  { Empty. No queries in this example. }
end;

Termination Section:
begin
  print('Mean queue length of B = '
        Cum_Q_Len / Last_Ev_Time)
end;
end. { of logical process B }
```

The explanation for this code is as follows:

5.1 State variables

There are only four variables in the state of process *B*. Two of those, *Seed* and *Q_Len* actually represent the state of the system being simulated. *Seed* is the random seed driving both the service time distribution and the decision about where a customer goes when it leaves *B*. *Q_Len* represents the length of the queue of customers lined up for service at *B*. In this case, since all customers are identical and queueing is FIFO, the state of the queue can be adequately represented by just its length.

The other two state variables, *Cum_Q_Len* and *Last_Ev_Time*, are part of the instrumentation of the model, and are necessary to calculate the main performance parameter of interest, the mean queue length.

5.2 Initialization Section

In this code all four state variables are initialized. This initialization is considered to occur at simulation time $-\infty$, i.e. before any events have taken place.

5.3 EventMessage Section

This code is invoked whenever an event is to be processed for *B*. An event message arriving at process *B*, signals one of two kinds of events. If the text of the message is *'CustomerArrival'* it signals the arrival of a customer, either from *A* or *C*. If the text is *'EndService'* it indicates that a service period has completed at *B* and that the customer just served should be moved along to either *C* or *D* while the one at the head of the queue (if any) should begin service.

The first thing the **EventMessage Section** does is read the simulation clock into the variable `Now` (using the `VirtualTime` call). This is necessary for the calculation of mean queue length.

The **EventMessage Section** then checks how many event messages have arrived at this simulation time, using the `MCount` call. This is necessary because in this model up to three distinct event messages may arrive at process *B* at the same simulation time, since one customer's service may end at exactly the same simulation time that two other customers arrive from *A* and *C*. The simulation of those actions together constitute a single *event* in TWOS by virtue of the fact that they occur at the same place (*B*), and the same simulation time. However, in this model the logic is such that any such compound event can be simulated by processing the (up to) three event messages serially, in any order. This is why the **for** loop that acts as the main control structure of the **EventMessage Section**.

In the case that an event message signals the end of some customer's service three things must be done. First, the customer must be sent on to the next queueing station. This is done by the `SendEventMessage` call. Note that the `'CustomerArrival'` event message is scheduled to be received at simulation time `Now`, i.e. at the same time as the current simulation time. This is because in a queueing model no time elapses between a customer's completion of one service and its entry into the next queue.

Second, the queue length must be decremented (to reflect the departing customer) and the statistical variables must be updated.

Finally, if the queue is still non-empty after one customer has left, then the service of the next customer must start. An event message indicating `'EndService'` is sent by *B* to itself, scheduling the time that the service period will be over.

If, on the other hand, the event message signals the arrival of a new customer, then only two steps are necessary. First, the queue length must be incremented and the associated statistics updated. Second, if the arrival of this customer changes the queue from empty to nonempty then the arriving customer must immediately start service and the end of his service must be scheduled.

All of this computation in the event section takes place in one instant of simulation (virtual) time, and thus constitutes a single atomic action.

5.4 Query Section

This model does not need to use the TWOS query mechanism, and thus this section is empty.

5.5 Termination Section

The termination section is executed after the simulation proper is completely finished. In this case all that needed is to calculate and print the final statistic.

6. Comparison with the Chandy-Misra approach

The best known methods for distributed simulation are based on ideas by Chandy and Misra. Unfortunately, no comprehensive quantitative comparisons between their techniques and Time Warp have yet been performed, primarily because of the sheer size of the undertaking. But here we will try to give at least some qualitative comparison between the two.

The Chandy-Misra methods share with Time Warp two requirements: (1) a simulation should be decomposed into *logical processes* (which we have been calling simply *processes*) each of which represents a *physical process*, i.e. a subsystem of the model to be simulated; and (2) the logical processes communicate only via timestamped event messages, each of which represents an interaction between subsystems at a particular simulation time. Both methods are asynchronous, in that they allow some processes to be ahead in simulation time while others lag behind in order to achieve greater concurrency.

But there is little resemblance beyond these basic facts. Time Warp and the Chandy-Misra methods implement different paradigms of discrete event simulation; they require different amounts of static knowledge about the model to be simulated; they differ completely in their approach to the critical mechanisms of synchronization; and they perform best in different regions of the space of all simulations. The rest of this section will cover these differences in more detail.

6.1 Differences in simulation paradigm

A program written for Chandy-Misra is not directly runable under Time Warp, and vice-versa, because they represent different views of discrete event simulation. A logical process LP receives a sequence of timestamped event messages $M_1, M_2, ..., M_n$, with timestamps $t_1 < t_2 < ... < t_n$ respectively. Under the Chandy-Misra mechanism, when an event message M_i with timestamp t_i is received by LP, LP simulates the behavior of physical process PP over the simulation time interval t_{i-1} to t_i, i.e. the *interval preceding* t_i. This sometimes requires event messages to be sent to other processes with timestamps strictly less than t_i. The logic of the method, in particular the requirement that messages be sent in increasing timestamp order along each channel (see below), guarantees that there can be no cycle of interactions allowing an event to effectively cause changes in the past.

In contrast, when a Time Warp logical process LP receives an event message with timestamp t_i, it simulates a *single instant* in the behavior of the physical process PP, not an interval of its behavior. During an event it can send additional event messages with timestamps greater than or equal to t_i, *but not lower.*

As a result, a programmer tends to imagine the behavior of his model as organized into a sequence of intervals in order to use the Chandy-Misra mechanism, but imagines it as a sequence of discrete events (instantaneous) in order to use Time Warp. Both paradigms are legitimate, but they force different idioms on the programmer for certain standard simulation effects, such as effectively preempting (or cancelling) a previously scheduled event.

6.2 Static restrictions

Under the Chandy-Misra mechanism the model is represented as a *network* of logical processes with discrete channels connecting them. A logical process may have any number of incoming or outgoing arcs, but the size and topology of the network is usually viewed as *statically declared*. This is not a trivial restriction; we cannot simply consider the network to be fully connected and then use only a subset of the channels, because (as we shall see) in all variations of the Chandy-Misra mechanism there is considerable overhead associated with unused channels.

Under Time Warp there is no network of channels connecting the processes. Instead, any process may send a message to any other at any time. The interaction topology is completely dynamic. It is thus easy to simulate systems such as colliding pool balls, war games, or particle interactions, that have the property that which objects interact with which others is not statically determined.

Another difference between the two methods concerns the order in which messages can be sent between two processes. Under the Chandy-Misra mechanisms a process must send messages in increasing timestamp order along each of its output channels. See Figure 4. If a process A at time 80 sends a message with timestamp 100 along channel c to process B, then A can never again send a message the same channel with a timestamp less than 100. It is quite common in simulations for a process such as A to want to send later, at time 90, a message to B with timestamp 95, effectively preempting (or modifying) the effect of the message with timestamp 100. This is not impossible under the Chandy-Misra mechanism, but the preempting message cannot be sent to B along the same channel c; it must instead be sent along another channel c'.

It may seem that establishing a second channel to handle the few cases when it is desirable to send messages out of order is at most a minor inconvenience. However, when sending messages out of order is rare, then the second channel c' established to handle that case is a

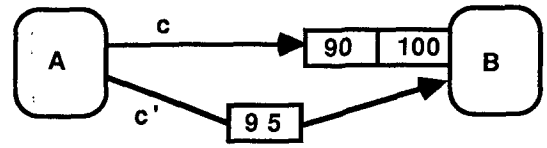

Figure 4: In the Chandy-Misra mechanism messages must be sent in timestamp order along each channel

rarely used, and it is exactly in those cases that the Chandy-Misra mechanisms have their largest overheads.

The Time Warp mechanism does not require messages from A to B to be sent in increasing timestamp order; they may be sent in any order, although usually the more inversions there are in the sequence, the more often the receiver will have to roll back.

6.3 Synchronization mechanisms

The Chandy-Misra and Time Warp mechanisms differ most significantly in their synchronization mechanisms. The Chandy-Misra mechanisms are *conservative*, in that a process is not allowed to receive a message with timestamp t until it is certain that no message will ever arrive with a timestamp less than t. In practice this means that a process must usually be blocked as long as any of its input queues is empty. Thus, if one of the input queues to process P is rarely used, then P will remain blocked most of the time.

Under Time Warp a receiving process does not have a separate input queue for each possible sender. Instead, all incoming messages are funnelled into a single timestamp-ordered queue. Time Warp is *optimistic* in that it allows a process to receive a message at time t with no guarantee that there will not be another with a timestamp less than t. Usually this optimism will be justified, but sometimes it will not; when a message does arrive with timestamp $t' < t$, the receiving process must roll back and cancel all incorrect side-effects back to time t'.

The major complication with the Chandy-Misra mechanism is that its basic policy of blocking a process when one or more of the input queues is empty often leads to deadlock. Any cycle in the network of interacting processes can be the seed of a local deadlock, which then tends to expand to become global. The major challenge in implementing the Chandy-Misra mechanisms, and the main differences among them, is in dealing with deadlock.

Many approaches have been studied for either avoiding deadlocks (e.g. the null message technique) or for breaking them (e.g. the circulating token technique), but no one method seems yet to work well in all cases [Misra 86]. Almost certainly a combination of mechanisms, dynamically selected, will be necessary in any complex, irregular simulation.

With the Time Warp mechanism there is no need for deadlock avoidance or deadlock breaking. There is a global mechanism for GVT calculation that is necessary for commitment of irreversible actions, but its invocation is driven by storage management and response time requirements, and not by the need to avoid deadlock.

6.4 Domain of high performance

The Time Warp mechanism requires considerable overhead in the form of state-saving and the handling of antimessages in order to make rollback possible. When rollback occurs, additional processor and communication resources are consumed. In contrast, overhead of the Chandy-Misra mechanism is almost entirely in the management of deadlock by whatever mechanism is in use.

These differences are not simply in the amount of overhead, but in the kind. Time Warp incurs its overhead in those parts of the model where there is activity. No state saving or message communication is necessary in those parts of the model that are quiescent. In addition, the overhead of rollback, when it does occur, occurs off of the critical path of the computation, i.e. not in those processes that are farthest behind in simulation time.

In the Chandy-Misra mechanism, however, most of the overhead is incurred where there is *inactivity* in the model. Deadlocks and unnecessary process blocking will be most common where there are unused or infrequently used channels, and hence it is around the inactive channels that there is the greatest need for null messages or deadlock detection tokens.

Although experimental verification is lacking, it would seem from the above discussion that the Chandy-Misra mechanism would probably be superior to Time Warp when the simulation can be decomposed into a statically-defined network of logical processes in which all or most of the logical channels have regular event message traffic. Where there are numerous pairs of processes that can interact but do so rarely, or (which amounts to the same thing) if the topology of communication changes dynamically, then Time Warp will be likely to perform better.

7. Time Warp as an Operating System

The Time Warp mechanism is described more fully in other papers, especially [Jefferson 82] and [Jefferson 85]). Our purpose here is to describe Time Warp in its role as an operating system, in contrast to other operating systems. We do so briefly, by comparing it module by module with more standard operating systems.

TWOS is structured as shown in Figure 5.

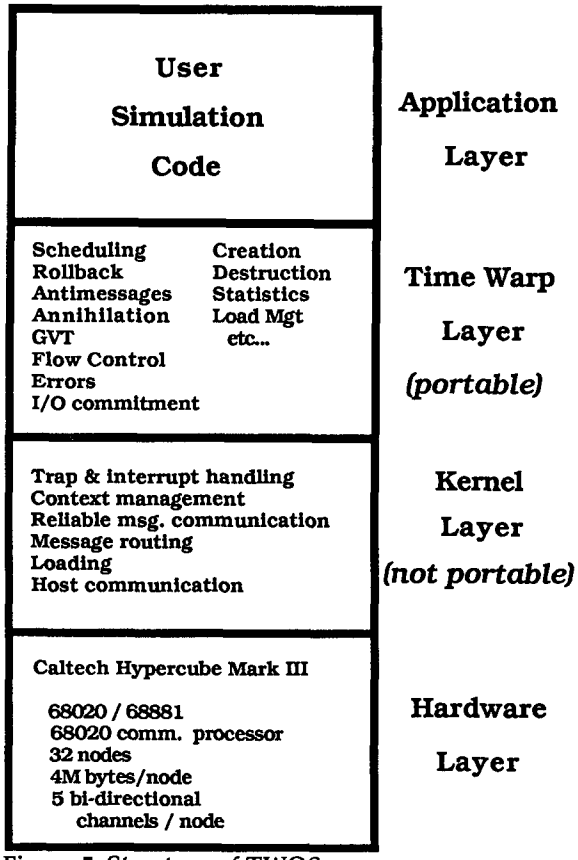

Figure 5: Structure of TWOS

The operating system is represented by the two middle layers. The lower of the two, called the kernel, provides basic interrupt handling, context management, and low-level message communication primitives. None of the kernel is specific to simulation; it is in theory quite generic. It is not necessarily portable, and some of it must be reimplemented on every machine that TWOS is to run on. Some of it is in assembly language.

The upper layer, called the executive layer or the Time Warp layer, contains all of the code that implements the Time Warp mechanism. This code is entirely written in C and is portable. (It has already been ported from the Mark II, 8086-based hypercube to the Mark III 68020-based machine.)

As described earlier, TWOS has the same overall structure as a conventional distributed operating system, but each of its parts contain very unconventional algorithms. The following subsections describe these differences in more detail.

7.1 Processor Scheduling

Most distributed operating systems that allow more than one process per node schedule each processor according to some time-sliced, multiple-queue mechanism with round-robin scheduling within each queue. The multiple queues distinguish high priority from low priority processes, or compute-bound from I/O-bound processes, etc., and different length time slices may be associated with each queue.

Time Warp's scheduling algorithm is not timesliced at all, but *pre-emptive lowest virtual time first.* Time Warp *always* executes the eligible process that is at the lowest virtual time, with arbitrary choice to break ties. A process will execute indefinitely, as long as it has the lowest virtual time of any process on its processor. If, while executing one process, a message arrival and rollback causes another to become farthest behind on that processor, then the first process is pre-empted and the second one runs.

In general a process is always eligible to execute as long as it has unprocessed messages remaining in its input queue. The only exception is in the handling of queries. When a process sends a query message it is suspended until either (a) the reply message arrives, in which case the process is resumed, or (b) another event or query message arrives with a lower timestamp, in which case the process rolls back out of the suspended state to whatever virtual time is appropriate.

7.2 Message Queueing

Most operating systems use FIFO message queueing at every stage of routing, for reasons of simplicity and fairness, and because preservation of message order is often required in application-level communication primitives.

But under TWOS messages are not necessarily processed in the order sent; they are processed in timestamp order. Hence, they are always enqueued, both during intermediate routing and at their final destination, in increasing timestamp order. Messages with low timestamps get preferential treatment and faster forwarding service than other messages, a convention that is consistent with the preferential scheduling treatment of processes with low virtual times.

Negative messages traveling in the forward direction and positive messages traveling in the reverse direction (for flow control) get additional priority since they will likely free space and prevent wasted effort at their destinations.

Negative messages cancel with their positive counterparts whenever they are found in the same queue. In principle, this can be an intermediate forwarding queue, but this latter embellishment is not yet implemented.

7.3 Process Synchronization

Most distributed operating systems provide various blocking-oriented message receive primitives, i.e. if no message of the class being waited for has arrived, then the process blocks until one does. In those systems that recognize remote procedure calls or transactions there may be an abortion mechanism as well.

Under Time Warp a process blocks only if it has no unprocessed messages in its input queue or if it is waiting for the reply to a query. But it does a full rollback immediately (even if executing) whenever a message arrives with a timestamp less than the process' current virtual time. A process can roll back out of the blocked state, then execute forward and reenter the blocked state.

7.4 Flow Control

In most operating systems the only aspect of storage management sensitive to the relative speeds of the processes is message flow control, and there are various protocols for blocking a sender so that it does not overflow the memory of the receiver.

For several reasons, however, flow control under Time Warp is much more critical and difficult. First, Time Warp must concern itself not only with incoming messages filling up memory, but also with outgoing messages (of which the sender keeps a negative copy) and saved states as well. Second, because any process can send a message to any other with no explicit channels, flow control cannot be done on a channel basis. It must be done on a process or node basis. Third, most operating systems delete a message and free storage as soon as the receiver has read it. But TWOS cannot do that because a rollback may require that the message be read again. Finally, because it executes most efficiently when there are many back states and messages available to support rollback TWOS generally attempts to run with memory almost completely full. This puts additional stress on the flow control and storage allocation mechanisms.

Time Warp's basic flow control tool is *message sendback* [Gafni 85]. There is not space here to describe the protocol in full, but it is based on the idea that when memory is full and space is needed for a new arriving message with timestamp t, then one way to make room is to find a message in an input queue with virtual send time greater than t, and return it to its sender, i.e. *unsend* it. This will likely cause the sender to roll back to a state before it sent the message, but it will then execute forward again and resend the message later. Although message sendback may seem unusual, it is merely the communication analog of process rollback.

7.5 Commitment

Some operations, such as output, destruction of an object, discarding an old state or message, and process

termination are computationally irreversible and thus require *commitment* from the operating system before they can be performed. Many operating systems need no commitment protocol at all, and just perform irreversible actions on request. Others have commitment protocols designed to guarantee atomicity of transactions or remote procedure calls.

Time Warp's commitment requirement is that no irreversible action can be committed at virtual time t until all events that might affect the action or cause its cancellation, namely those at virtual times less than or equal to t, are complete. Therefore, from time to time TWOS calculates an estimate of the quantity called Global Virtual Time (GVT), defined to be the minimum virtual time of any uncompleted event or message transmission in the application. Once GVT is known to be greater than or equal to some value t, then Time Warp can commit all output requests at virtual time less than t, release all message and state buffers with virtual times less than t, and report to the user any errors outstanding from virtual times less than t.

8. The Performance of TWOS

We are now engaged in a lengthy performance tuning and measurement program for TWOS. Since the goal is to execute multiprocess simulations as quickly as possible, the primary evaluation criterion is the time to completion of benchmarks. We are also interested in secondary performance measures, such as memory usage, the fraction of processor time spent in activity that ends up being rolled back, the fraction of messages that are negative, and the net processor utilization. Much of that data we do not yet have.

All of the measurements we present here were taken on the Mark III hypercube in the July of 1987. In each case TWOS was set to save the state of each process after every event, i.e. to take snapshots maximally often to ensure minimal rollback cost. Processes that had events rarely had there states saved correspondingly rarely. We do not know yet if this setting is optimal; it could well be that the cost of taking additional snapshots is not worth the savings in cost per rollback. Also, we set the interval between GVT calculations at 5 seconds, except that toward the end of a run it was reduced to 1 second. GVT calculation is a significant source of overhead, and we do not wish to do it any more often than is necessary to keep from running out of storage, but since termination can only be detected when GVT is updated, if we retained the 5 second interval to the end of a run our uncertainty in the time of termination would be as much as 5 seconds, which would bias our timings.

Since we do not have dynamic process migration in TWOS we have had to try many different assignments of processes to processors, and in each case we are data from those runs that ran fastest. In the few cases where we show more than one data point for a given number of processors, they are for different configurations of the same simulation.

The overhead per event message in TWOS is currently at least 3 milliseconds for messages sent within one processor, and 4.5 milliseconds when the messages are sent off-processor. These numbers were measured by running a trivial application that does nothing, and is basically "all overhead". The overhead per event includes (a) the copying of the event message from the sender's memory to TWOS, (b) packeting and depacketing, (c) creation of an antimessage copy retained by sender, (d) lazy-cancellation search to see if it is already present in the output queue, (e) lookup of the destination process in the routing table, (f) memory management, and queueing time on both ends, (g) transmission delay, (h) scheduling and interrupt handling at the receiver, (i) saving state between events, and (k) occasional calculation of GVT. Not included are costs for flow control, and rollback, since neither occurred in the trivial application used in these measurements.

In all cases the performance was measured without output. Including I/O made measurements unreliable because the low bandwidth communication out of the Hypercube could not keep up with the speed of the computation. However, all of software overhead to perform output except the physical transmission of the data is included, e.g. the routing of output requests to the *stdout* object on Node 0, the queueing of that output, and the commitment protocol.

We used two benchmark simulations in the initial evaluation of TWOS. One is a version of the Game of Life, designed to test Time Warp on a regularly structured model. The other is a fragment of a military command and control model that represents irregularly structured models.

8.1 The Life Benchmark

The Game of Life is a simple two-dimensional deterministic array automaton in which each cell has a 1-bit state whose value at time $t+1$ depends on its value and those of its 8 neighbors at time t. We programmed a toroidally-connected 256 x 256-cell version of the game, decomposed into processes in three different granularities:

(a) 1024 processes, each representing an 8 x 8 region;

(b) 256 processes, each representing a 16 x 16 region;

(c) and 64 processes, each representing a 32 x 32 region.

The reason for the different versions is to vary the ratio of computation to communication to test the effect of granularity on TWOS performance. The game was programmed in a 'dumb' way, so that each process recomputes the state of the cells in its jurisdiction at each time step, with no optimization. At each time step a

process receives a message from each of its neighbors, indicating their old states, and then sends a message to each of them with its new state. In most cases our measurements were made on a subcube of the hypercube so that the load in the simulation was balanced.

The Life Game, of course, has a tremendous amount of natural parallelism, and one can get good speedup from executing it concurrently in a synchronous manner without resorting to Time Warp. But Life is a good test for a distributed simulation mechanism for several reasons. First, it has an enormous amount of internal feedback, with every process being involved in many message communication cycles of every even length. Second, every object receives eight messages at every virtual time, so this is an opportunity to test the ability of TWOS to treat them as parts of a single message.

The results are summarized in Figure 6 where we plot time-to-completion of simulations up to time 10. In these cases the speedup is slightly sublinear. This seems reasonable considering that the special structure of the Life Game and its synchronous, time-stepped nature are not taken advantage of by Time Warp.

In this graph it is clear, as we would expect, that the fine-grained decompositions do not perform nearly as well as the more coarse-grained ones. In a simulation as regular as this the total TWOS time-overhead is proportional to the number of processes, and thus one would expect 16 times the overhead for the fine-grained decomposition as in the coarse-grained one. Furthermore, the fine-grained decomposition requires 16 times the memory overhead as the coarsest decomposition, because there are 16 times as many processes, and 16 times as many messages to be buffered. As a result we were unable to run the fine-grained decomposition (with 1024 processes) on fewer than four processors.

8.2 The COMMO* Benchmark

Our major benchmark, COMMO*, is designed to represent irregular, military-type simulations. It is derived from a piece of the FOURCE wargame built by the U.S. Army in White Sands, New Mexico. It was designed two years ago by a one of the authors (FPW), with little consideration of the behavior of Time Warp. We believe that TWOS can speed up models designed without knowledge its structure, and example corroborates this.

COMMO* consists of 130 processes representing division, brigade, and batallion staffs that send orders, intelligence reports, status reports, and other communications up and down the chain of command during a conventional battle. The various command staffs have 17 different message classes handled in different ways with different priorities and staff delays. Further complications arise because of competition for time on the various war communications media (radio links, telephone, courier, etc.) , and because messages are sometimes lost in staff processing. The model contains a mixture of high- and low-frequency feedback loops. It has a long ramp-up time before its behavior stabilizes, and another long ramp-down time as it heads toward termination. The ramp-up and ramp-down time is included in our timings even though there is less concurrency available during those parts of its execution.

Each execution involved 21,045 events. There were 88,241 event messages committed (i.e. not including those that were annihilated). Hence, the average event involved 4.2 event messages. There were also 14,110 queries (and the same number of replies). Thus there were always at least 116,461 messages transmitted (events, queries, and replies), not including additional messages that were annihilated by antimessages.

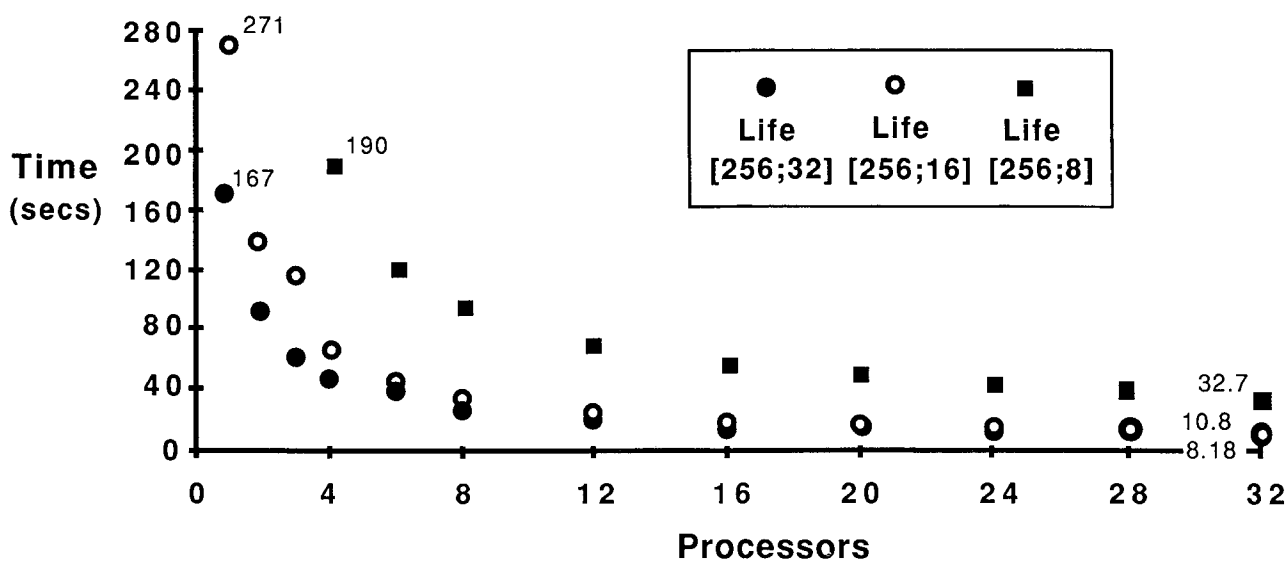

Figure 6: Time to completion of a 256 x 256 cell Life game, divided into processes of size 32 x 32, 16 x 16, and 8 x 8 cells respectively.

During the intermediate stable period of COMMO*'s execution events happen at each integral simulation time, and at each such epoch there are approximately 75-85 processes with events scheduled (mostly, but not exactly, the same processes each time). These events vary over an order of magnitude in the real time it takes to simulate them, so the model is not very well balanced. We regard this as typical of the kinds of simulations people will actually write. The 15 most computationally intensive processes (of 130) account for about 3.0% each of all cycles when COMMO* is executed sequentially. Since that 3% must be executed sequentially under TWOS as well, we know *a priori* that there can be no more than a factor of 33 speedup possible in this application, even if all processes were independent of one another. Since they are not at all independent there is surely considerably less than 33-fold concurrency available, but it is difficult to estimate how much less. The important thing is that COMMO* is exceedingly irregular in its behavior. It is intended to be as realistic and complex as possible for its size. Further details are available on request from the authors.

The graph in Figure 7 shows the time-to-completion of COMMO* under TWOS as a function of the number of processors. The minimum time of 166 seconds was with 24 nodes. After 16 processors with a time of 201 seconds (when the 15 cycle-hogs could all be by themselves on different processors) there is little additional improvement.

In Figure 8 the same timing data is plotted as speedup. There are two curves, one calculated using as a basis the time to execute COMMO* under TWOS on one node, and the other, proportional to the first, using as a basis the time it took to execute COMMO* using a sequential event-list simulator on one node of the Hypercube. As the graph shows, we obtained a maximum speedup (relative to TWOS on one node) of 10.66 using 24 processors. At 16 nodes the speedup was already 8.62. For regularly-shaped computations one can usually sustain linear speedups until some critical point where the performance abruptly flattens out. For irregular computations one expects a smoother decline in efficiency, which is exactly what we observe here. Notice that the speedup in Figure 8 is nearly linear for small numbers of processor, but that diminishing returns sets in after about seven nodes. This is to be expected with a small simulation that has only a modest amount of concurrency available; for larger models the near-linearity should be sustainable much longer. After 24 nodes the speedup declines slightly and rather erratically. We do not know as of this writing whether that is because of characteristics in the model, or (more likely) because we have not yet found the best assignments of processes to processors for the largest numbers of nodes.

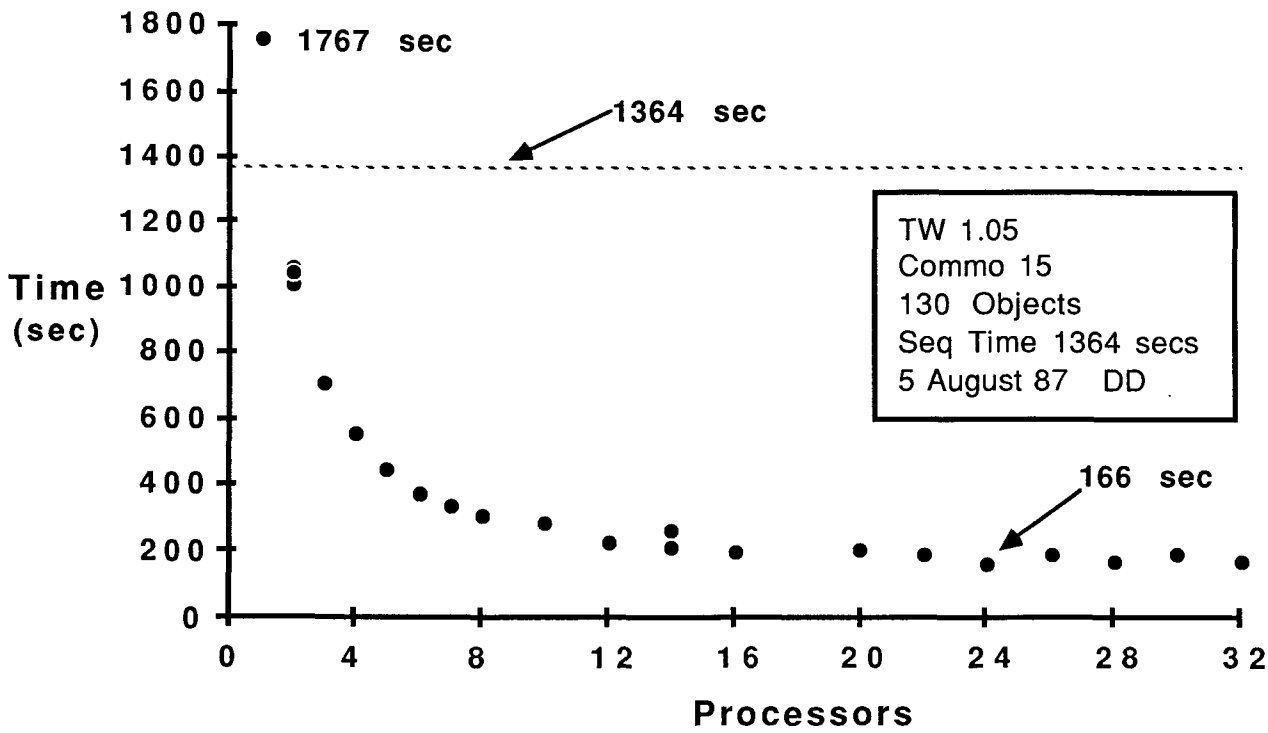

Figure 7: Time to completion of the irregular model COMMO*

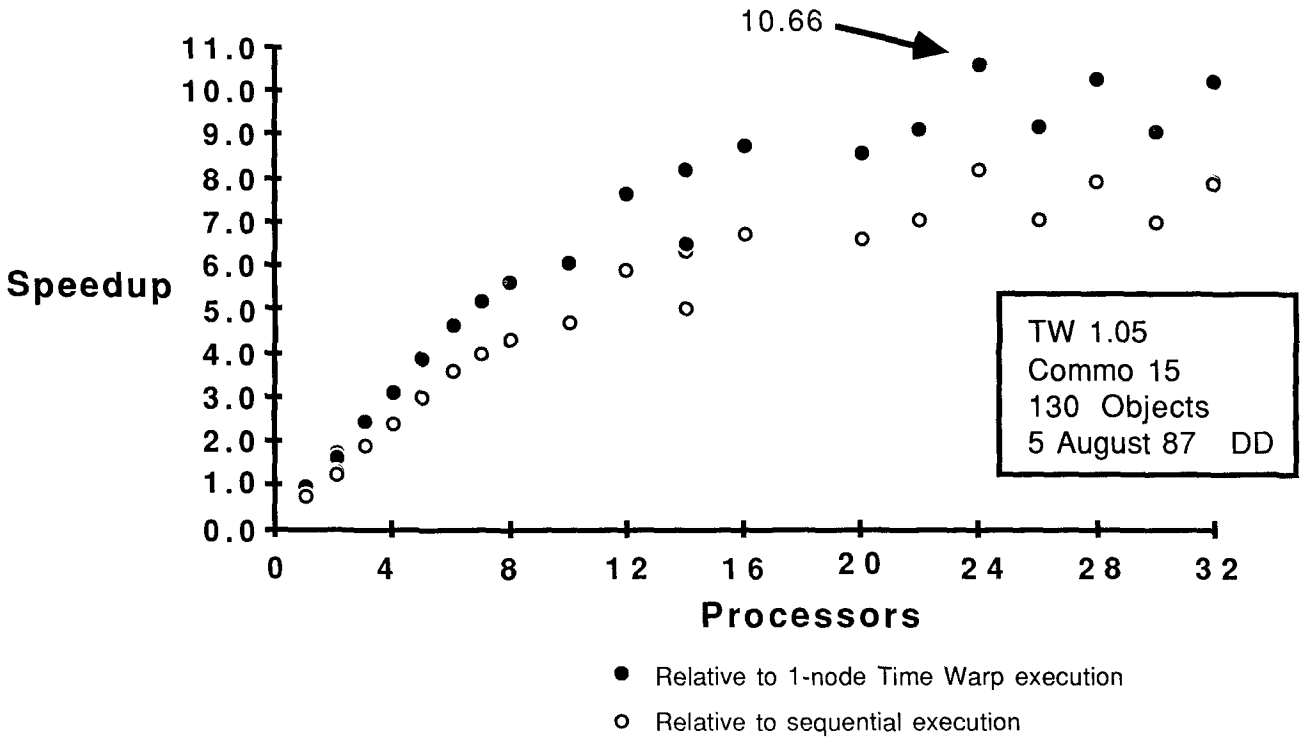

Figure 8: Speedup of COMMO*

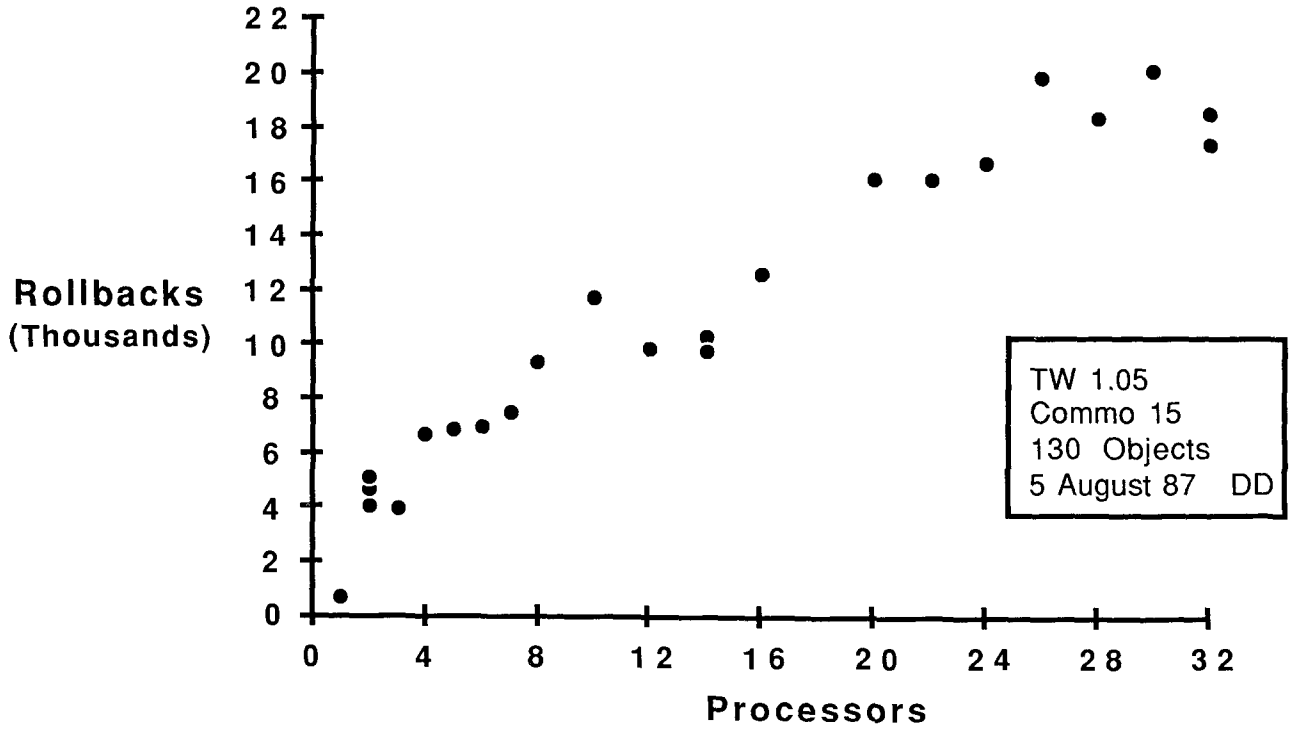

Figure 9: Number of rollbacks in runs of COMMO*

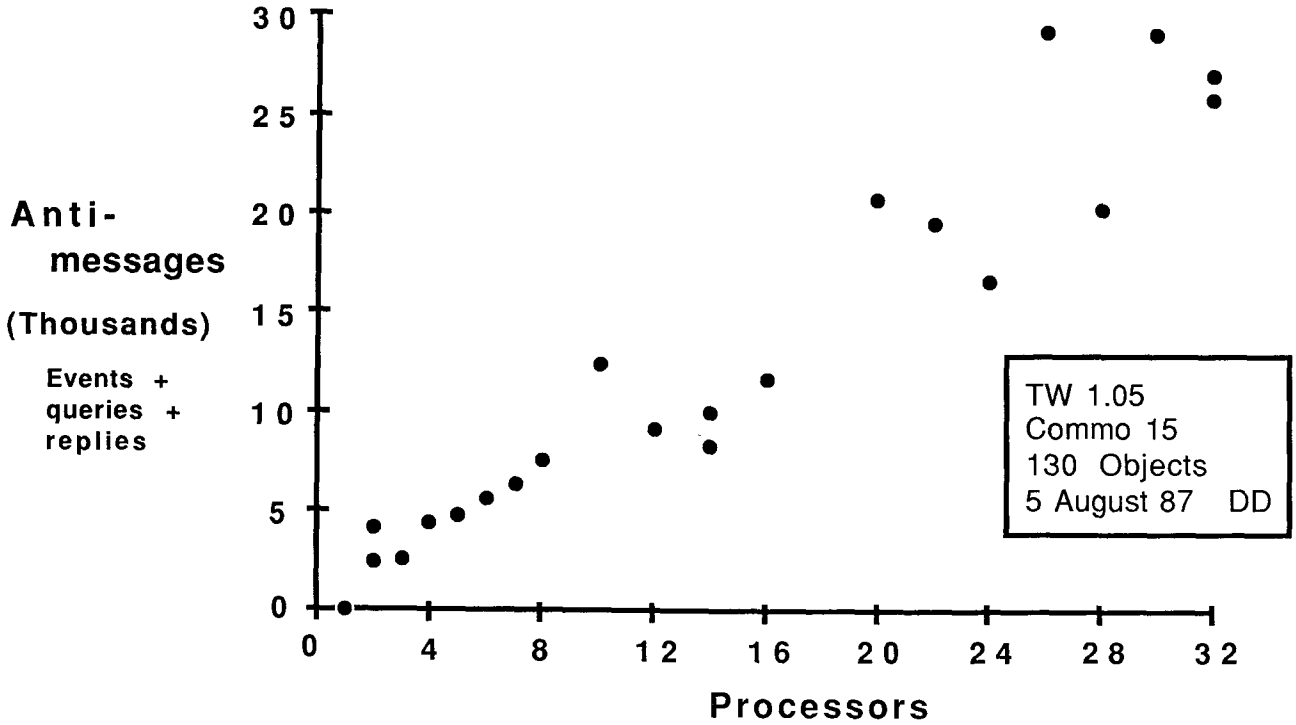

Figure 10: Number of antimessages sent during runs of COMMO*

The total number of rollbacks experienced during execution is shown in Figure 9. We count only those rollbacks that cause recomputation of events or queries; we do not count "technical rollbacks" that involve setting back a virtual clock without any recomputation, e.g. a rollback from ∞ to a finite time. The latter numbered from about 39,000 to 51,000. As can be readily seen, the number of rollbacks generally increases with the number of processors used. Combining this with the results of Figure 8 indicates that achieving more speedup requires *more* rollbacks, contrary to what one might expect at first thought. This is consistent with the theoretical observation that rollbacks generally do not occur in those portions of the execution that are the current bottleneck, i.e. are farthest behind in simulation time. The single fastest run (24 processors) did not have an unusually low number of rollbacks.

One of the early performance questions about Time Warp was the amount of overhead caused by negative messages. Figure 10 shows that across all of the runs of COMMO* the maximum number of antimessages transmitted was slightly more than 29,272 (in the run with 26 processors). Each annihilated a positive message. Since 116,461 messages of all kinds were transmitted but were *not* annihilated, the total message traffic in that run was 175,005 messages. Thus, a maximum of 58,544 out of the 175,005 messages were synchronization overhead, or about 33.4% of the total.

9. Conclusions:

The Time Warp Operating System now runs reliably on the JPL Mark III Hypercube, and is capable of extracting at least an order of magnitude of speedup in at least one relatively small and irregular simulation. We have every reason to believe that much more speedup is available in larger models, and that we will be able to demonstrate that when we have access to more than 32 processors.

Much more empirical work, particularly with additional and larger benchmarks being built now, is necessary before we will fully understand the dynamics of the Time Warp mechanism. Among the important questions not yet addressed are:

(a) How does Time Warp's performance degrade as memory gets tight?

(b) How much additional performance gain is possible from dynamic load management?

(c) How should the key tuning parameters (frequency of state saving, frequency of GVT calculation, etc.) be set?

(d) Where are there opportunities for hardware support to reduce overhead and allow for reduced granularity?

(e) How should Time Warp be reimplemented to take full advantage of shared memory architectures?

(f) What tools and environments should be built to support distributed simulation?

These are questions we will be investigating in the next years.

Acknowledgments

This work was funded by the U.S. Army Model Improvement Program (AMIP) Management Office (AMMO), NASA contract NAS7-918, Task Order RE-182, Amendment No. 239, ATZL-CAN-DO.

The authors would like to thank Jack Fanselow and Dave Curkendall of JPL and Geoffrey Fox of Caltech for their longstanding cooperation with this project, and for lending Mark III time for making the measurements reported here. We thank Col. Kenneth Wiersema and the Army Model Improvement Program for consistent sponsorship over three years. We also wish to acknowledge the contributions of Orna Berry, Anat Gafni, and Andrej Witkowski to the theory of Time Warp, and of Henry Sowizral as the co-inventor of the Time Warp mechanism.

References

[Berry 86] Berry, Orna, "Performance Evaluation of the Time Warp Distributed Simulation Mechanism", Ph.D. Dissertation, Dept. of Computer Science, University of Southern California, May 1986

[Chandy 81] Chandy, K.M., and Misra, Jayadev, "Asynchronous distributed simulation via a sequence of parallel computations", *Communications of the ACM*, Vol. 24, No. 4, April 1981

[Fox, 85] Fox, Geoffrey, "Use of the Caltech Hypercube", *IEEE Software*, Vol. 2, p. 73, July 1985

[Gafni 85] Gafni, Anat, "Space Management and Cancellation Mechanisms for Time Warp", Ph.D. Dissertation, Dept. of Computer Science, University of Southern California, TR-85-341, December 1985

[Jefferson 85] Jefferson, David, "Virtual Time", ACM *Transactions on Programming Languages and Systems*, Vol. 7, No. 3, July 1985

[Jefferson 82] Jefferson, David and Sowizral, Henry, "Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control", Rand Note N-1906AF, the Rand Corporation, Santa Monica, California, Dec. 1982

[Joyce 87] Joyce, J., Lomow, G.A., Slind, K., Unger, B.W., "Monitoring Distributed Systems", *ACM Transactions on Computer Systems*, Vol. 5, No. 2, May 1987

[Lamport 78] Lamport, Leslie, "Time, clocks and the ordering of events in a distributed system", *Communications of the ACM*, Vol. 21, No. 7, July 1978

[Li 87] Li, X., Unger, B. W., "Languages for Distributed Simulation", *Proceedings of the Conference on Simulation and AI, Simulation Series*, Vol 18, No. 3, January 1987

[Misra 86] Misra, Jayadev, "Distributed Discrete Event Simulation", *Computing Surveys*, Vol 18, No. 1, March 1986

[Peterson 85] Peterson, J.C., J. Tuazon, D. Lieberman, M. Pinel, "Caltech/JPL Hypercube Concurrent Processor", *Proceedings of 1985 International Conference on Parallel Processing*, St. Charles, Ill., Aug. 1985

[Reynolds 82] Reynolds, Paul, "A Shared Resource Algorithm for Distributed Simulation", *Proceedings of the 9th International Symposium on Computer Architecture*, Austin, Texas, IEEE, New York

[West 87] West, D., Lomow, G., Unger, B.W., "Optimizing Time Warp Using the Semantics of Abstract Data Types", *Proceedings of the Conference on Simulation and AI ,Simulation Series*, Vol 18, No. 3, January 1987

[Xiao 86] Xiao, Z., Unger, B.W., Cleary, J., Lomow, G., Li, X., Slind, K., "Jade Virtual Time Implementation Manual", Research Report No. 86/242/16, Dept. of Computer Science, University of Calgary, Calgary, Alberta