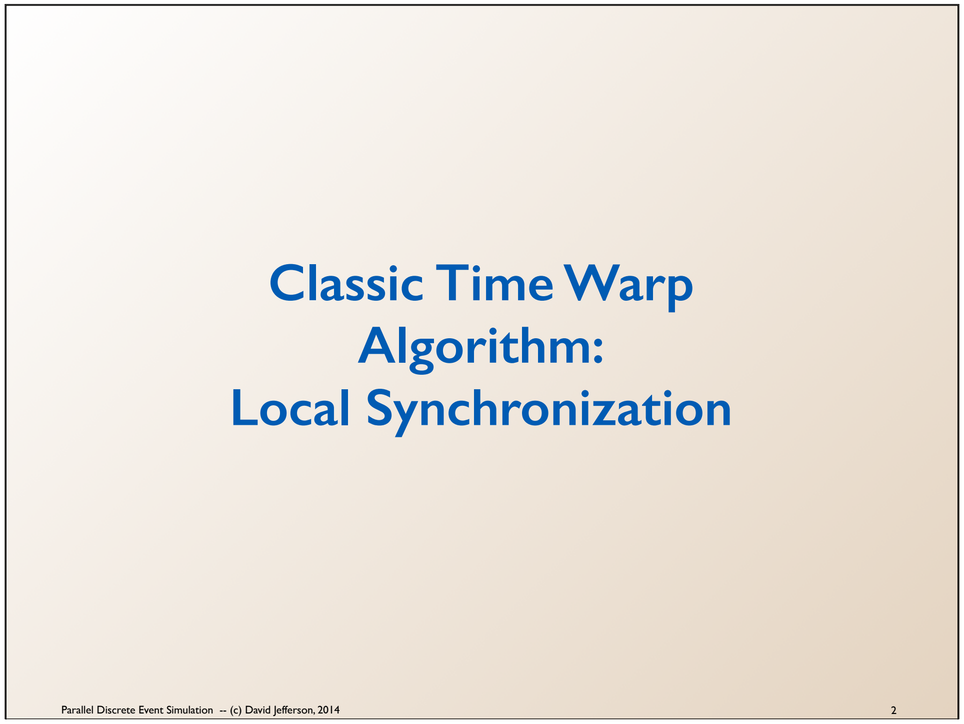
Parallel Discrete Event Simulation Course #8

David Jefferson
Lawrence Livermore National Laboratory
2014

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

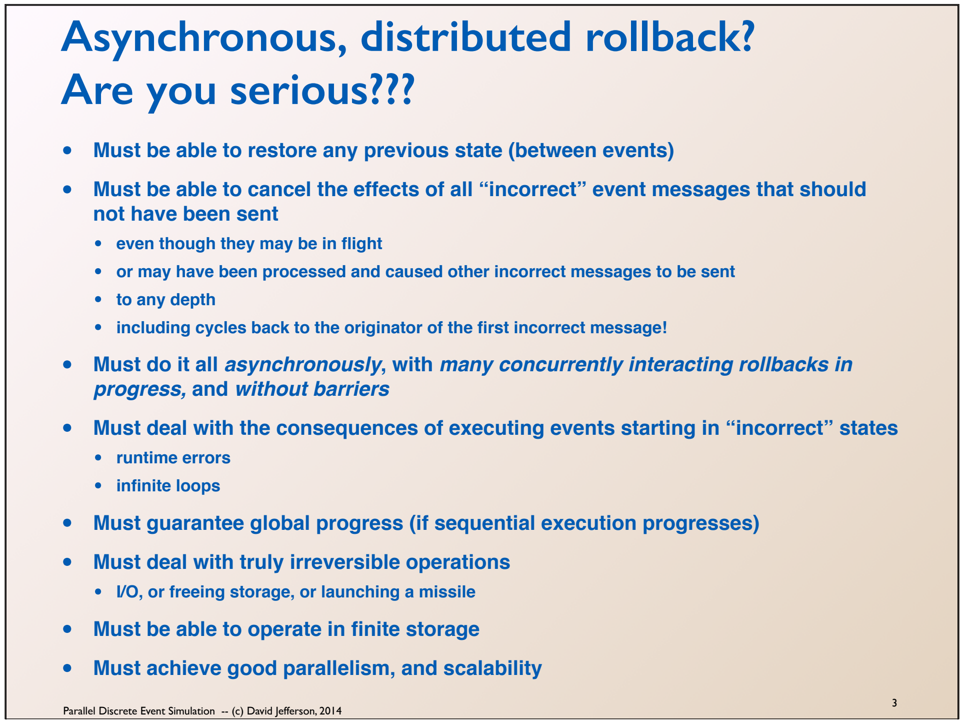
Release Number: LLNL-PRES-652291



Classic Time Warp Algorithm: Local Synchronization

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 2

We call this the “classic” Time Warp algorithm because it most closely resembles the very first implementation. Current implementations differ in many ways, though not fundamentally. This presentation will also emphasize clarity and abstraction and symmetry in the algorithm, so that it is easiest to explain, even if actual implementations differ in many details.



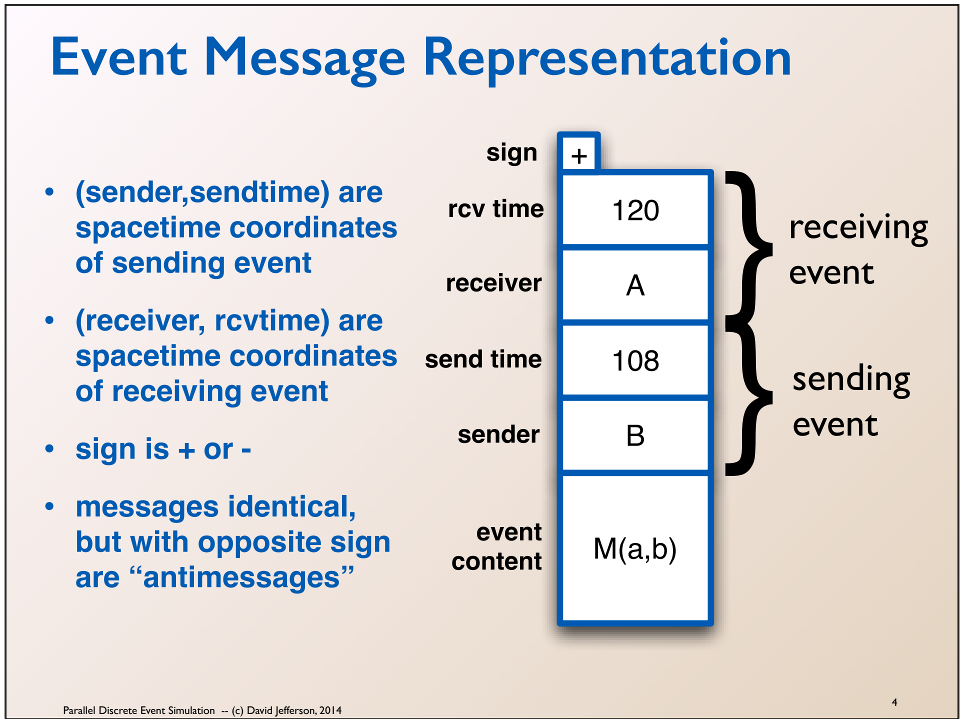
Asynchronous, distributed rollback? Are you serious???

- Must be able to restore any previous state (between events)
- Must be able to cancel the effects of all “incorrect” event messages that should not have been sent
  - even though they may be in flight
  - or may have been processed and caused other incorrect messages to be sent
  - to any depth
  - including cycles back to the originator of the first incorrect message!
- Must do it all *asynchronously*, with *many concurrently interacting rollbacks in progress,* and *without barriers*
- Must deal with the consequences of executing events starting in “incorrect” states
  - runtime errors
  - infinite loops
- Must guarantee global progress (if sequential execution progresses)
- Must deal with truly irreversible operations
  - I/O, or freeing storage, or launching a missile
- Must be able to operate in finite storage
- Must achieve good parallelism, and scalability

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

3

This shows the list of challenges we have to overcome for the Time Warp algorithm, or any optimistic PDES algorithm, to be practical. Most people with a background in asynchronous distributed computation who have not seen optimistic PDES algorithms are inclined to believe that doing this is either literally impossible, or at least hopelessly complex and slow.



Event Message Representation

- (sender,sendtime) are spacetime coordinates of sending event
- (receiver, rcvtime) are spacetime coordinates of receiving event
- sign is + or -
- messages identical, but with opposite sign are "antimessages"

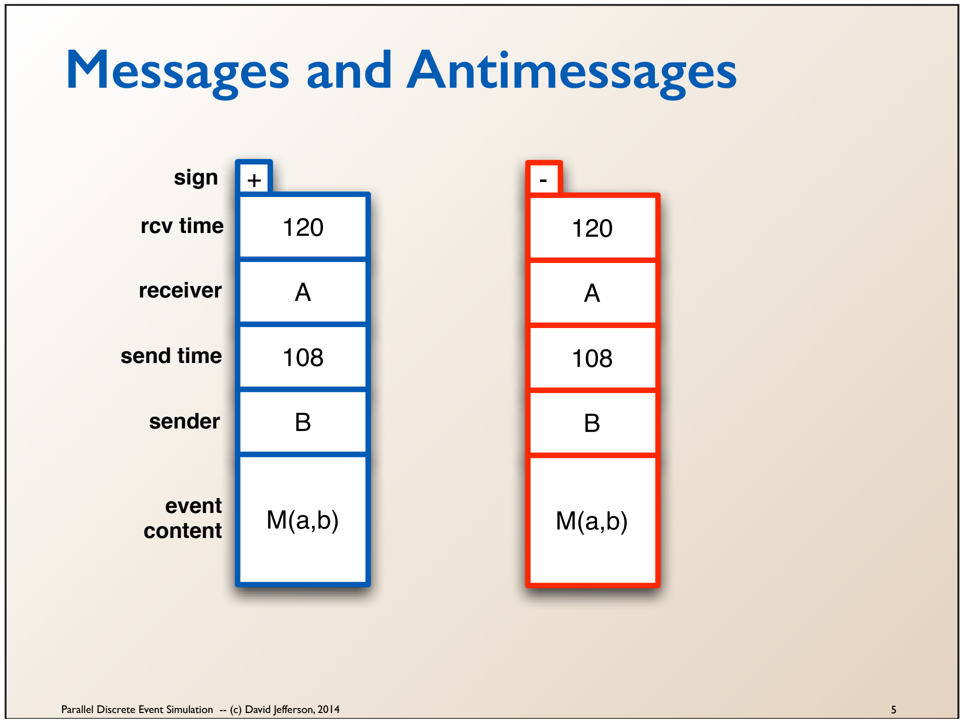
sign
+
rcv time
120
receiver
A
send time
108
sender
B
event content
M(a,b)
receiving event
sending event

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

4

An event message in the TW algorithms holds the spacetime coordinates of the sending event (B,108) and the spacetime coordinates of the receiving event (A,120). It also has as sign, + or -, whose purpose will become apparent.

For reasons of symmetry, we consider saved states (forward reference) to also have a "send time" and a "receive time". The "send time" is the time of the event that produced the state, and the "receive time" is the time of the event that consumes the state, i.e. the time of the next event.



Messages and Antimessages

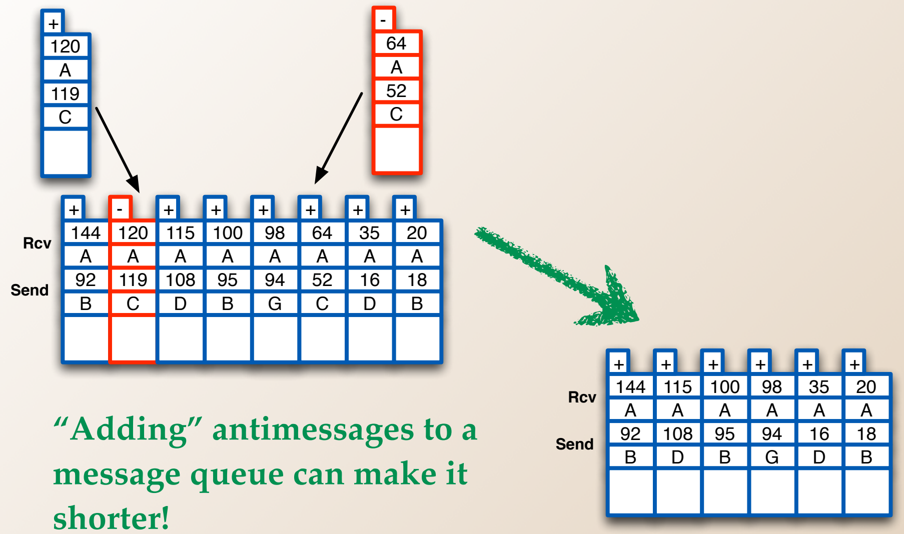
sign
rcv time
receiver
send time
sender
event content
+
120
A
108
B
M(a,b)
-
120
A
108
B
M(a,b)

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 5

Two event messages that are identical in all respects except for their signs are “antimessages” of one another.

As we will see, this terminology is apt because if antimessages come into “contact” with one another (by being enqueued in the same queue) they mutually annihilate. More generally, as we will see later, messages are always created in antimessage pairs and destroyed in antimessage pairs. Hence, “charge” is conserved, and the sum of all charge in the simulation is always zero.

Message-AntiMessage Queueing Discipline



“Adding” antimessages to a message queue can make it shorter!

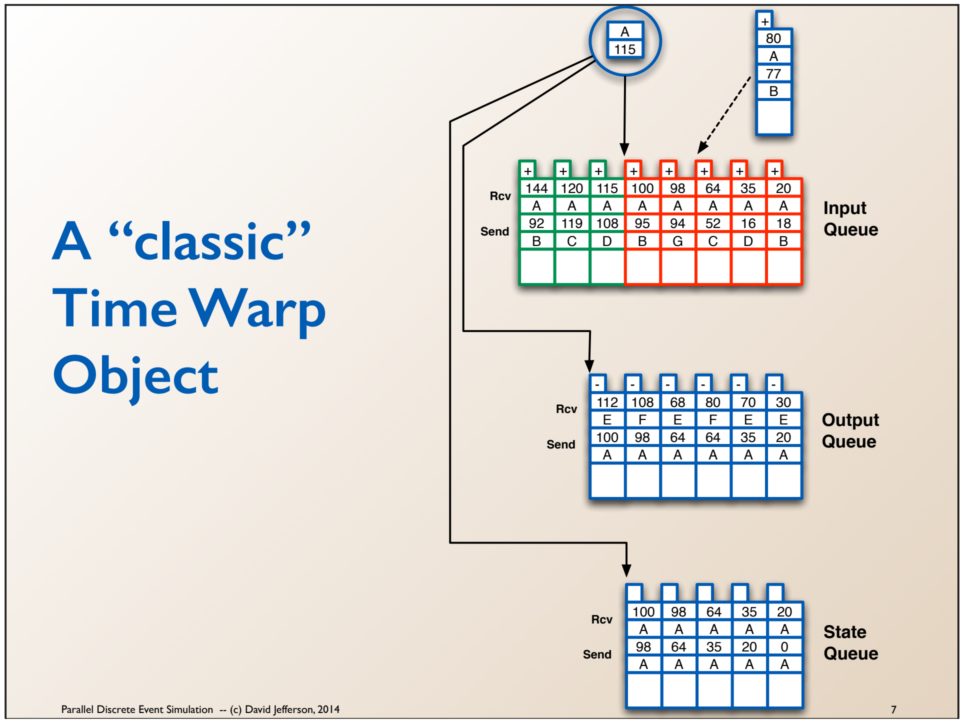
This is a “before” and “after” diagram. The message queue before (which happens to be an input queue since it is sorted by receive time) has 8 messages in it. (One of those messages is negative--that can happen, although it is infrequent, since we do not assume message order preservation. Although the positive message is always sent before the corresponding negative message, the negative message may have arrived and been enqueued at the receiver first.).

Two new messages arrive for enqueueing that happen to be antimessages to two other already in the queue. When they are “added” to the queue, the result is two annihilations and the queue gets shorter.

As far as I know this is the only “collection” data type appearing in CS literature which admits of “negative” objects like this, so that “adding” to a collection results in the collection getting smaller. This is not the same as just adding a `queue.delete(element)` method to the data type, because such a method has no effect if the element is not present in the collection, whereas adding an anti-element does have an effect, which is manifest either immediately to annihilate with its counterpart, or later if and when its counterpart is enqueued

Note also, that it is not forbidden to have two or more identical copies of the same event message in a queue--that constitutes a tie and calls for invocation of a tie breaking rule, but is otherwise OK. If, later and antimessage of one of them arrives, it annihilates with only one of the messages, leaving the others present. Message-antimessage annihilation “conserves charge”.

The Time Warp algorithm with its message-antimessage terminology could, of course, be re-described without the colorful elementary particle analogy. However, I think it helps to reveal the symmetries of the algorithm--there will be many more to come. So if you begin to think that the analogy is a little strained, please hold off until you see the way it develops later, particularly when it comes to the flow control and storage management parts of the TW algorithm.



A “classic” Time Warp Object

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 7

Event 115 is executing at the moment

Each incoming and outgoing event message, and each state, is labeled with the spacetime coordinates of its “sending” and “receiving” events.

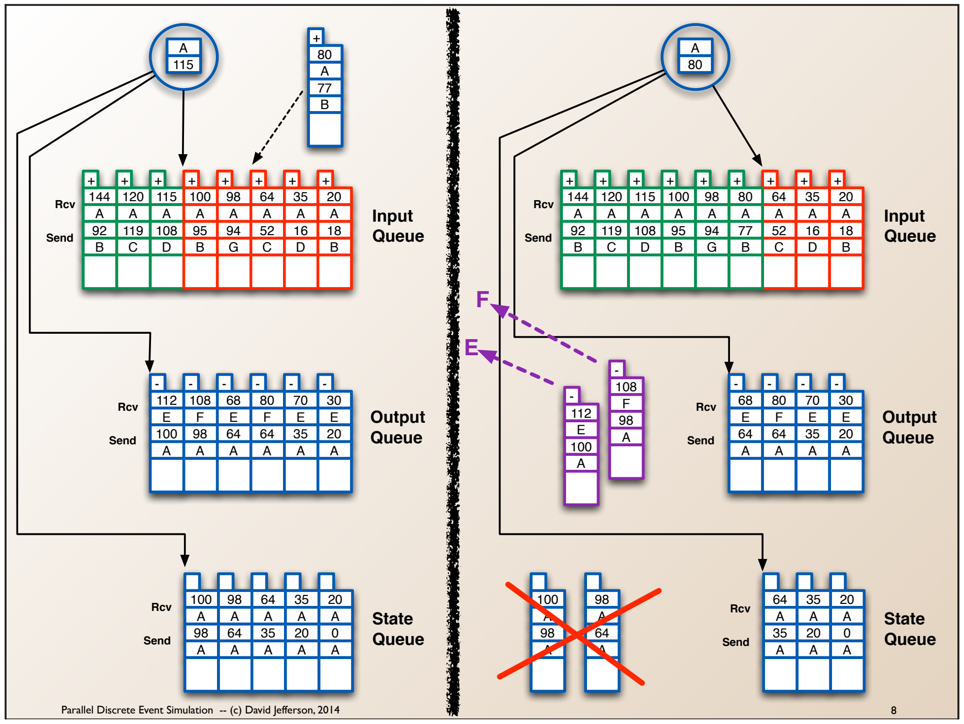
Input queue contains incoming event messages and is sorted by Receive Time. (Send Times are not necessarily in sorted order.) Input messages are usually all positive (but not always). The input queue contains both past (processed) and future (unprocessed) messages

Output queue contains outgoing event messages (negative copies) and is sorted by Send Time. Output messages are generally all negative (but not quite always—more later about that). The output queue often contains only past messages (but in later variations we will correct that to allow “future” output messages in certain interesting TW variations, e.g. “lazy cancellation”).

State Queue contains snapshots of states take between events, and is sorted by both send time and receive time-- the sorting is identical either way, so there is no distinction. The snapshots are only of static memory, and sometimes heap memory -- not stack memory, as the stack is always empty between events

Saved states are all “neutral”. Actually, this is something of a stretched convention, but it makes some sense, since a state is both an output from one event and an input to the next. The state queue contains only past states (but in later variations we will consider variations of TW in which “future” states make sense).

The reason I mention that in some variations of TW there are such things as “future output messages” (sent in the future) and “future states” (generated in the future), is to emphasize the symmetry of the TW object representation and algorithm among the input queue, output queue, and state queue of a TW object. The future states or future output messages are less frequently useful than the obvious desirability of future input messages--but this is just a performance issue. Logically, all three queue are symmetric, but the symmetry is broken by performance considerations.


Rcv
Send
Input Queue
Output Queue
State Queue
F
E
Parallel Discrete Event Simulation -- (c) David Jefferson, 2014
8

This diagram represents the before and after of the arrival of a straggler message.

Object A is at simTime 115 when an event message arrives in the past with a timestamp of 80. Two events at time 98 and 100 should not have been executed. The event at time 115 has been partially executed--we are still in the middle of executing it--and it may have sent some but not all of the event messages it would have. In any case, whatever it did is likely to be wrong.

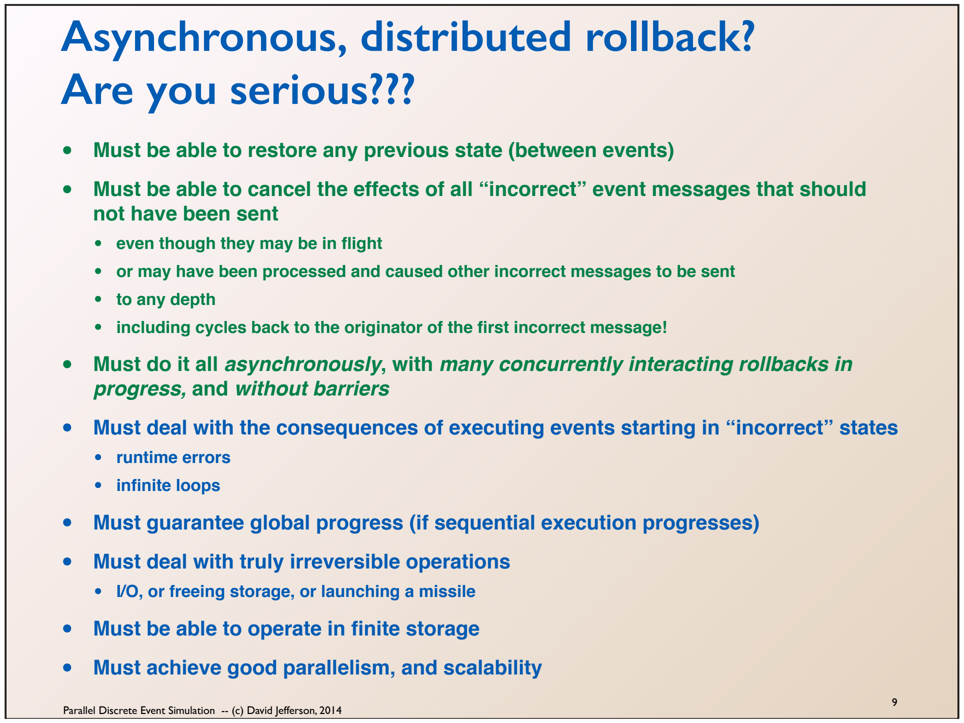
We have to roll back to the state at time 80, i.e. to the state saved after execution of the event at time 64, the last correctly-executed event before time 80. The rollback consists of the following actions. Strictly speaking the kind of rollback we are describing here is called “aggressive cancellation”, and is in some ways the most “optimistic” of the optimistic algorithms. An alternative is called “lazy cancellation”, and another is called ??

The rollback consists of the following steps.

1) Insert the straggler message into the input queue where it belongs in the sort. It may be an antimessage and annihilate with another message already in the input queue. That make no difference in the algorithm at all--antimessages are treated identically.

2) Interrupt the event in progress (115). Restore the state saved after event 64 as the current state of the object. Delete the two subsequently saved states created by events 98 and 100, since they are (probably) incorrect (and one of them has been partially modified by event 115). Note that in a rollback variation called “lazy re-evaluation” these states would not actually be deleted at this time--and hence it really is possible to have “future” states in the queue. And in another variation called “sparse state saving” where we don’t save states between every two event, but do it less often than that, then we might have to roll back farther than time 80 restore to an even earlier state.)

3) From the output queue, find the antimessages to the messages sent incorrectly after time 80, dequeue them, and deliver them to their receivers--the same objects that the original (incorrect) positive messages went. Note that this includes any messages sent by the event that was in progress (and was interrupted), in this case event 115. Surprisingly (!) that is all that is required to exactly undo the effects of those positive messages, whether they have been delivered yet or not, or have been processed or not, or have caused generation of a tree of further, probably incorrect, distributed computation. The fact that this antimessage mechanism works in all cases, and allows the simulation globally to make progress asynchronously, independent of the speed of execution of the objects or the latency of message delivery, and regardless of the possibility of many interacting rollbacks in progress simultaneously, is a key observation at the foundation of most optimistic methods. (However, some less aggressive variations, e.g. *risk free* algorithms (in Paul Reynolds’ taxonomy) do not transmit event messages until they can be committed, and thus have no need for antimessages. This comment is a forward reference, and I don’t know whether I will get back to it in the course.)



Asynchronous, distributed rollback? Are you serious???

- Must be able to restore any previous state (between events)
- Must be able to cancel the effects of all “incorrect” event messages that should not have been sent
  - even though they may be in flight
  - or may have been processed and caused other incorrect messages to be sent
  - to any depth
  - including cycles back to the originator of the first incorrect message!
- Must do it all *asynchronously*, with *many concurrently interacting rollbacks in progress,* and *without barriers*
- Must deal with the consequences of executing events starting in “incorrect” states
  - runtime errors
  - infinite loops
- Must guarantee global progress (if sequential execution progresses)
- Must deal with truly irreversible operations
  - I/O, or freeing storage, or launching a missile
- Must be able to operate in finite storage
- Must achieve good parallelism, and scalability

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

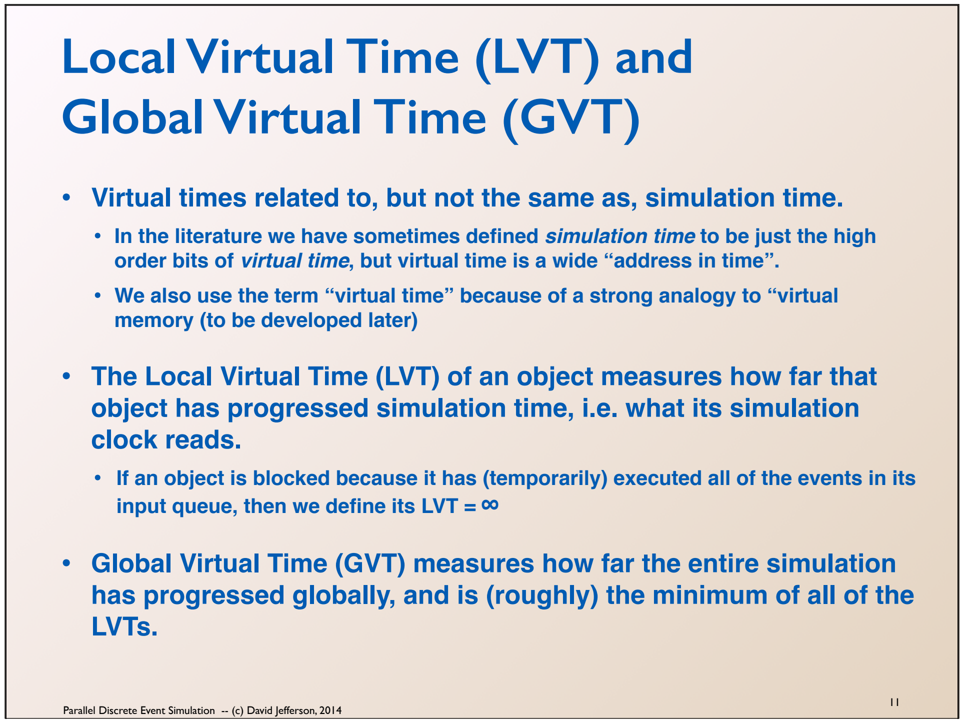
9

This shows the list of challenges we have to overcome for the Time Warp algorithm, or any optimistic PDES algorithm, to be practical. Most people with a background in asynchronous distributed computation who have not seen optimistic PDES algorithms are inclined to believe that doing this is either literally impossible, or at least hopelessly complex and slow.

Global Virtual Time (GVT) and Commitment: Global Synchronization

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

10

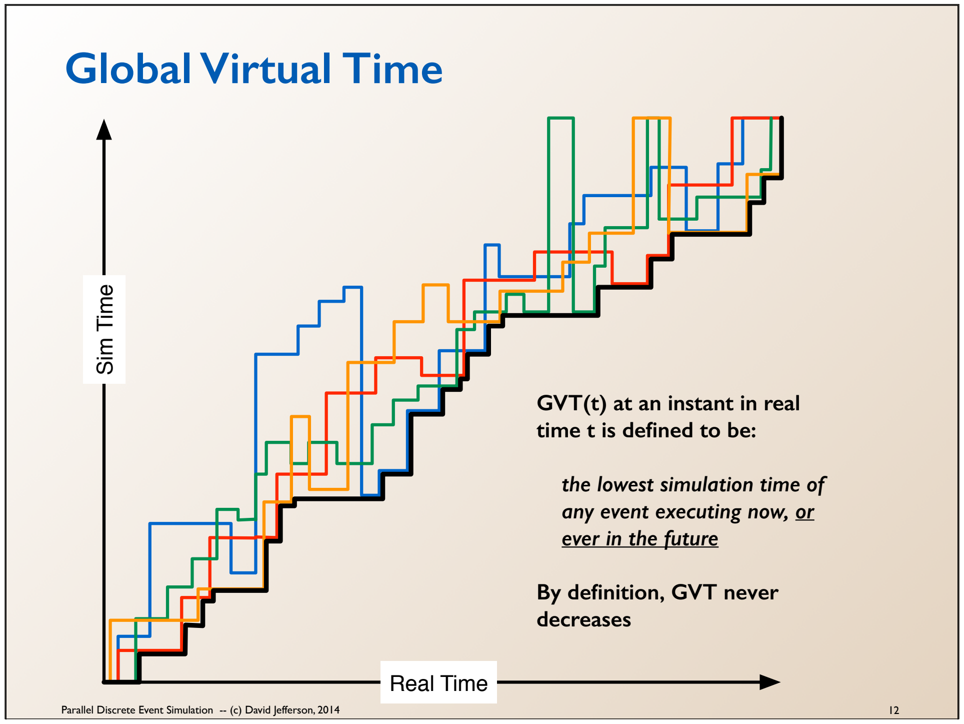


Local Virtual Time (LVT) and Global Virtual Time (GVT)

- **Virtual times related to, but not the same as, simulation time.**
  - In the literature we have sometimes defined ***simulation time*** to be just the high order bits of ***virtual time***, but virtual time is a wide "address in time".
  - We also use the term "virtual time" because of a strong analogy to "virtual memory (to be developed later)
- **The Local Virtual Time (LVT) of an object measures how far that object has progressed simulation time, i.e. what its simulation clock reads.**
  - If an object is blocked because it has (temporarily) executed all of the events in its input queue, then we define its LVT = ∞
- **Global Virtual Time (GVT) measures how far the entire simulation has progressed globally, and is (roughly) the minimum of all of the LVTs.**

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 11

LVTs go forward and backward in time, but more often forward. GVT *never* goes backward, and is *always* (at any instant of wall clock time) a lower bound for all LVTs at that instant.



Global Virtual Time

Sim Time
Real Time

GVT(t) at an instant in real time t is defined to be:

the lowest simulation time of any event executing now, __or ever in the future__

By definition, GVT never decreases

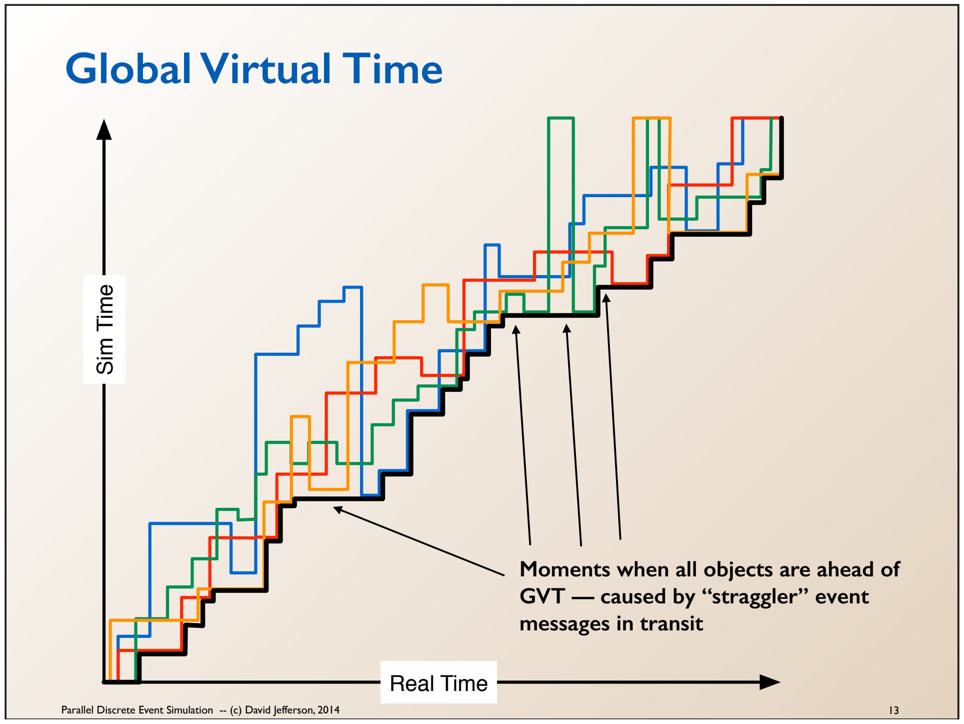
Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

12

The *local virtual time* (LVT) of an object at an instant is the simulation time to which an object has progressed, i.e. what its simulation clock reads.

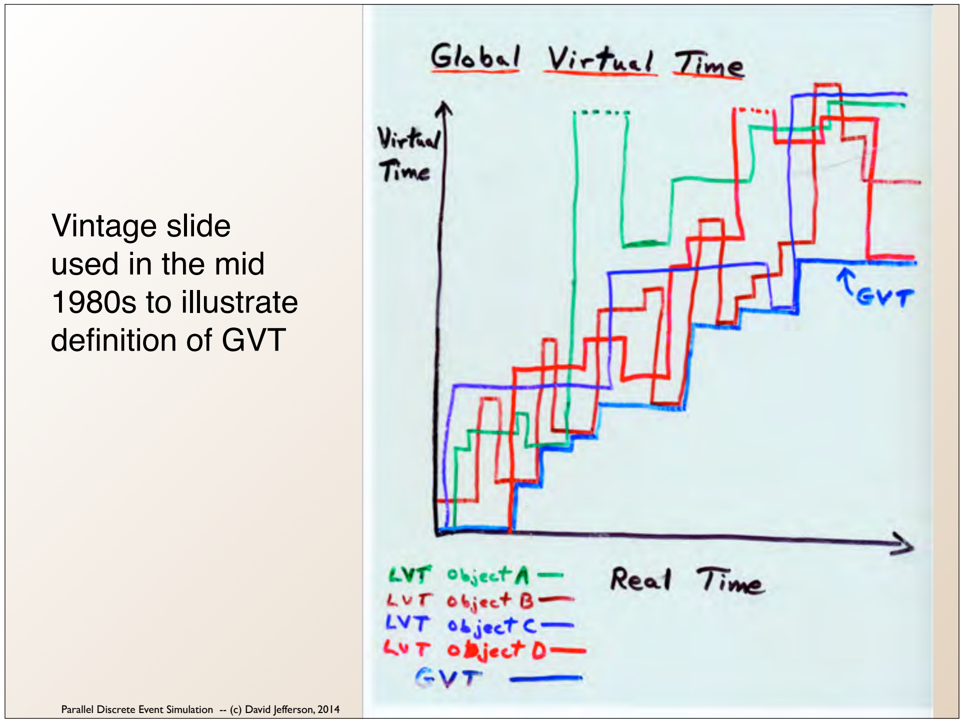
The colored lines each represent the LVT of one object in the simulation that goes forward and rolls back as the simulation executes.

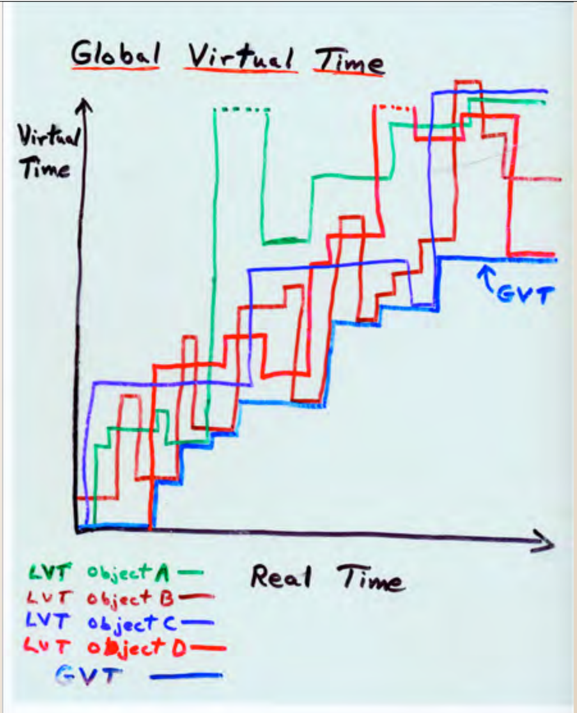
The black line, acting as a tight monotonic lower bound envelope for all of the LVT curves, is Global Virtual Time, GVT.



Global Virtual Time

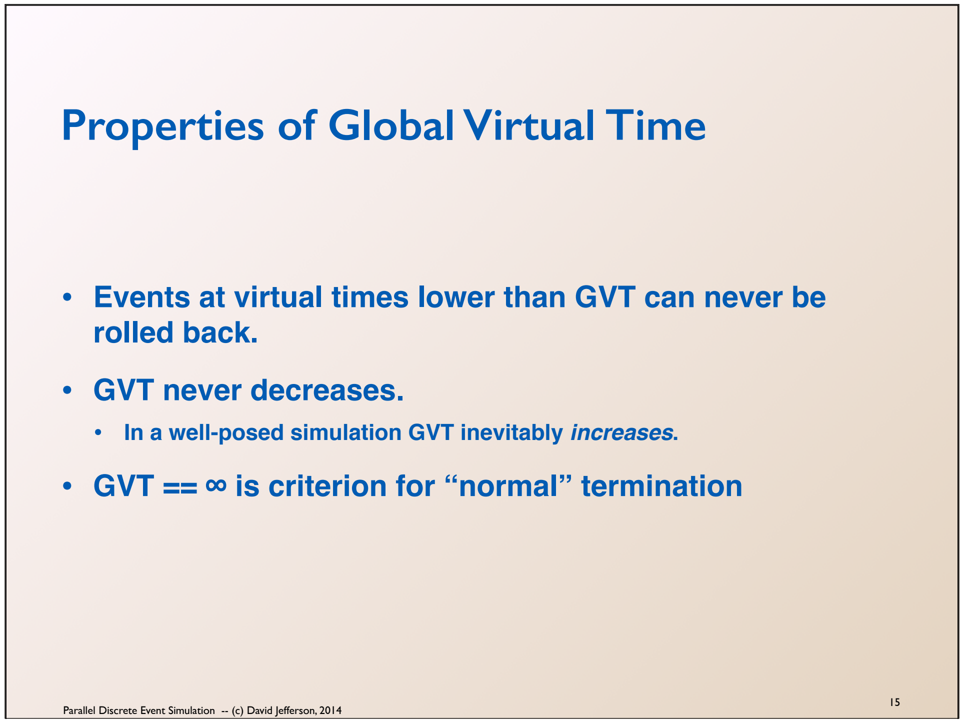
There are a few places in this diagram where GVT is strictly lower than the minimum of all LVTs. That will happen at times when a message is in transit that happens to carry a lower timestamp (receive time) than the LVT of any object. When it is delivered, it will cause the receiving object to rollback to the message's received time.


Vintage slide used in the mid 1980s to illustrate definition of GVT



Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

Vintage slide for definition of virtual time (1984 or so). For those of you who don't recognize this ancient technology, this was a hand-drawn diagram with colored felt-tip pens on plastic transparencies. I still have the original 30 years later.



Properties of Global Virtual Time

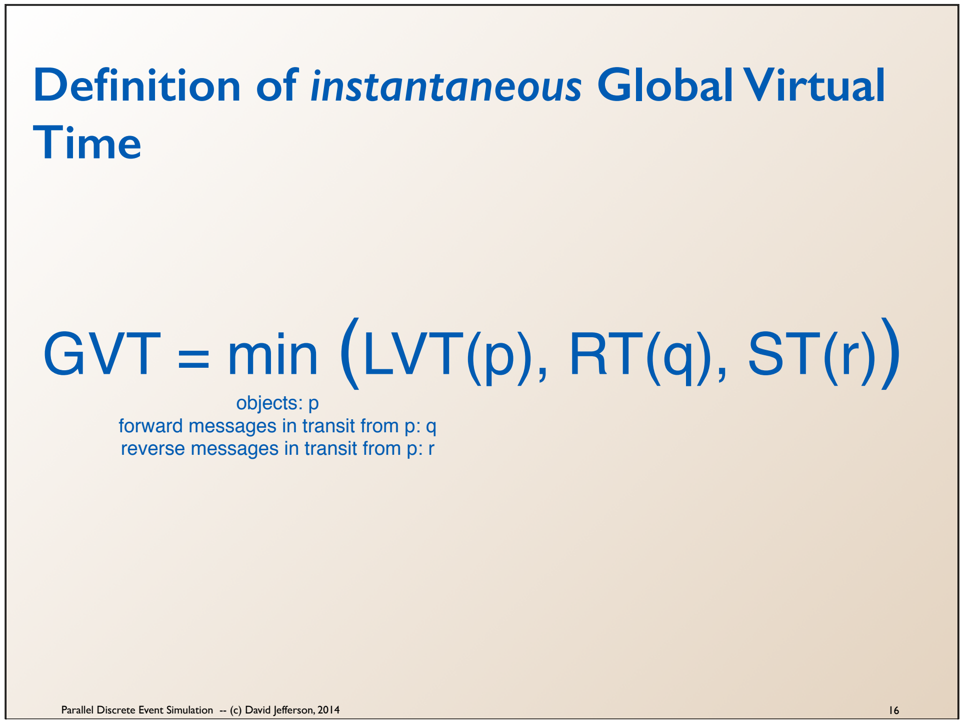
- **Events at virtual times lower than GVT can never be rolled back.**
- **GVT never decreases.**
  - **In a well-posed simulation GVT inevitably *increases*.**
- **GVT == ∞ is criterion for "normal" termination**

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

15

By definition, GVT never decreases, and in fact it must increase unless the simulation has a bug in it like an infinite loop that would also affect the sequential execution of the same model.

GVT == ∞ if and only if all objects are at time infinity and no messages are in transit. In that case all objects have run out of events to process, and the entire global simulation terminates normally. Termination detection is the same as detecting the GVY == ∞.



Definition of *instantaneous* Global Virtual Time

$$GVT = \min \Big(LVT(p),\ RT(q),\ ST(r) \Big)$$

objects: p
forward messages in transit from p: q
reverse messages in transit from p: r

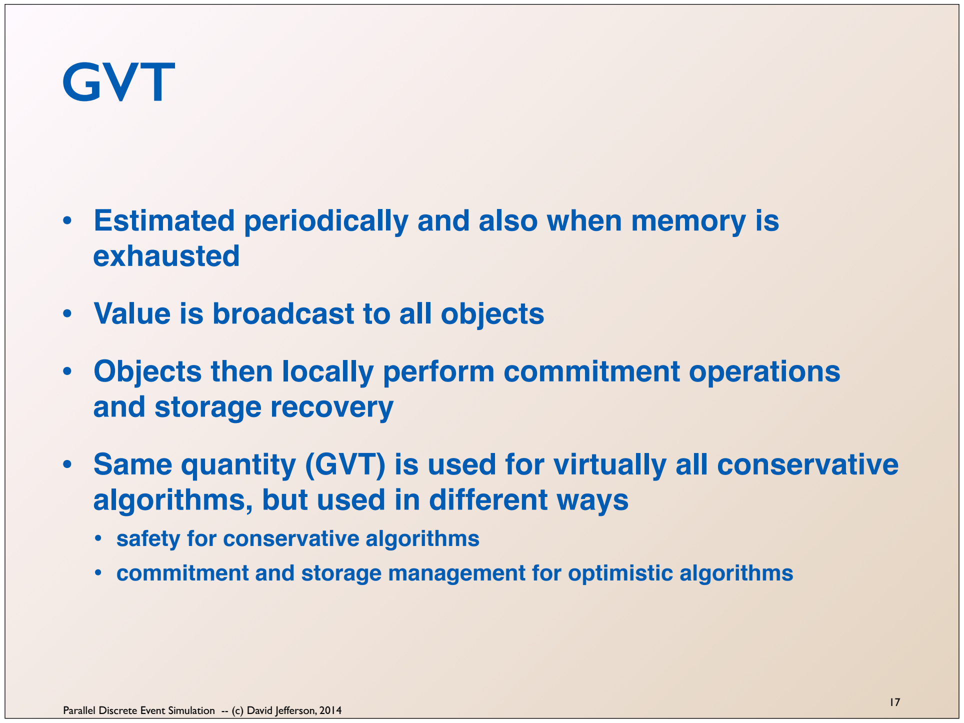
Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 16

LVT(p) = Local Virtual Time, i.e. the simulation clock value of Object p
RT(q) = Receive Time of the (positive or negative) event message q that is in transit
ST(r) = Send Time of (positive or negative) event message r that is in transit in the reverse direction from receiver's output queue to sender's input queue (for storage management/flow control)

This is an *instantaneous* definition. It could be only calculated exactly if we stopped the simulation globally and waited for delivery of all messages. However, in practice, we calculate an *estimate* of it asynchronously, without a barrier, while objects continue executing and messages continue to be transmitted.

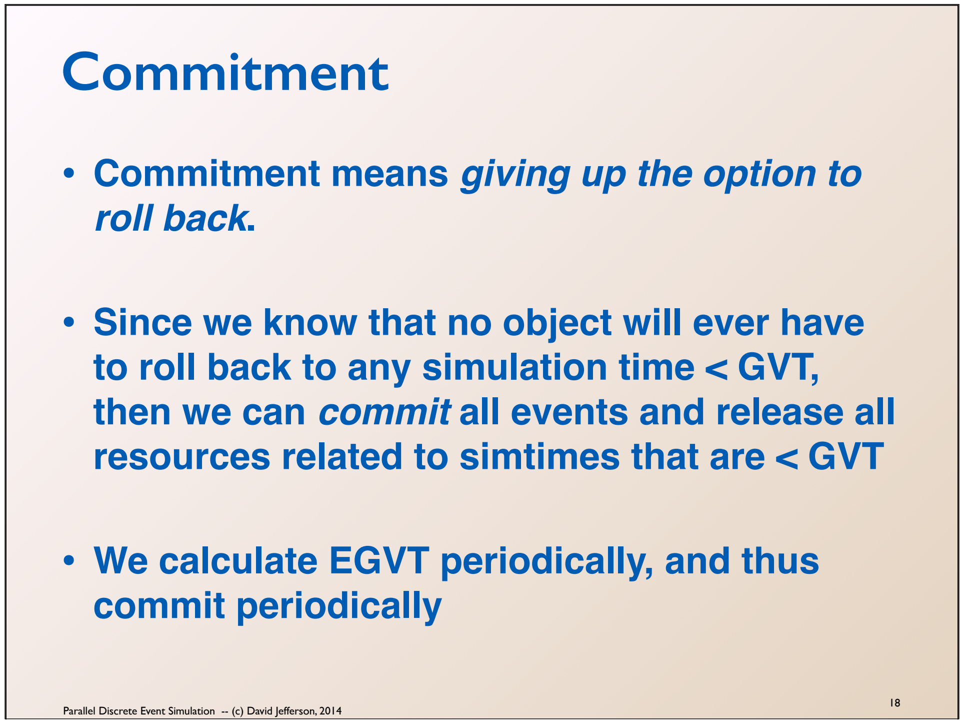
At least half a dozen algorithms for estimating GVT and broadcasting the result without any barrier synchronization have been published. All take time O(log n) where n is the number of processes. They take advantage of the fact that a message may be in transit if it has been sent but not acknowledged yet (in a low-level reliable transmission protocol). Just because a message has not yet been acknowledged, that does not mean it has not actually been delivered yet--it may have been, but the ack has not yet arrived. In that case the message will be included in the "min" operation of both sender and received. But that does not change the estimated GVT value.

The estimate is guaranteed to be low, which is the direction you want it to be. An estimate that is too high would cause Time Warp to commit events that are not yet safe from rollback--that would be a disaster. But an estimate that is too low just delays the commitment of some events that are in fact safe to commit.



GVT

- Estimated periodically and also when memory is exhausted
- Value is broadcast to all objects
- Objects then locally perform commitment operations and storage recovery
- Same quantity (GVT) is used for virtually all conservative algorithms, but used in different ways
  - safety for conservative algorithms
  - commitment and storage management for optimistic algorithms

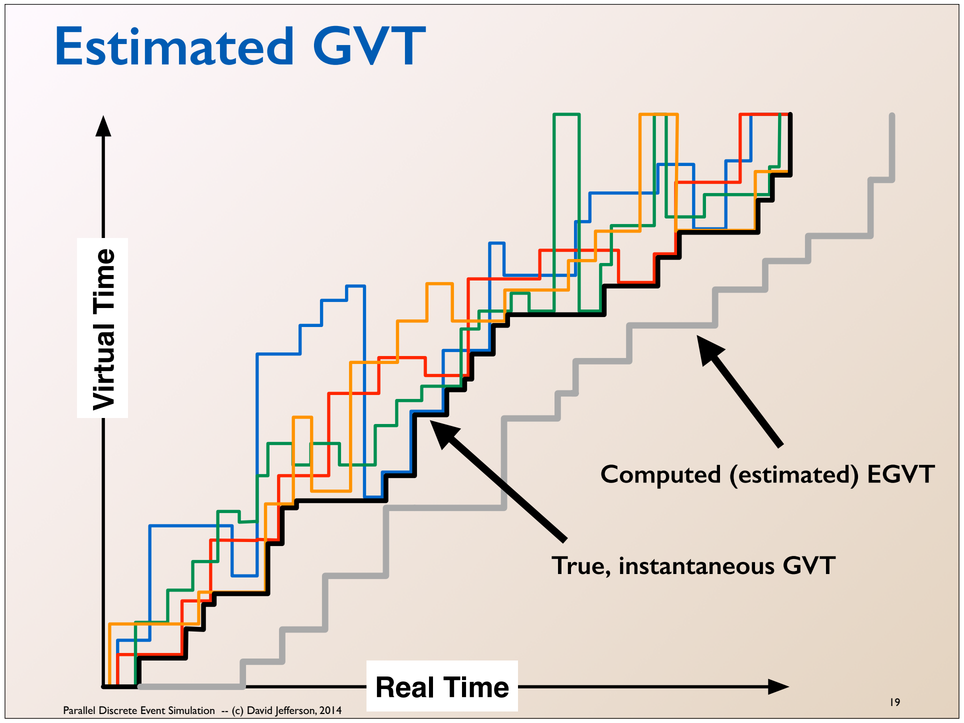


Commitment

- Commitment means ***giving up the option to roll back*.**
- **Since we know that no object will ever have to roll back to any simulation time < GVT, then we can *commit* all events and release all resources related to simtimes that are < GVT**
- **We calculate EGVT periodically, and thus commit periodically**

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

18



Estimated GVT

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

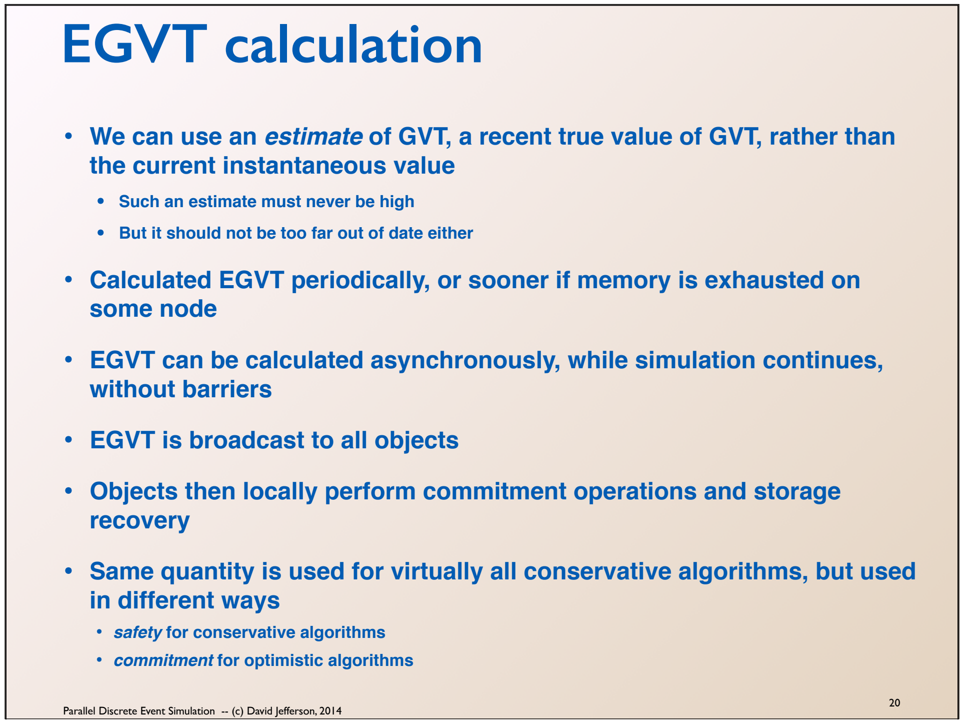
19

True, instantaneous GVT cannot be calculated without a barrier pausing the simulation which, besides yielding very poor performance as the simulation parallelism declines to zero, is also difficult to schedule if out of memory, and also unnecessary. We want to calculate GVT asynchronously, and only when necessary, without pausing event execution.

Instead of calculating true GVT, we calculate an *estimate* of GVT, EGVT, both periodically. The estimate needs to satisfy two key properties:

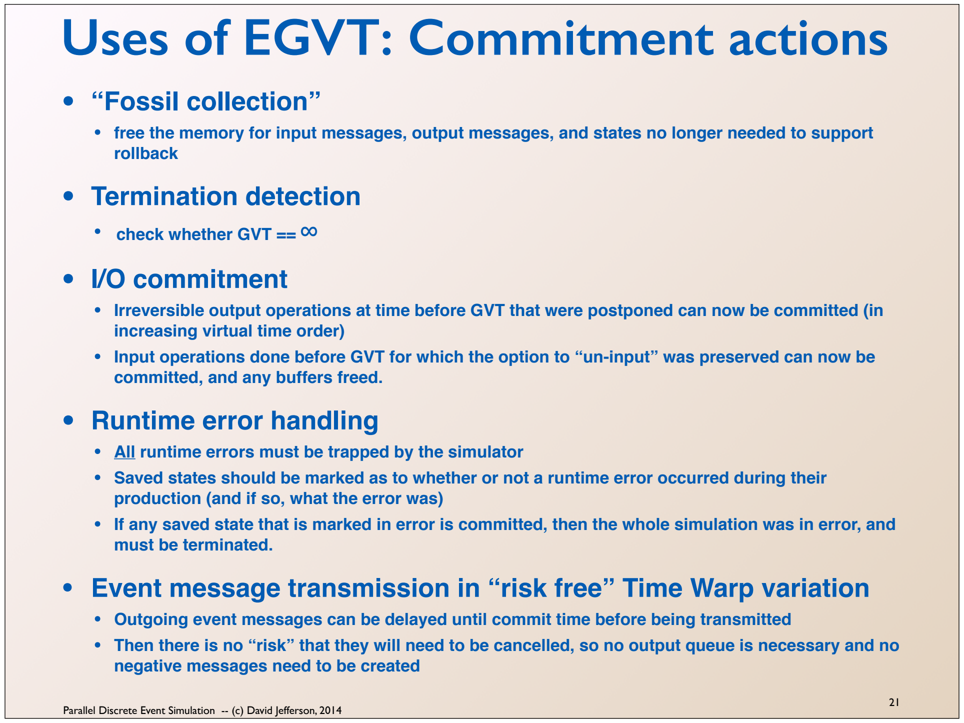
1) It is always less than or equal to true instantaneous GVT, i.e. it is a lower bound on true GVT.
2) It "tracks" true GVT, never very far behind true GVT.

Regarding the second condition, the definition of "tracking" is not obvious. We do not mean that it stays within a constant difference or constant ratio of numeric value of GVT. What we mean is that at the time a new value of EGVT is computed, EGVT is a value that GVT exceeded no longer ago than a constant amount of real time earlier. Thus EGVT is an "out of date" value of GVT, a value that GVT exceeded no more than a short time earlier. How short a time? The time it takes to compute EGVT! In other words, EGVT is greater than or equal to the value that GVT has at the start of the computation of EGVT, but less than or equal to the value GVT has when EGVT calculation is complete.



EGVT calculation

- We can use an *estimate* of GVT, a recent true value of GVT, rather than the current instantaneous value
  - Such an estimate must never be high
  - But it should not be too far out of date either
- Calculated EGVT periodically, or sooner if memory is exhausted on some node
- EGVT can be calculated asynchronously, while simulation continues, without barriers
- EGVT is broadcast to all objects
- Objects then locally perform commitment operations and storage recovery
- Same quantity is used for virtually all conservative algorithms, but used in different ways
  - *safety* for conservative algorithms
  - *commitment* for optimistic algorithms



Uses of EGVT: Commitment actions

- “Fossil collection”
  - free the memory for input messages, output messages, and states no longer needed to support rollback
- Termination detection
  - check whether GVT == ∞
- I/O commitment
  - Irreversible output operations at time before GVT that were postponed can now be committed (in increasing virtual time order)
  - Input operations done before GVT for which the option to “un-input” was preserved can now be committed, and any buffers freed.
- Runtime error handling
  - All runtime errors must be trapped by the simulator
  - Saved states should be marked as to whether or not a runtime error occurred during their production (and if so, what the error was)
  - If any saved state that is marked in error is committed, then the whole simulation was in error, and must be terminated.
- Event message transmission in “risk free” Time Warp variation
  - Outgoing event messages can be delayed until commit time before being transmitted
  - Then there is no “risk” that they will need to be cancelled, so no output queue is necessary and no negative messages need to be created

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 21

“Fossil collection” is supposed to remind you of “garbage collection”, i.e. the recycling of storage that can never be accessed again. Events that are for time lower than EGVT can never be rolled back, so it is safe to release the memory that has been used to hold the input messages, output messages, and states associated with those states.

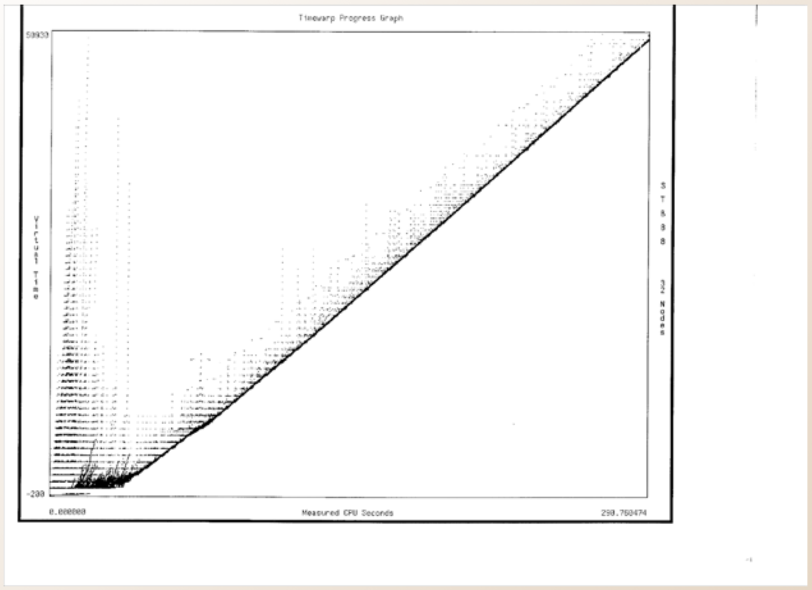
We already mentioned that EGVT == ∞ equivalent to global normal termination. Abnormal termination, however, generally occurs at some finite simulation time. Sometimes when the simulator is told to cut off execution artificially at some particular simulation time tmax, that is implemented by cutting it off whenever GVT is first calculated to be >= tmax.

Input is generally handled by buffering input data so that it can be “un-input” in the case of a rollback. The data buffers are released once GVT has increased bast the time of the event that did the inputting, to where the data will never have to be un-input.

Output is handled by buffering the data until such time as the event requesting the output is committed. Then the data can be physically written out, and will never have to be urn-output.

If an optimistic simulation needs to both read and write the same file or database, then that file or data base has to be treated as one or more full-fledged Objects in the simulation, and it need to have a simulation clock (virtual time) and it must be able to roll back.

A tactical military simulation (1)

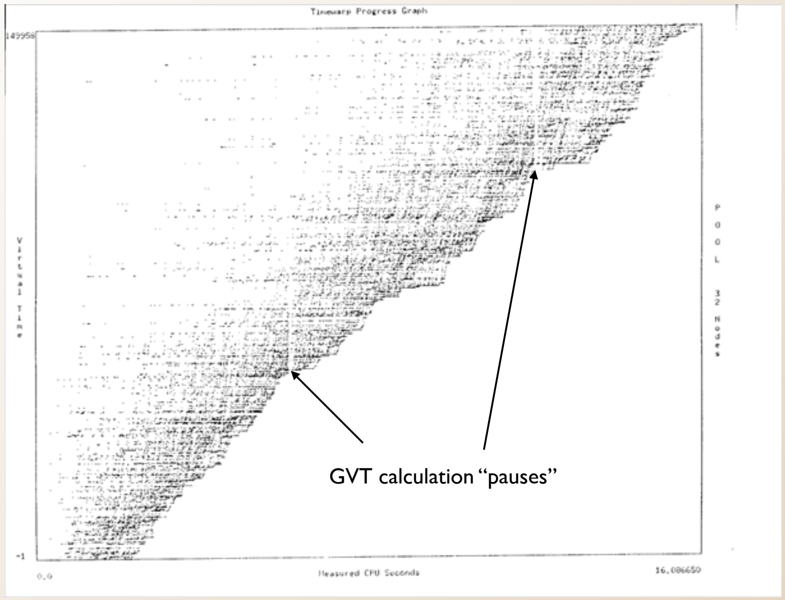


This is a simple force-on-force tactical military simulation. Each dot represents on object recorded as being at a particular virtual time at a particular moment of wall clock time. Notice the large amount of running ahead and rolling back as the simulation gets started, but then it runs quite stably and linearly thereafter. Also not that there is a definite, clean lower envelope that represents GVT.

This data was collected at JPL circa 1990 on the 64-node Caltech Hypercube (Motorola 68020 processor, 4 MB per node) running the Time Warp Operating System. Sponsored by TRADOC, U.S. Army.

Why present data so old. Partly because I have it and it illustrates the behavior of Time Warp, and partly because modern implementations have nowhere near as much instrumentation and as many analysis tools as we had in the 1980s.

A pool ball simulation on 32 nodes



Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 23

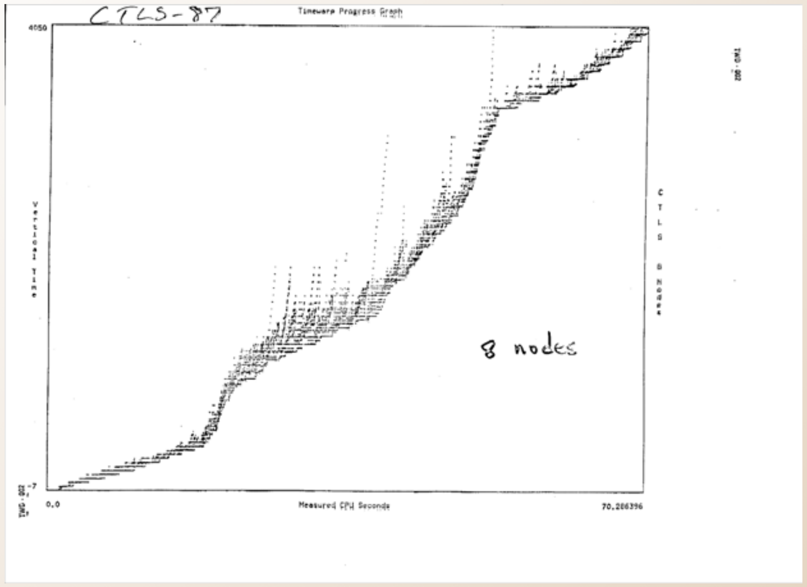
This is a trace of 16 seconds of a simulation of pool balls (or hockey pucks) moving frictionlessly and colliding with each other and with the walls of the rectangular table. Again, each point represents one object which clock reads a particular virtual time at a particular wall clock time.

The objects in the simulation are (a) pool balls and (b) rectangular subregions of the pool table. Events are (i) collisions of pool balls with each other, or (b) with the edge of the table, or (c) the edge of a pool ball entering a region, or (d) the edge of a pool ball leaving a region. (A pool ball could thus be "in" up to 4 regions at once.) The pool balls are initially still and one is whacked with a lot of energy. It takes a little time before many of the balls are engaged in activity through collisions.

GVT was calculated periodically, and you can clearly see two such times when there are momentary "pauses" in simulation activity. Although it looks like these were synchronous pauses for GVT calculation, they really were not. GVT was calculated asynchronously with simulation activity, but the processing and messages used in the GVT calculation were always of higher priority than that used for event execution and event message transmission, so it had the effect of a pause.

This data was collected at JPL circa 1990 on the 64-node Caltech Hypercube (Motorola 68020 processors, 4 MB per node) running the Time Warp Operating System. Sponsored by TRADOC, U.S. Army.

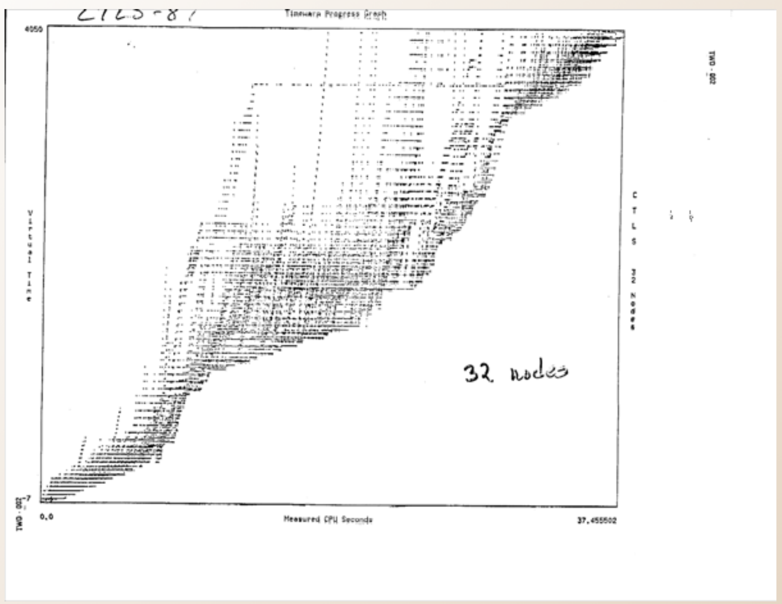
Tactical military simulation II on 8 nodes



This is a trace of a different (and more complex) tactical force-on-force military simulation. It was run on only 8 nodes, and while the speed of progress through virtual time varies considerably, as indicated by the slope of GVT, the progress was relatively tight, with only a few occasions where an object ran forward ahead of GVT and eventually rolled back. ***But compare this to the next slide.***

This data was collected at JPL circa 1990 on the 64-node Caltech Hypercube (Motorola 68020, 4 MB per node) running the Time Warp Operating System. Sponsored by TRADOC, U.S. Army.

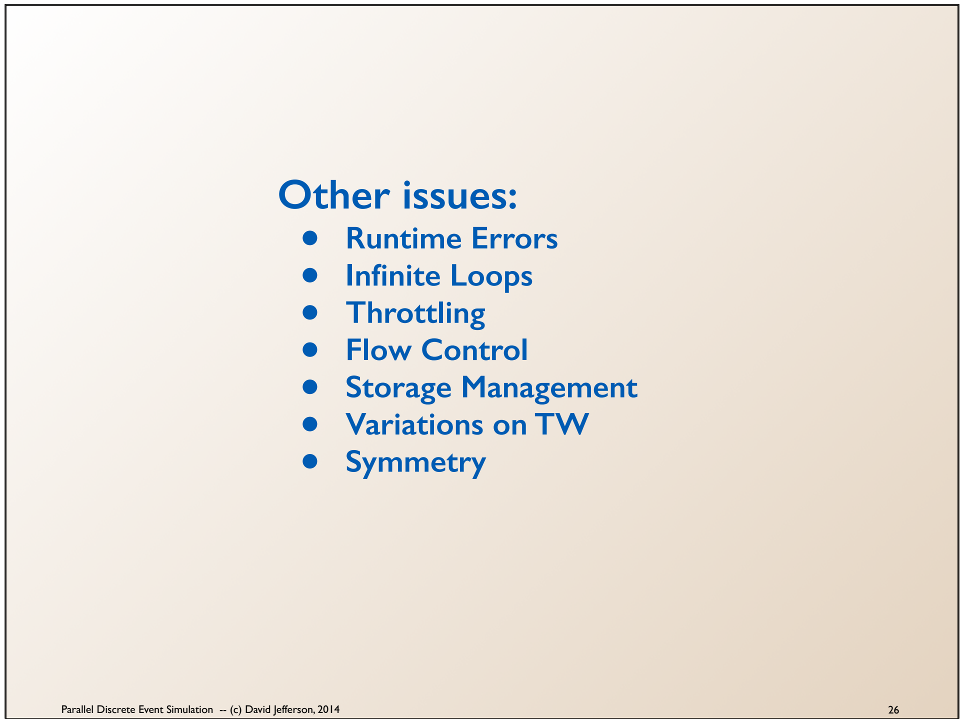
Tactical military simulation II on 32 nodes



Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 25

This data is for exactly the same model as the previous slide, but spread over 32 nodes instead of 8 nodes. Notice that the processors now have much more free time to execute far ahead speculatively, and this there is a lot more rollback. But still, even though there was a lot more “wasted” activity, this execution was still ***almost twice as fast*** are the one on the previous slide, completing in 37.45 seconds, compared to 70.28 seconds on the previous slide.

This data was collected at JPL circa 1990 on the 64-node Caltech Hypercube (Motorola 68020, 4 MB per node) running the Time Warp Operating System. Sponsored by TRADOC, U.S. Army.


Other issues:

- Runtime Errors
- Infinite Loops
- Throttling
- Flow Control
- Storage Management
- Variations on TW
- Symmetry