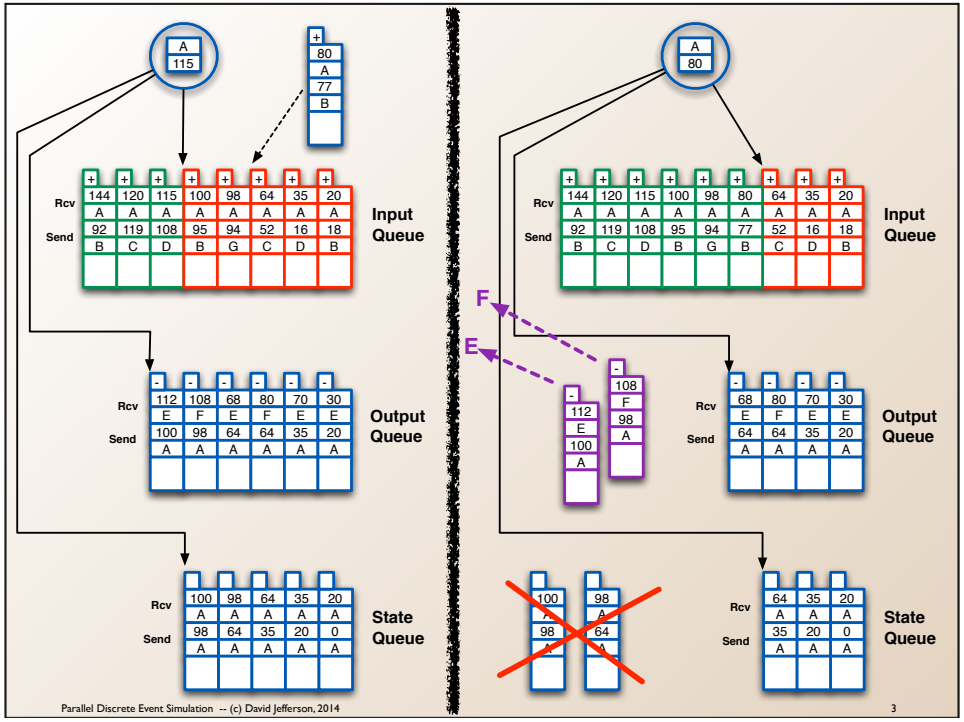
Parallel Discrete Event Simulation Course #10

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-652953

Reprise



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 ??

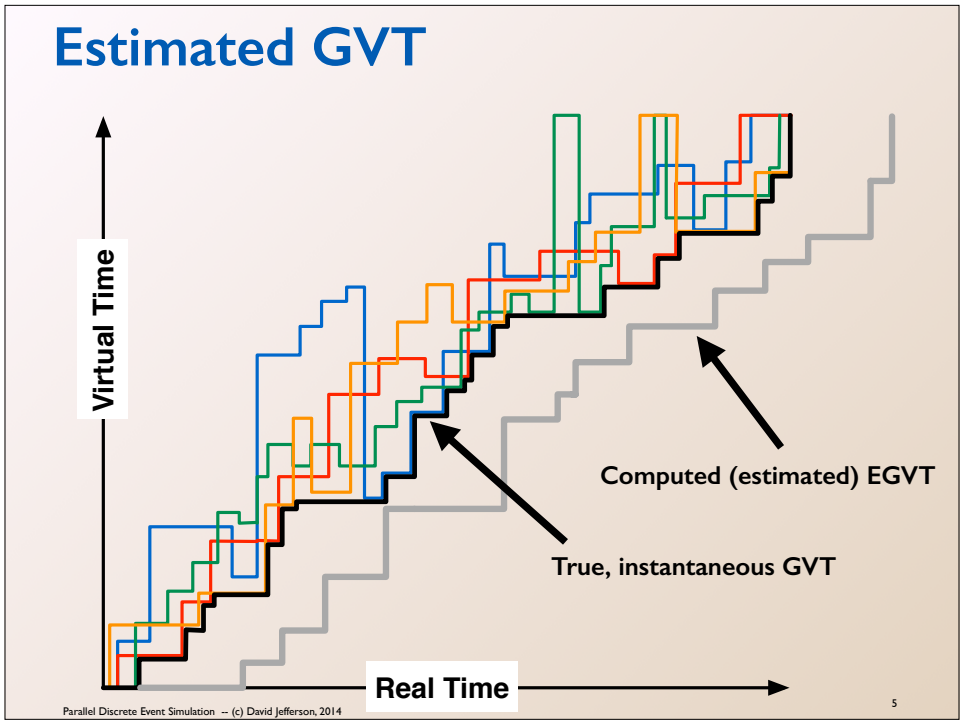
The rollback consists of the following steps.

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.)

Global Virtual Time (GVT) and Commitment: Global Synchronization



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

GVT estimation algorithms (EGVT)

- **Many algorithmic variations published in the literature**
  - **Synchronous and asynchronous**
  - **One-sided and two-sided**
  - **Require end-to-end message acks or not**
- **The most useful ones have these properties**
  - **Execute concurrently with event execution**
  - **Take (log n)-time (assuming constant message latency)**
- **For a survey see**

  Fujimoto, *Parallel and Distributed Simulation Systems*, Wiley, 2000, Section 4.4

  or look up "simulation GVT calculation" in Google

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

"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.

Other issues

- Runtime Errors
- Infinite Loops
- GVT estimation algorithms
- Scheduling
- Shared Memory implementations
- Multicore implementations
- Flow Control
- Storage Management
- Throttling
- TW as an operating system
- Variations on rollback
- Variations on message cancellation
- Mixed conservative and optimistic synchronization
- Checkpoint / Restart
- Reverse computation
- Measured performance at extreme scale
- Fault recovery
- Load balancing
- Symmetry
- Applications beyond PDES

Runtime errors

- During a “correct” event:
  - Runtime error caught
    - Will behave the same way as it would in the sequential algorithm
  - Runtime error uncaught (e.g. errant pointer)
    - Both optimistic and sequential algorithms may misbehave, but possibly differently.
- During an “incorrect”event, executed from an “incorrect” state
  - Runtime error caught
    - TW should stop executing the event, mark the object as being “Erroneous”, and “output” a tentative error message
    - If the event is rolled back, remove the “Erroneous” mark, and proceed normally
    - If an Erroneous event is committed, then the error message is output and the simulation should be terminated globally
  - Runtime error uncaught (e.g. errant pointer)
    - Program will misbehave, ***and that misbehavior may not be corrected by subsequent rollback!***
    - Hence, we really need a ***proof*** that an event method cannot cause an uncaught error from ***any*** state before we can safely execute it optimistically.

Infinite loops within events

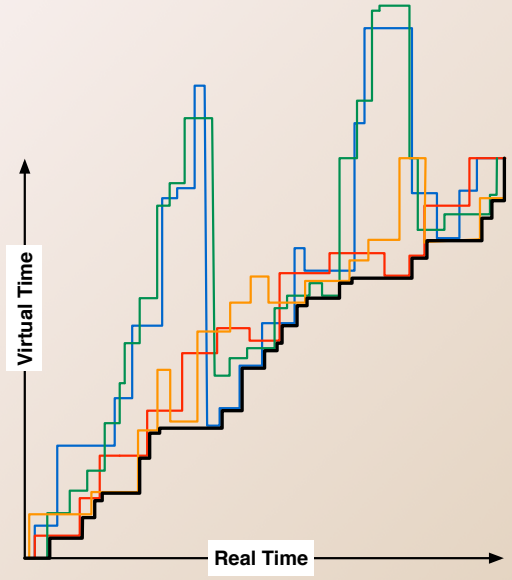
- During a "correct" event:
  - The event behaves the same way it would in the sequential algorithm
  - The infinite loop is then "correct"
  - GVT never increases
  - Tough!
- During an "incorrect"event, executed from an "incorrect" state
  - If events are interrupted when stragglers arrive that would cause a rollback
    - Eventually a straggler will arrive (guaranteed) which will cancel this event or require a rollback to an earlier state.
    - Execution will proceed properly and duplicate behavior of the sequential algorithm
  - If events are executed uninterruptably
    - The infinite loop is never interrupted
    - GVT never increases
    - This is *not* what the sequential algorithm does!
    - Hence, we really need a ***proof*** that an event method cannot cause an infinite loop from ***any*** state before we can safely execute it optimistically.

Scheduling

- **Assume**
  - k cores per compute node
  - multiple (sequential) processes or tasks per compute node
  - multiple objects or LPs per process or task
  - Each process (task) can be viewed as a "fat object"
  - LVT of a process (task) is the minimum LVT of any object in it
  - A process is only blocked (LVT = ∞) if all objects in it are blocked
- **Natural scheduling discipline for processes in shared memory and objects within processes, is LVTF (lowest virtual time first)**
  - System functions (EGVT calculation, storage reclamation) prioritized over event execution
  - On each compute node always run the k (non-blocked) processes with lowest LVTs
  - Within each process, run the one object with lowest LVT
  - Preëmptive scheduling within events preferred, but not always implementable

Throttling excess optimism

- Conservative algorithms often make slower progress than they should -- they are too conservative.
- Optimistic algorithms often progress (locally) faster than they should, and then have to roll back a lot of work -- they can be too optimistic
- The costs of excess optimism
- Throttling is any mechanism that reduces excess optimism
- Throttling algorithms in the literature
  - Moving Time Windows (w)
  - Breathing Time Warp



What is the cost of excess optimism?

1) Extra storage impact
2) The cost of local rollback, which may be mostly storage reclamation, but in some algorithms (like reverse computation) may require compute time proportional to that used in forward execution.
3) Overhead costs in other processes for dealing with messages that should not have been sent (unless conservative message transmission was used).

Moving time windows: This throttling protocol prevents runaway execution into the future by setting a simulation time window size w. The protocol requires an Object to block once it reaches time (GVT + w), i.e. there is a global barrier at (GVT+w). Once all processes have reached that time, then GVT is re-estimated, and and a new (GVT+w) window is enforced.

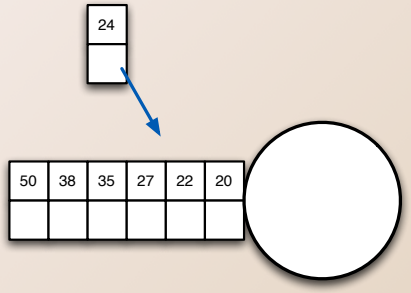
Window size w can, of course, change dynamically and adaptively. But what value should w have? And when should it be increased or decreased, and by how much? Today this is handled empirically. Without a good theory for adaptively setting w, I find it unsatisfactory.

Variation: The window size can be in number of events instead of units of simulation time. That has the virtue that it does not use the arithmetic properties of simtime, but only the total ordering properties. But otherwise the same issues arise as when the window size is in units of simulation time: how big should the window be, and how should it be increased and decreased.

Storage Management in Time Warp

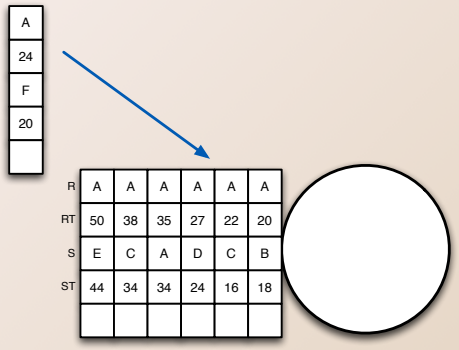
Message flow control in PDES

- Flow control is the memory management problem that arises when producers send data faster than consumers process it.
- For graph-oriented, conservative PDES protocols with FIFO order preservation on each channel then *windowing protocols* on a per-channel basis work fine.
- But windowing does not work for optimistic PDES:
  - No channelization, so windowing does not apply
  - Messages not transmitted or processed in FIFO order!
  - Messages not immediately deleted even after they are processed!
  - When memory is full, a message with a low time stamp may still arrive and there may be no room for it!



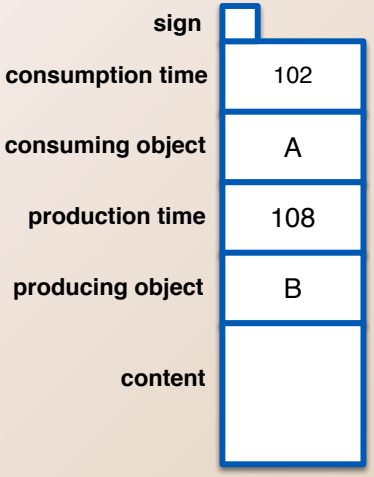
Optimistic flow control

- Suppose a message from F arrives at A when A's memory is full? What do we do?
- We cannot just block the sender, F — it is already too late!
- We cannot just drop the message, because it has to be processed before some others we have enqueued.
- EGVT update may release some space, but it is not guaranteed to.
- Answer: *Send back the highest RT message from A to E, in this case causing E to roll back to 44, before E sent it.*
- This is a message sent *backward in space* and *backward in virtual time*!
- In this protocol a message from F to A causes E to roll back!



Generic Time Warp storage item

- All chunks of TW storage (states and messages) have similar "headers" with specific producer and consumer events
- The "lifetime" of the object is specified by its production time (PT) and consumption time (CT)
- If CT < EGVT, the chunk can be fossil collected
- If PT > EGVT, the chunk is speculative and not absolutely required yet and can be deleted with rollback of producer.



Cancelback protocol: Generalization to all dynamic storage

- **Whenever memory has been fossil-collected, storage is full, and a new item (state, input msg, output msg) has to be allocated:**

  Find *any* item in memory (including the new item) with ST > EGVT (preferably with maximal ST):

  if none: `fail: out of memory`

  if item is a state: delete it

  if item is an output message: send it forward

  if item is an input message: send it backward

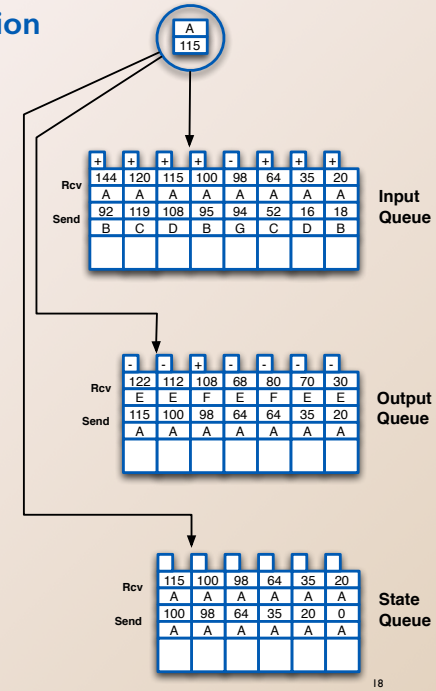
  Roll back "victim" object if necessary

- **Note: Item in memory can be found in a queue of another *completely unrelated object* in the same memory pool.**

- **Provably space optimal (but not time optimal)!**

- **See**

  Jefferson, David, "Virtual Time II: The Cancelback Protocol for Storage Management in Time Warp", Proc. ACM Symp. on the Princ. of Distributed Computing (PODC), Quebec City, Quebec, August 1990



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

18

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 model 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

This shows the list of challenges we have to overcome for the Time Warp algorithm, or any optimistic PDES algorithm, to be practical.

In green I have listed those that we have covered so far

Time Warp as an operating system

- **temporal coordinate system:**
  - nothing comparable in OS world
- **process name space:**
  - flat space, like MPI, unlike other OSs
- **process (LP) scheduling:**
  - LVTF - lowest virtual time first
- **synchronization:**
  - virtual time, rollback
- **determinism:**
  - guaranteed, unlike with most OSs
- **interprocess communication:**
  - timestamped event messages to named objects
- **queueing:**
  - virtual time order, with antimessage annihilation
- **flow control:**
  - message sendback
- **memory management:**
  - fossil collection of past, throttling or cancel back of future
- **delayed commitment**
  - tied to EGVT
  - nothing comparable in OS world
- **memory alloc / dealloc:**
  - freeing tied to commitment
- **error handling:**
  - tied to commitment
- **I/O:**
  - tied to commitment
- **job termination:**
  - tied to commitment

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 20

One way to view Time Warp is as a special purpose parallel operating system. It can be viewed as having all of the architecture of a parallel OS, but different algorithms in place of most of the OS components.

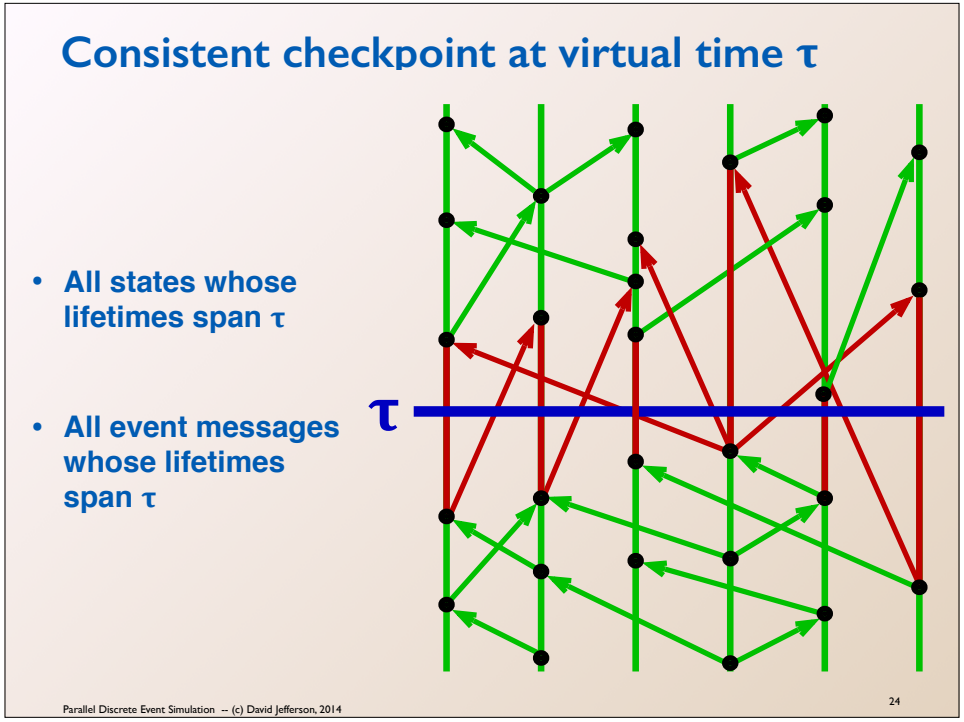
1) *Temporal coordinate system:* TW offers a temporal coordinate system (address space), namely virtual time, which has no analog in other OSs.
2) *Process name space:* TW has a flat, global process name space not offered by most parallel OSs, although it is offered by MPI.
3) *Process scheduling:* TW generally uses LVTF (lowest virtual time first) scheduling among logical processes. The only comparable scheduling algorithm used in conventional OSs is a strict priority-driven scheduling. LVTF can be replaced by any scheduling mechanism that eventually executes the lowest
4) *Synchronization:* TW uses virtual time and rollback (with some process blocking) as its fundamental synchronization mechanism. Other OSs use various kinds of process blocking mechanisms in the context of locks, semaphores, messages waiting, etc.
5) *determinism:* TW guarantees deterministic execution; other OSs do not
6) *interprocess communication:* TW uses event messages signs and both sender and receiver event stamps; other OSs have a variety of mechanisms for interprocess communication, including shared memory, pipes, TCP/IP, signals, MPI, etc.
7) *queueing:* TW queueing is a priority queue with virtual Time as priority, but with the added feature of antimessage cancellation. Most message queues in other OSs are FIFO.
8) *flow control*: Flow control in TW can use message sendback, whereas that is not possible in other OSs because the sending processes cannot be rolled back.
9) *memory management:* process memory is managed by a combination of fossil collection (reclamation of space with time stamps in the past) and cancel back (reclamation of space in the future). Nothing like it exists for conventional OSs.
10) *delayed commitment:* in TW commitment of irreversible operations is delayed until commitment time. In most OSs, since there is no rollback, and commitment time is immediate. (Notable exceptions are DB management systems viewed as OSs where there is also delayed commitment.)
11) *memory alloc & dealloc:* Memory deallocated by events is delayed until commitment in TW. No such feature of convention OSs.
12) *error handling:* In TW it is bound up with commitment. Not so in other OSs.
13) *I/O:* In TW it is bound up with commitment. Not so in other OSs.
14) *job termination:* In TW it is bound up with commitment. Not so in other OSs.

End Reprise

Checkpointing

Checkpointing

- **A checkpoint in other systems generally means a snapshot in *real time*.**
  - **It requires a (real time) barrier to start the checkpoint**
- **A snapshot in Time Warp take place at an instant of *virtual time*.**
  - **A checkpoint at virtual time τ is a snapshot of all states at time τ and all messages that were sent before time τ but are received after τ**
  - **This also requires a barrier of sorts, but it is a *virtual time barrier*!**



The blue horizontal line represents virtual time **τ** at which the checkpoint is taking place.

Vertical segments represent states as computed by the previous event and input to the next event in the same object.

Arrows represent event messages.

A consistent checkpoint requires saving all of the states and event messages that cross **τ**, i.e. are produced at or before time **τ** and are consumed after **τ**. The red states and event messages in this diagram represent a consistent checkpoint at time **τ**. The event messages should be stored with their receiving objects.

On restart from this checkpoint, the saved states are restored to their respective objects, and the saved messages become the input queue of their respective receiving objects. The virtual clocks of all objects are set to **τ**, and EGVT is **τ**.

At that point, normal execution is resumed.

Checkpoint calculation

- A checkpoint at time τ consists of all messages and states that "straddle" time τ
- To take a checkpoint at time prescheduled time τ:
  - Whenever an object crosses time τ going forward, it saves a local snapshot of its state.
  - Whenever an object rolls back over time τ it throws away the local saved snapshot
  - Whenever a message is sent before time τ to be received after time τ, the message is saved by the *receiver* as part of the local snapshot (unless annihilated).
  - Whenever a message is sent in reverse from a time after τ to one before τ, remove it from the snapshot.
- When EGVT exceeds time τ the entire snapshot is committed!
- Note that this algorithm is
  - barrier free, and
  - nonblocking!
- To take a checkpoint at time NOW (at the behest of a single LP)
  - Pause any commitment activity in progress
  - Roll back all objects ahead of time NOW back to time NOW
  - Use the above checkpoint algorithm and continue event execution
  - Continue commitment if paused
  - When EGVT exceeds time NOW the entire snapshot is committed!
- Checkpoint at time NOW is generally impossible for conservative mechanisms

Time Warp Variations

Variations on state saving and rollback

- **Snapshot state saving**
  - Binary state save (including unallocated heap and garbage)
  - Binary state save with serialized heap (reachable, non-garbage only)
  - Periodic state save (save only every k events)
  - Incremental page-based state save — save only dirty pages
  - Periodic incremental — save save only dirty pages every k events
  - Linux fork() as state save
- **Reverse computation (!)**
  - Source language one-time state save
  - Source language incremental state saving during events
  - More sophisticated techniques
- **Issues**
  - Cost of state saving
  - Cost of restoring state during rollback
  - Variables that are *not* saved in the forward direction, and *not* restored on rollback
  - Suitability for fault recovery

State saving is usually the big overhead in Time Warp. States, by which we mean both static and heap storage (but not stack storage, since the stack will be empty between two events, which is when we save states) may be as little as a few tens of bytes in size, or many times larger, up to several GB in extreme cases. The heap in particular is very slow to save, because we either have to save all of the garbage data as well as the good data, or else we have to traverse and serialize only the non-garbage data. Either option is slow. (Some systems just prohibit heap storage altogether to avoid the problem.) So the consensus view is that saving whole states between every two events is untenable--it is usually way too slow. Something must be done to reduce state saving overhead.

Periodic state saving: This technique amounts to saving the state of an object every 10 or 20 (or some other number) of events instead of between every two events. This reduces the amount of time spent saving states by 90% or 95% perhaps, but when a rollback occurs, the cost is that you usually have to roll back farther than strictly necessary because the exact state you wanted to restore was not saved. And you also have to execute a few more events in the forward direction that you otherwise would not need to, just to re-create the correct state you were trying to roll back to.

Incremental state saving by pages: If the simulator has access to the page tables (as it does if the OS permits it, or if the simulator is the operating system) then it is only necessary to save the pages of state that have changed during an event, rather than all of the pages of the state. This will require applying several page-table-deltas in order to reconstruct a state during a rollback. These days no one does this because (a) pages are too large, so you end up still copying too much data when only a small part of a page is written during an event) and (b) operating systems so not provide APIs for this anyway.

Incremental state saving by variables: Here we try to save state deltas by logging the old value of each variable the first time it is changed during an event. We use the log to restore all old values in the case of a rollback. This either a lot of hand-work on the part of the programmer or some very sophisticated compiler help.

Variations on message cancellation

- **Aggressive Cancellation**
  - During rollback from t2 to t1, immediately cancel all messages sent at times t1 <= t < t2
  - This is "classic" Time Warp algorithm discussed so far
- **Risk Free execution — no cancellation**
  - Treat outgoing messages as output
  - Do not transmit them until EGVT exceeds the send time of the event message
  - No antimessages necessary
  - Primary rollbacks still occur, but no secondary or tertiary rollbacks
- **Lazy cancellation**
  - During rollback from t1 to t1, do not immediately send anti messages
  - Wait to see if they are necessary
- ***Arbitrary mixed variations of these mechanisms work fine!***

Lazy Cancellation

Lazy Cancellation

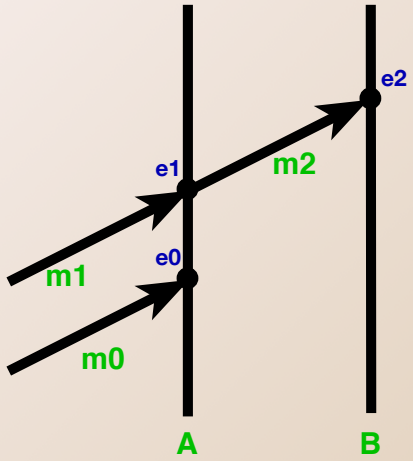
- During rollback from t2 to t1, do *not* cancel any output messages.
- During re-execution forward from t1 to t2, compare messages generated this time to those sent last time:
  - Messages generated this time and not sent previously are transmitted this time
  - Messages generated by the code this time that were sent previously and are already in the output queue are not retransmitted and cause no action
  - Messages sent previously but not re-generated this time are cancelled, i.e. their antimessages transmitted
- Lazy Cancellation leaves messages in the future of the input queue, as well as the past
- Lazy cancellation is one optimistic mechanism that allows beating the critical path lower bound

Lazy Cancellation can beat the Critical Path performance bound!

- Lazy cancellation can win when two events affect disjoint parts of the state.
- Special case: Events that do not affect the state of the object, but just cause message transmission
- Lazy cancellation is not the only variant that can beat the critical path, but it is the most useful.

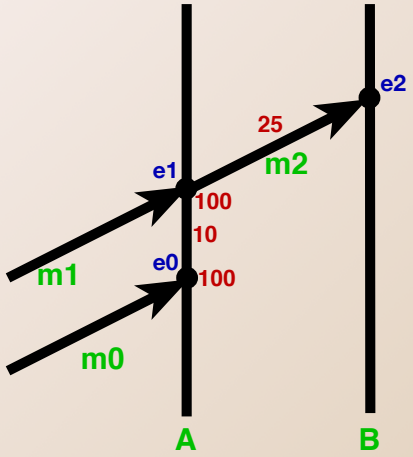
Critical Path Example

- This is the "correct" causal sequence relations among the events:
  - e0 occurs before e1
  - e1 schedules e2
- Event message m1 causes A to send event message m2 to B



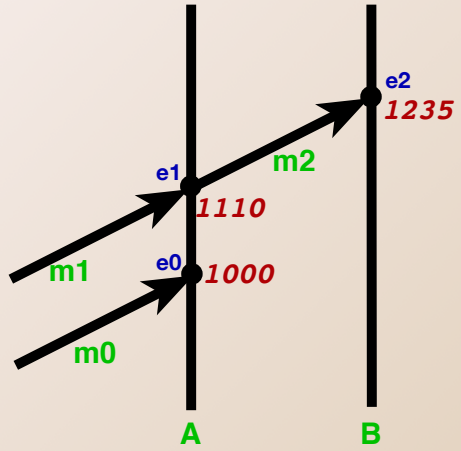
Critical Path Example

- Events take 100 μsec of wall clock time
- Event transitions take 10 μsec
- Messages take 25 μsec
- Critical path argument applies to *committed* messages, states and events



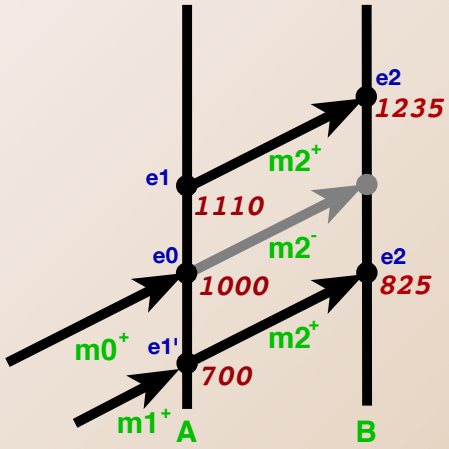
Critical Path Example

- Assume the critical time for event e0 is 1000. Then
  - critical time for e1 is 1110
  - critical time for e2 is >= 1235



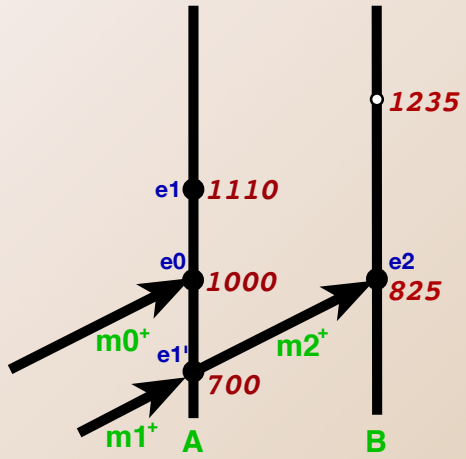
Aggressive cancellation

- m0 and m1 arrive out of order.
- m0 arrives at its critical time, but m1 arrives early
- m1 is executed in the wrong state, and sends m2, which causes B to process m2
- When m0 arrives, A must roll back to the virtual time of m0 and send the antimessage for m2.
- The antimessage may cause B to roll back even though m2 was correct and so was the processing of e2.
- e1 is executed again when A is in the correct state, and again sends m2 to B.
- e2 is executed again, even though it was correct the first time



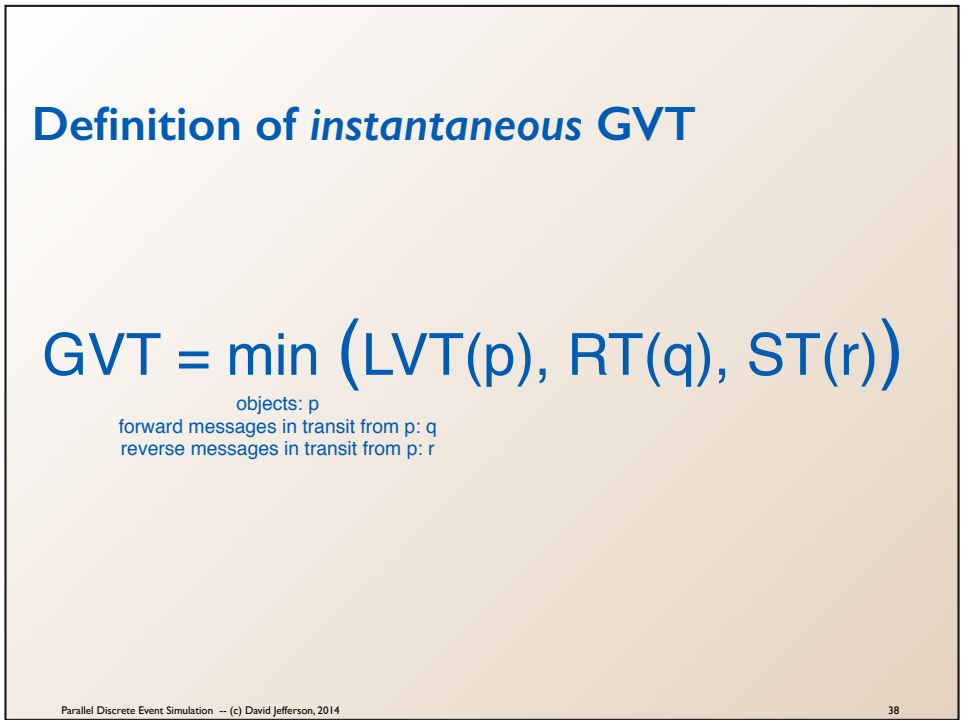
Lazy Cancellation

- m0 and m1 arrive out of order.
- m0 arrives at its critical time, but m1 arrives early
- m1 is executed in the wrong state, and sends m2, which causes B to process m2
- When m0 arrives, A must roll back to the virtual time of m0 *but it does not at this time send the antimessage for m2.*



Mixed conservative and optimistic synchronization

- GVT redefined to include distinguish between conservative and optimistic objects
- Conservative objects contribute to GVT through their lookahead value, not their LVT
- Optimistic objects are always at times >= EGVT
- Conservative objects are always at times <= EGVT
  - They must block rather that cross it until EGVT recomputed
- Conservative objects lookahead values must always be >= EGVT



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.

Definition of *instantaneous* GVT *with mixed conservative and optimistic objects*

GVT = min (LVT(c)+LKHD(c), LVT(p), RT(q), ST(r))

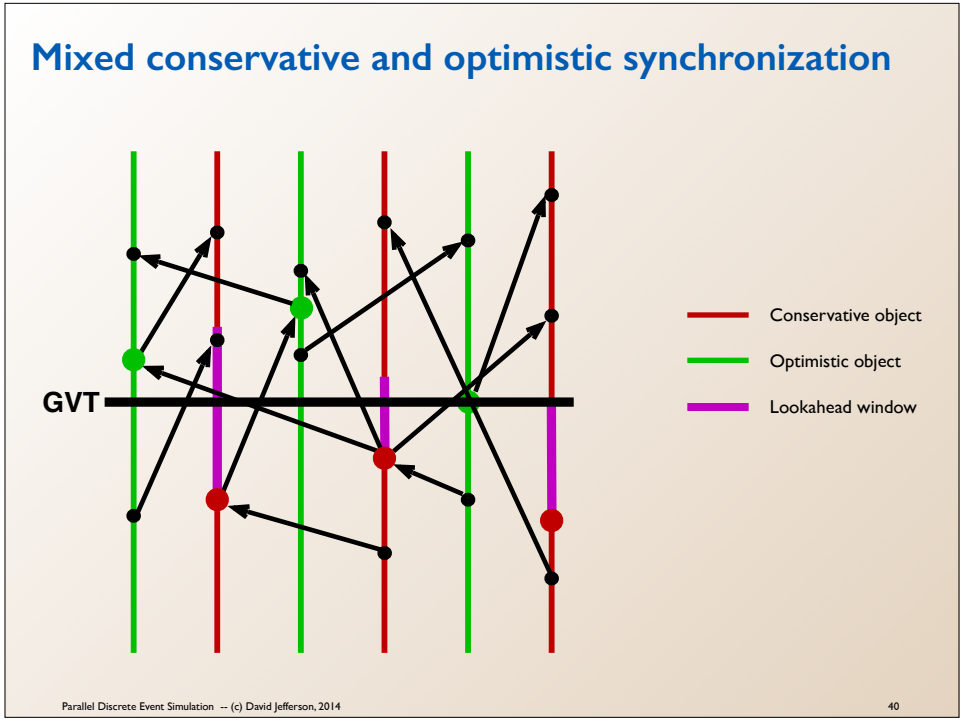
conservative objects: c
optimistic objects: p
forward messages in transit from c or p: q
reverse messages in transit from p: r

LVT(p), LVT(c) = Local Virtual Time, i.e. the simulation clock value of optimistic object p or conservative object c
LKHD(c) = LookAhead value for conservative object c, i.e. lower bound on the timestamp of any message it will send in the future
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.



Conservative objects must always be at time <= GVT
Conservative objects have a lookahead window and their lookahead time must be >= GVT
Optimistic objects must always be at times >= GVT

End