Parallel Discrete Event Simulation Course #14

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-654176

Next Two Weeks!

- Two "finale" lectures
- I will make an audacious but speculative argument
  - Optimistic parallel discrete event simulation can be viewed as a new parallel programming paradigm for many scalable applications, *not just simulation.*
  - Put another way: From this point of view all sufficiently large cooperative parallel computations can fruitfully be viewed as simulations
- Will touch on many exascale (and beyond) issues
  - synchronization
  - debugging
  - fault recovery
  - load balancing
  - power management
  - space-time symmetry
  - parallel programming methodology
- Looking for your feedback and ideas!

Virtual Time for Most Large Scale Computations

Parallelism, communication, and synchronization today are too complex!

- Multiple units of parallel computation
  - processes, MPI tasks, chares, logical processes, application threads, kernel threads, hardware threads, SIMD elements
- Multiple communication mechanisms
  - MPI messages, MPI collectives, native packets, pipes, sockets, RPC, shared memory, shared files
- Multiple synchronization primitives
  - interrupts and interrupt masking, atomic R-M-W instructions, SIMD synchronization, locks, semaphores, messages, critical sections, timeouts, transactions, barriers, and … rollback, and virtual time
- ***Most of these constructs are fundamentally nondeterministic!***

Simplify some of the complexity by using Virtual Time

- Virtual time is a *temporal coordinate system* that plays a *logical* role in the computation
  - It is an abstraction of *real time*, much as an *address space* is an abstraction of *real space*
  - It has many of the same properties of Newtonian time
  - It allows time to be *addressable* and *random access*, just as we make space addressable.
- Heretofore the only broad computational paradigm that makes explicit use of a temporal coordinate system is simulation.
- But what if we view *any* computation as taking place in the context of both a temporal and spacial address space — *virtual space-time*?

***Virtual time* — not just for simulation any more!**

Two distinct recommendations

- Consider framing *any* large computation as a virtual time computation.
- Consider optimistic (rollback-oriented) implementation of virtual time.
- How would it affect generic issues in all large scale computation … ?
  - debugging
  - synchronization
  - fault recovery
  - load balancing
  - energy management
  - programming methodology

Determinism

Determinism

- Repeated execution of the same binary on the same platform with identical inputs yields identical outputs
  - Bit for bit
- It includes *scale independence* and *configuration independence*

All virtual time programs are deterministic!

- Replace threads, processes, chares, etc. with *objects* of the kind we have described in this course.
- Replace *all synchronization and mechanisms* with structures defined in terms of virtual time
- Replace *all communication mechanisms* with structures defined from event messages
- Require a *deterministic* virtual time tie-breaking rule
- Exclude access to real time, real randomness, real node addresses, and real memory addresses inside of event methods
- Then – It is impossible to write a nondeterministic virtual time program!
- This is true for both conservative and optimistic implementations.

Significance of Determinism to High Performance Computing

Debugging methodology can be greatly simplified

- Any nondeterminism observed is immediately known to be a fault or else a bug in the underlying runtime system, OS, or hardware.
- Classic "sequential" reasoning and instrumentation used to narrow down the location of a bug works for parallel applications, because the semantics of virtual time are sequential.
- No timing-dependent Heisenbugs are possible at the application level — passive instrumentation that affects real time behavior does *not* affect virtual time behavior
- Breakpoints can be introduced *in the runtime system* (without resorting to instruction replacement) to pause the global computation at a particular virtual time for closer inspection with power tools.
- Time stepping through various intervals of virtual time is possible and reproducible.
- With optimistic (rollback-oriented) implementation, fast backward time-stepping through virtual time can be implemented based on suppression of fossil collection (until you run out of RAM).
  - We already discussed low-overhead, non-barrier optimistic checkpointing as well earlier in the course

Arbitrary fault detection and recovery

- Because of application-level determinism, any discrepancy indicates a failure in system layers below
  - transient or permanent HW failure of some kind (including in the comparison)
  - a bug in runtime system or OS
- Arbitrary single faults affecting the application are *detectable* by *duplicate* computations
  - Not just memory or communication faults — *transient processor faults also*
  - Arbitrary single faults are correctable by triplicate computation and voting
- These techniques only work for arbitrary single faults if the application is deterministic
  - With a nondeterministic computations, the fact that states and messages do not disagree means nothing.
  - There may not even *be* corresponding states and messages
- In addition, with optimistic virtual time, transient faults are also *correctable* by quasi-local local rollback
  - No need to restore global checkpoint!

Transient processor faults are not even detectable in today's architectures — there is no hardware for it today and in general there can't be without duplication somewhere.

Even if detectable, such faults are not correctable without even more mechanism.

Synchronization

Synchronization is about the relative timing of events in a computation

- It is about constraints of the timing relationships among events
  - It is about the *total order of events* only,
  - … not their real time speed or performance
  - Recall that with any implementation of virtual time the runtime system is concerned *only with the ordering properties* of virtual time values, *not the arithmetic properties*
- We can thus re-interpret the definitions of various synchronization constraints as referring to *virtual time* instead of real time
- Examples
  - mutual exclusion
  - database transactions
  - barriers

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 14

Because virtual time has exactly the properties of real time that synchronization depends upon, we can consider every synchronization primitive and re-interpret its definition and/or implementation with virtual times substituted for real time. A lot of very interesting things happen.

Mutual Exclusion

- Two actions, P and Q, are *mutually exclusive* if they *do not overlap in time*, i.e. either P completes before Q starts, or vice-versa.
  - Or if they are executed in a semantically equivalent way
- Usually implemented as a race: The first one to start prevents the second one from starting until the first one completes
- It is *nondeterministic* which one wins.
- Generally implemented with busy-waiting, locks, semaphores, etc., and proper prelude and postlude code in P and Q.
  - Deadlock is a hazard if not done correctly
- These implementations apply only to *sequential segments of code* that somehow share access to a lock or semaphore
- They are not naturally generalizable to the case where P and Q are themselves big parallel computations.
- These implementations preclude any parallelism between P and Q

Mutual Exclusion in Virtual Time

- Instead, interpret the definition of mutual exclusion to mean *non-overlapping in virtual time* rather than in real time
- Simply allocate an interval of virtual time to P and a non-overlapping one to Q.
- No locks or semaphores required. No prelude or postlude code required.
  - Deadlock is impossible!
- Works even if P and Q are arbitrarily large and complex parallel computations, not just sequential fragments
- P and Q can execute in parallel as long as they do not conflict. If they do, one the one in the later virtual time interval will (partially) roll back.
- The two may execute in either order or in parallel, but …
  - If there is a conflict, the one allocated the earlier virtual time interval always "wins"
  - This is *deterministic mutual exclusion*
  - Regardless of actual execution order their committed results will be as if executed in virtual time order
- You cannot write *nondeterministic mutual exclusion* in virtual time
  - … unless you assign virtual times by nondeterministic mechanisms, which we have excluded
- On the other hand, you cannot write *deterministic mutual exclusion* with locks or semaphores
  - … which are inherently nondeterministic

Database transactions

- An *atomic* transaction `P` is an action that is, *in effect,* mutually exclusive with *all* other actions in the computation
  - i.e. during its execution no other code can modify or observe its intermediate states
- Database transactions generally have to be atomic
  - Order of transaction execution generally nondeterministic
  - Optimistic, multiversion concurrency control mechanisms go part way toward optimistic virtual time synchronization but …
    - use transaction abort rather than full rollback
    - are still nondeterministic in the serialization order of transaction execution
  - None of the concurrency control mechanisms are readily generalizable to arbitrary internally parallel or distributed atomic actions
  - Generalization to arbitrary *nested* transactions is complex

Virtual Time Atomic Actions

- In Virtual Time *all single events are already primitive atomic actions*
- An arbitrary complex parallel or distributed action can be made atomic by allocating it a window of virtual time that does not overlap that used by any other part of the computation
  - All transactions must be *allocated* short segments of virtual time unique to themselves — that's all there is to it
  - *Nested transactions* are easily accommodated by allocating *nested regions of virtual time* to them.
- Even distributed transactions that access the same data objects can proceed in parallel
  - If there is a conflict, the one with the lower region of virtual time will always win
  - Transactions commit in virtual time order
  - Apparent order of transaction execution fully deterministic

Barrier Synchronization

- If P and Q are *parallel programs*, then let us write

  { P ; Q }

  where ; is a barrier synchronization operator.

- Semantically, barrier synchronization is the composition of (partial) functions. If F_P and F_Q are (partial) functions over system states corresponding to programs P and Q, then

  $$F_{\{P\ ;\ Q\}} = F_Q \circ F_P$$

- Operationally, barrier synchronization means
  - Execute P, starting from an initial input state
  - Execute Q, where the output state of P is the input state of Q
  - The output state of the whole computation is the output state of Q
  - Information is only transmitted from P to Q, but never the reverse direction

Conservative Implementation of Barrier Synchronization

- There must be a way of indicating which processes are involved in a particular barrier instance. In MPI that is a *communicator*.
- In each process we need a specific call to a barrier function, e.g. MPI_`barrier(communicator)`, at the exact point in the logic where the barrier occurs
- The conservative implementation of `{ P ; Q }` relies on process blocking

```
Start all parallel parts of P; if any part of P fails, abort
Each process, when it executes the barrier() operator, blocks
    until all parallel components of P to finish in real
    time(including any threads or processes created by P)
Start all parts of Q
```

Barrier Synchronization

- Blocking until all parts of `P` to finish *in real time* is not formally required
- All that is required is that it *appear* that way, though it is not obvious how else to do it
- Parts of Q can be started before parts of P finish, or even P and Q can be done out of order, as long as the formal definition of barrier synchronization is satisfied.
- Compilers routinely reorder statements across ; - boundaries all the time, as long as it makes no semantic difference.

Optimistic Virtual Time Synchronization

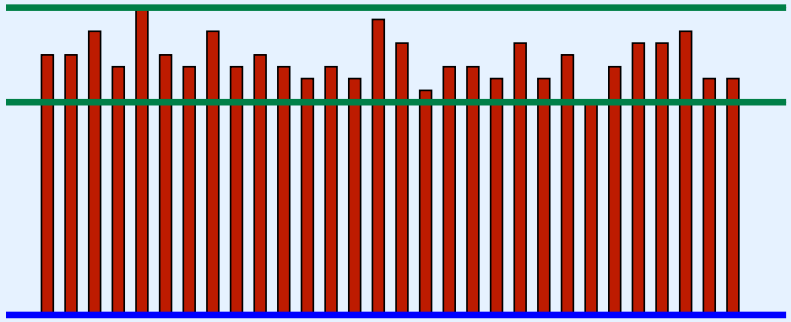
- Fundamental observation:

  With virtual time there is always a perfect global barrier between any two distinct virtual times!

- To create implement a barrier as in `{ P ; Q }`, just make sure that all events in `P` take place at lower virtual times than any of those in `Q`!
- Because the barrier is *global*, no construct like a communicator is required.
- Because we are using virtual time no specific call to any kind of `barrier()` programming primitive is required.
  - We have our temporal coordinate system to use instead of points in the sequential code.
  - We can name points in virtual time, and we can calculate them!

Performance comparison between conservative and optimistic virtual time barrier synchronization

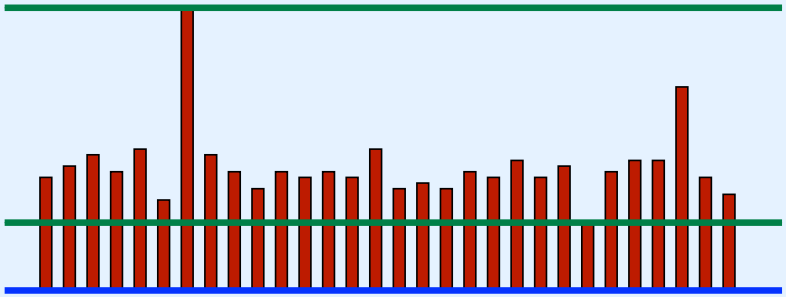
Good performance case for conservative barrier implementation



- Time between barriers across all processes relatively uniform
- High average parallelism between barriers
- Relatively short time real time interval between first arrival at barrier and last

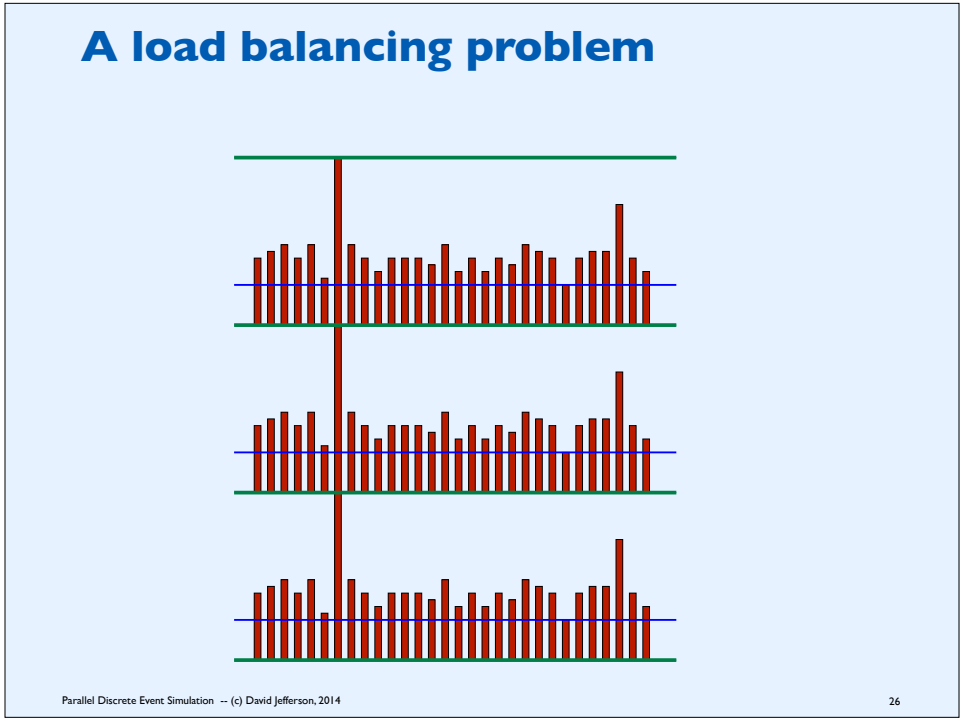
Time goes upward in this diagram. The blue line is the completion of the previous barrier synch. The green lines represent the beginning and end of the next barrier synch. The red bars indicate how much computation each processor does before it reaches the barrier call, after which it stays blocked until the last one reaches the barrier.

Bad performance case of conservative barrier synchronization



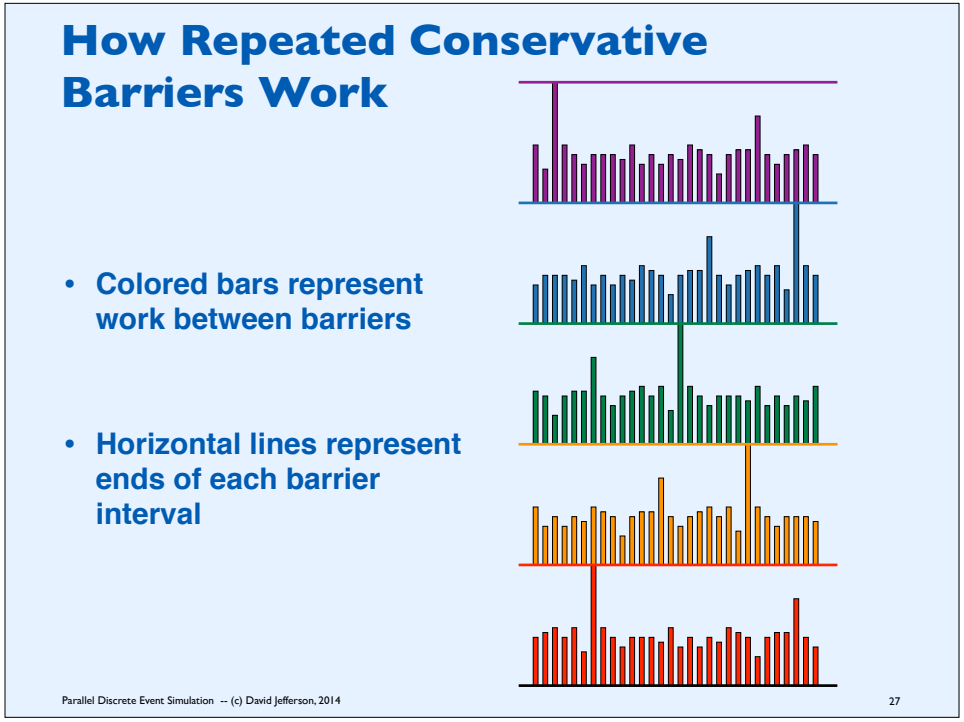
- Time between barriers across all processes highly non-uniform
- Low average parallelism between barriers
- Relatively long time real time interval between first arrival at barrier and last

Time goes upward in this diagram. The blue line is the completion of the previous barrier synch. The green lines represent the beginning and end of the next barrier synch. The red bars indicate how much computation each processor does before it reaches the barrier call, after which it stays blocked until the last one reaches the barrier.



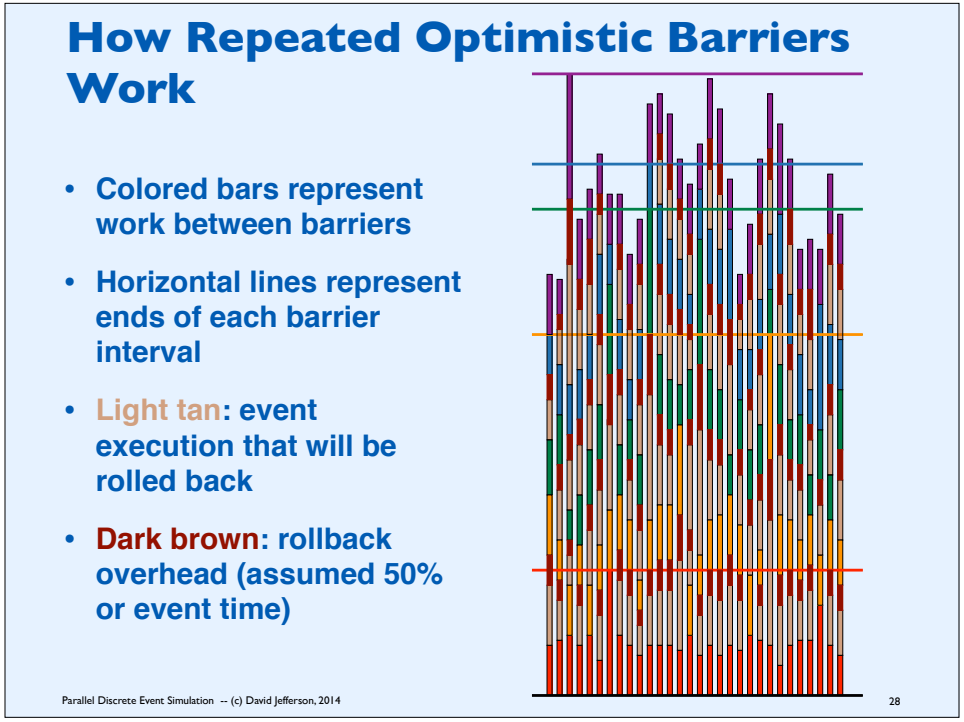
Time goes upward in this diagram. In this case the blue lines are the start of a barrier synchs, and the green lines represent the ends of barrier synchs. The red bars indicate how much computation each processor does before it reaches the barrier call, after which it stays blocked until the last one reaches the barrier.

In this case with the repeated barriers it turns out that the same processor is the bottleneck each time, taking the longest time to reach the barrier call. Neither conservative nor optimistic synchronization alone can solve this performance problem. It is essentially a load balancing problem, and the computation should be rearranged so this does not happen.



Here we have a diagram illustrating repeated barriers, but one that is more or less statistically load balanced, meaning that the bottleneck process moves around and is not always the same.

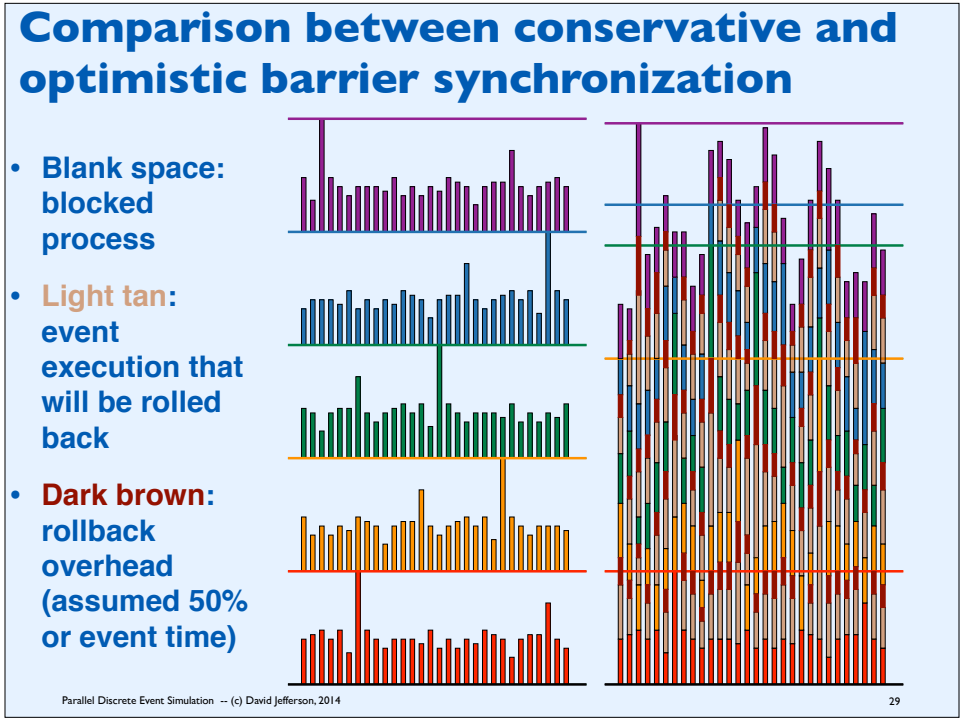
The synchronization is conservative, meaning that processes block at the barrier until all of the other processes reach the barrier. The time between barriers is determined by the time the last one reaches the barrier. Hence the vertical colored bars represent time when the processors are doing useful work, and the space above the vertical bars that is blank represents time that the processors are idle. In this example, more than half all processor time is idle, waiting for whatever bottleneck process is the last to finish during each barrier cycle.



This diagram is a little hard to explain. It is explains how optimistic synchronization would work on the same repeated barrier synchronization as shown in the previous slide. The red, orange, green, blue and purple vertical bars are the exact same ones, and the same length, and in the same order, as on the previous slide, representing computation that has to be done between the barriers. But because this is optimistic synchronization, we need to represent committed executions of the code (color) and rolled back executions of the code (light tan) as well as rollback overhead (dark brown). The time it takes to do a rollback is assumed to be 50% of the time it took to execute the event in the forward direction, so the dark brown bars are always half as long as the light tan bars just below them in this diagram.

Also to understand this diagram you must understand that each process is supposed to be communicating with its left and right neighbors using event messages. Any colored bar that is strictly higher than the same colored bar to both its left and right is the slowest of the three. It presumably has its communication already from its left and right neighbors by the time it finishes one round of computation, and can start the next round without delay. Thus, in the bottom row, the third process counting from the left finishes later than its two neighbors, and so it can start on the next round of computation immediately and will not need to rollback because of any late messages from its neighbors. Hence, the third process starts the yellow computation, part of the second round, immediately after finishing the red computation of the first round.

By contrast, its two neighbors are faster than at least one of their neighbors (in particular, the third process). So although the optimistically start their next round of computation, it is going to be rolled back when process #3 finishes and sends them a (straggler) message. However, the straggler message does not interrupt the execution of the event that was going on — we do not presume preemption, so the event continues until it completes before the rollback starts, Hence, each light tan bar, representing a rolled back event, is the same length as the colored bar above it, which represents the execution of the same event which gets committed rather then rolled back. In between the light tan bar and the colored bar above it is a dark brown bar that is 50% as long, which represents the overhead of rollback.



This slide shows the diagrams in the previous two slides side-by side. You can see that the conservative barrier execution has a lot of time when the processors are idle, waiting at the barrier for the last process to reach it before starting the next round of computation. But the optimistic barrier synchronization has no idleness. It is always executing (at least until the last round is finished). The execution is in most cases speculative and ends up getting rolled back. But even so, the five rounds of the computation finish slightly sooner that with the conservative synchronization.

A key parameter for the repeated barrier computation is the ratio of the worst case execution time of any process in a round to the average case execution time among the processes in a round. If that ratio is high, then conservative execution performs very poorly and optimistic execution often wins. If that ratio is low, however, then the overhead of optimistic synchronization often causes it to perform worse than conservative synchronization.

Fault Recovery

Detecting and correcting transient computational faults

- Today we have various mechanisms for detecting and correcting *memory* errors and *data transmission* errors
  - Generally variations on parity, checksums, and ECCs.
- But we have no general redundancy mechanisms in place for even *detecting* outright *computational errors*
  - The only possibility is duplication of the computation and comparison
  - But that only works if the computation is deterministic
- And even if we *detect* them, we currently have no way of *correcting* them except by global restoration of a global checkpoint

Reverse computation cannot reliably restore a previous state in the presence of a fault

- What is supposed to happen

$$\langle S_1 \rangle \rightarrow E^+() \rightarrow \langle S_2 \rangle \rightarrow E^-() \rightarrow \langle S_1 \rangle$$

- What happens if there is a fault in forward computation

$$\langle S_1 \rangle \rightarrow E^+() \rightarrow \langle S_2 \rangle \rightarrow E^-() \rightarrow \langle S_1 \rangle$$

- What happens if there is a fault in reverse computation

$$\langle S_1 \rangle \rightarrow E^+() \rightarrow \langle S_2 \rangle \rightarrow E^-() \rightarrow \langle S_1 \rangle$$

- The story is no better if the fault occurs in the runtime system or OS

Notice that if we want to use rollback to recover from faults, we cannot use reverse computation to accomplish it. With reverse computation, if we start in state **S1** and execute $E^+()$ correctly then we get to **S2**, and if we then execute $E^-()$ correctly we get back to **S1**.

But if a fault happens during execution of $E^+()$, and we do not get to **S2**, but instead to faulty state, then executing $E^-()$ correctly does not necessarily get us back to **S1** as it is supposed to, but likely to another faulty state.

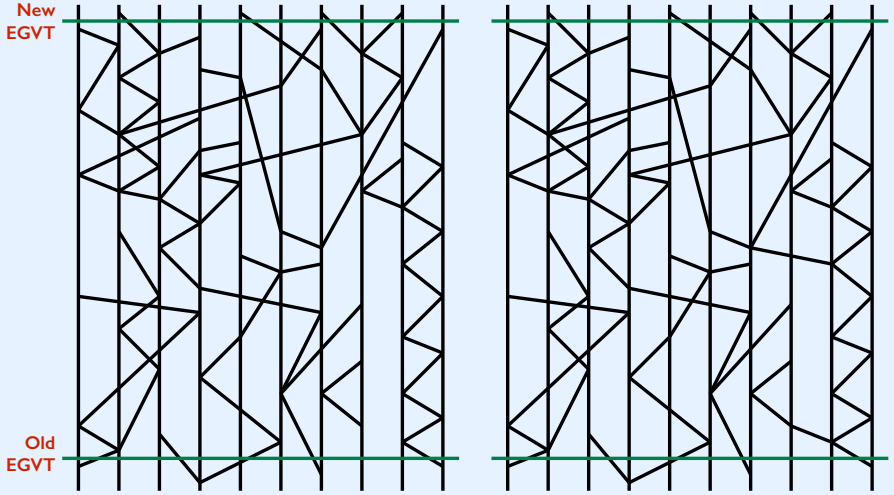
A similar problem arises if the fault occurs not during $E^+()$, but during $E^-()$. Again, starting from correct state **S2**, we do not get back to correct state **S1** as we are supposed to, but to a faulty state.

Can we use rollback to recover from faults?

- We would probably not consider it except that we are have a rollback mechanism in place for synchronization anyway!
- But with virtual time transient computational faults are *correctable* without global restoration of a global checkpoint
- However
  - We must duplicate the entire computation just to detect computational errors
  - We must use state saving, not reverse computation, for rollback
  - We must check for errors at commit time by comparing states and messages in the duplicate computations
- Corrections can be done
  - semi-locally
  - asynchronously
  - in parallel with the rest of the computation
  - no (conservative) barrier required

Run entire computations in duplicate

- Obviously requires twice the RAM right off the top, and twice the cores if there is to be no speed sacrifice.

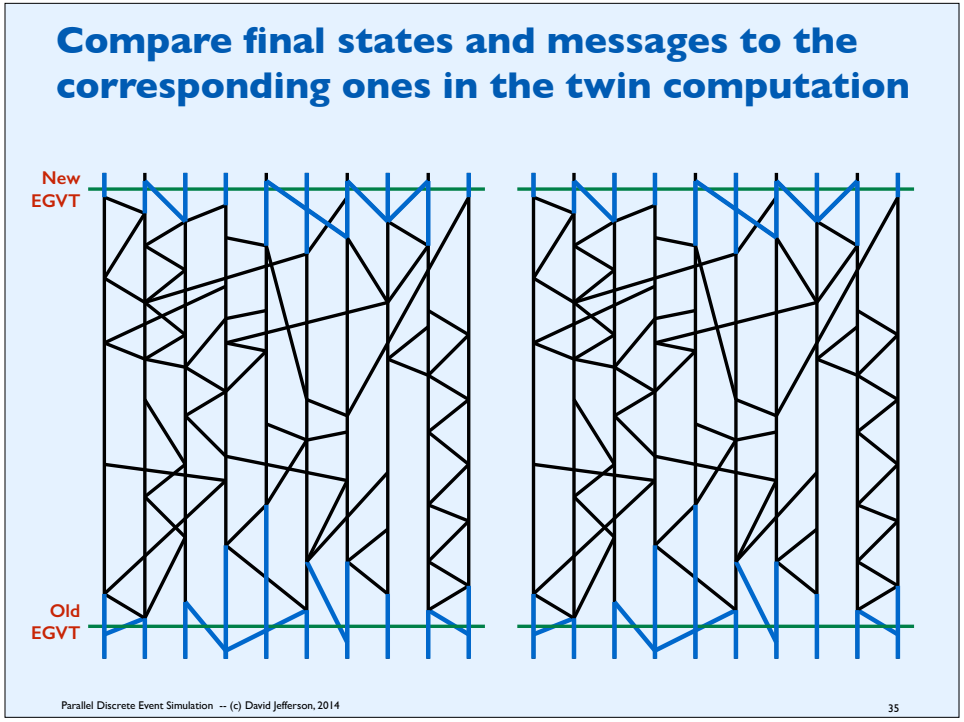


Because this is a deterministic virtual time computation, both of the space-time graphs will look identical.

The lower green line is the virtual time at which the last full state saves occurred, so we can roll back to those states without doing reverse computation. The states and messages crossing that line were validated by comparing them to their twins in the other computation.

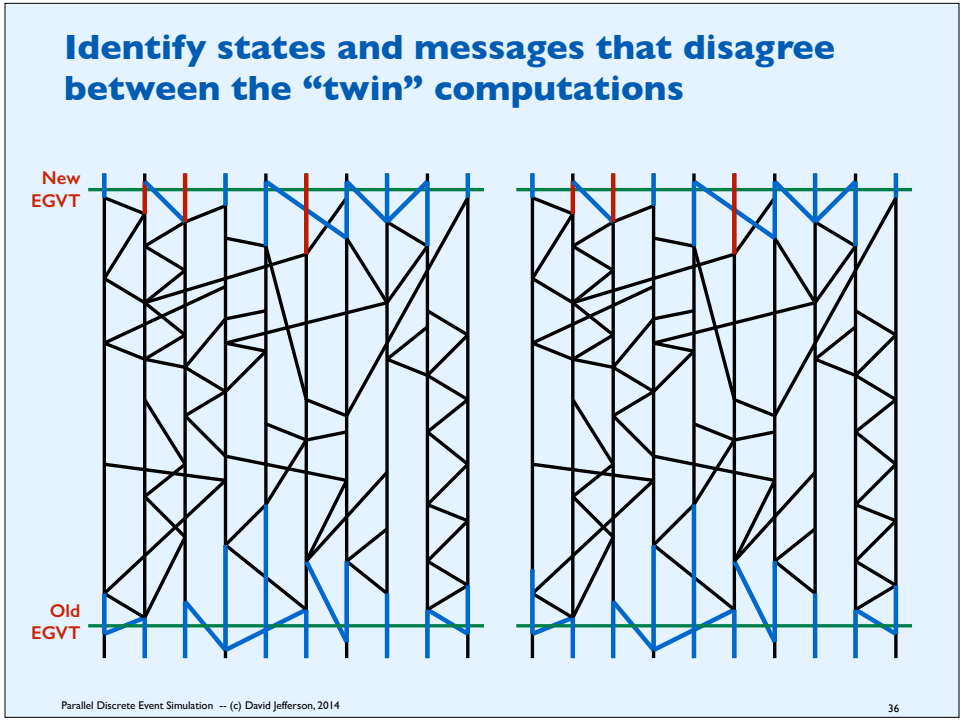
The upper green line represents the newly calculated EGVT. At this point we are about to commit to this new EGVT and discard all older state and message information. But first, as part of commitment, we have to make sure it is correct, i.e. that the new states and messages we are going to save and from which we cannot roll back, have not been damaged by a fault of system bug.

(Slight additional complications in the algorithms on these next slides arise if objects are created or destroyed, but we will ignore them.)



The blue segments represent the states (vertical) and messages (horizontal) that will be saved after commitment. All else will be discarded.

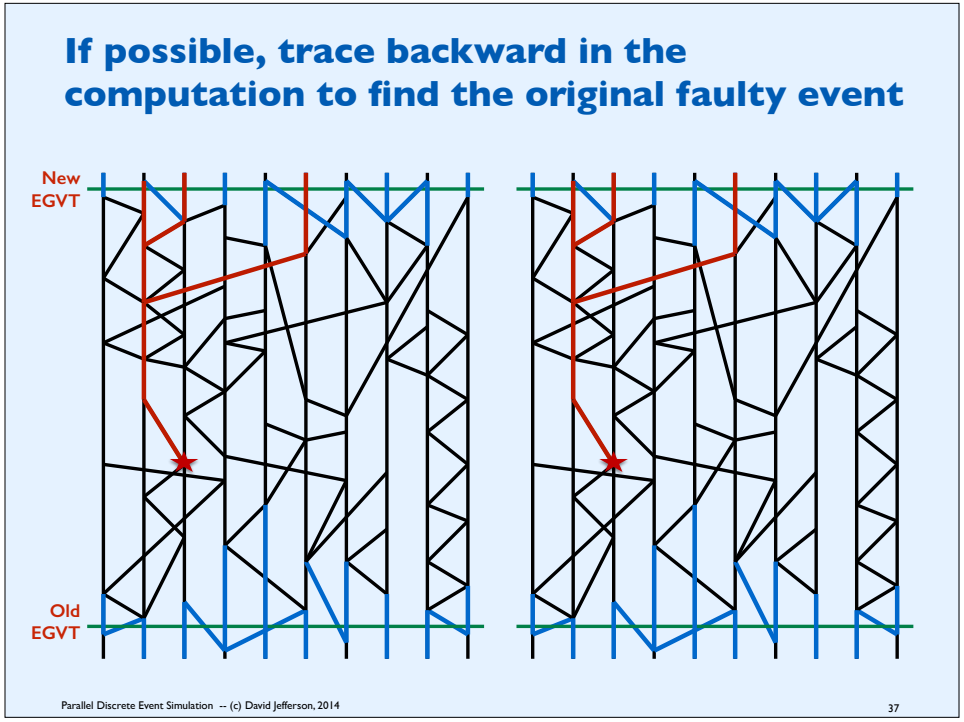
The first step is to compare these states and messages from one computation to those of the other twin computation. If they are all identical, then the commitment can continue, and all but the blue states and messages can be discarded.



However we may find discrepancies in some of the states or messages that we compare. In this case the red lines represent states that disagree with the corresponding ones in the other duplicated computation.

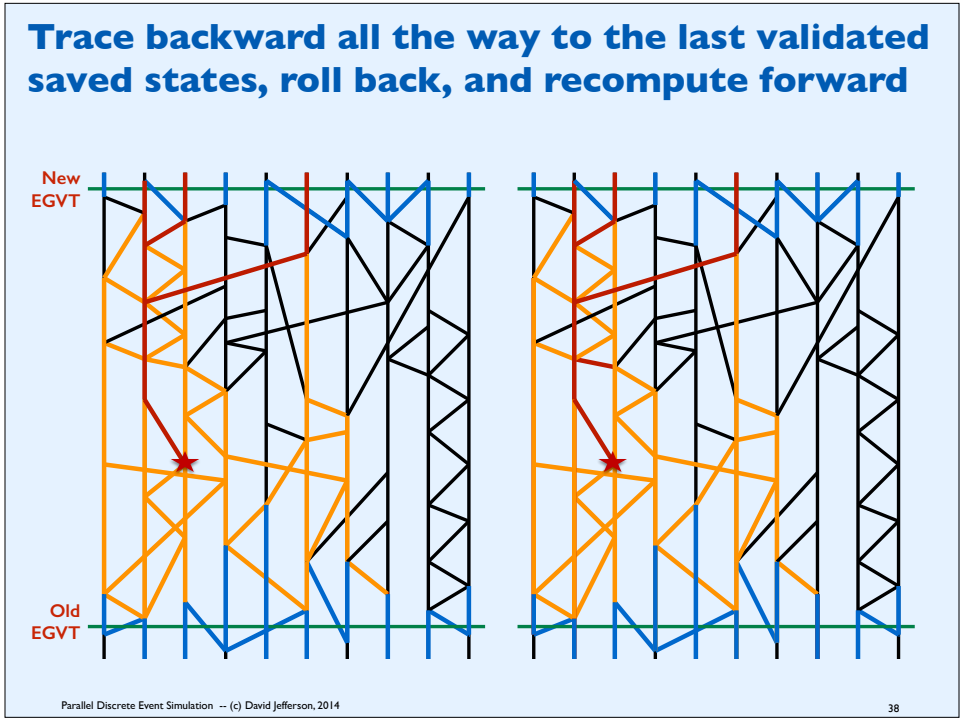
We don’t know at this point which ones are correct and which are wrong — indeed in principle they could all be wrong, though that should be extraordinarily unlikely. We also don’t know if these discrepancies represent the results of multiple faults, or just the spreading pollution of one original fault. If they are multiple independent original faults, some of the faults could have occurred in one computation and others could be in the twin.

Regardless, the following procedure is unchanged.



The original faulty event(s) have inputs that agree in the twin computations, but outputs that do not. If the rollback mechanism uses full state saves for every event or full saved state deltas of some kind (e.g. dirty pages), then we can trace backward in both computations, comparing comparable states and messages between the two, to discover the original faulty event. The original faulty event would be an event in which all of the inputs to the event (states and messages) agree between the two computations, but the outputs differ. Having found that event, we know that it either executed wrong in one computation or in the other (or extremely rarely, both), but we don't know which. The **red** arcs are where the corresponding states or messages in the twin computations disagree. The starred event is the one that had the original failure in one or the other computation (but we don't know which one).

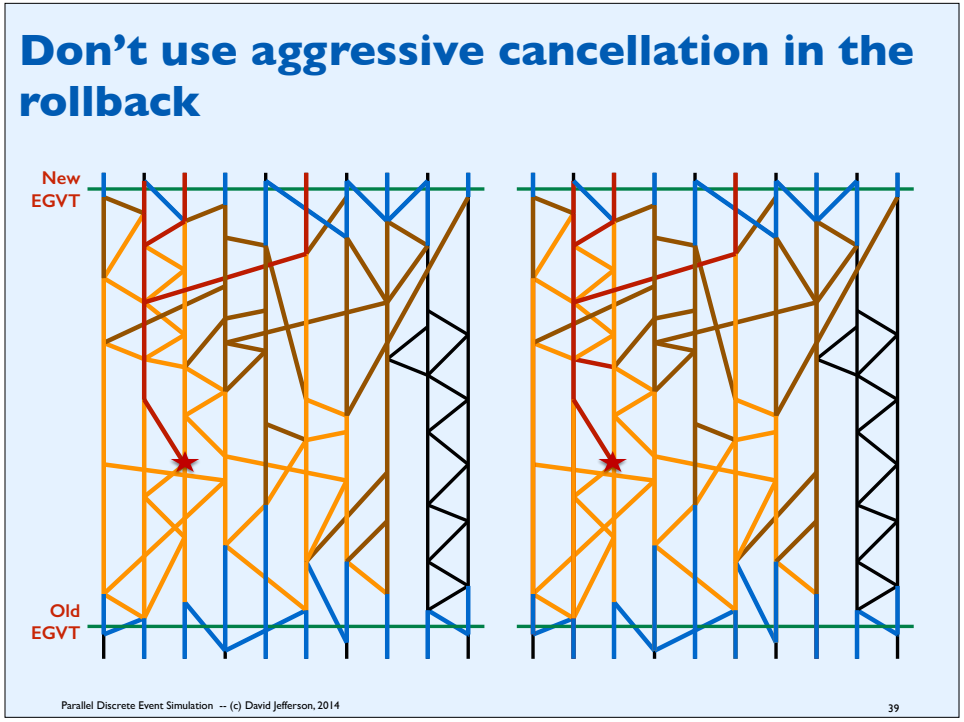
If there is a way to reconstruct this state reliably in both computations (e.g. because we are doing full up front state saving (snapshotting) between every two events) then we could just restore those states to the object in question in both computations and let them execute forward again from there using lazy cancellation. Both computations will re-compute the red tree of incorrect or potentially incorrect messages and states, and if the final states and messages they compute at EGVT agree this time, then the problem has been perfectly corrected. If not, then there is either a bug in the runtime system/OS, or there is a permanent fault (or a second transient fault — presumably so rare that it is negligible), and we should then abort.



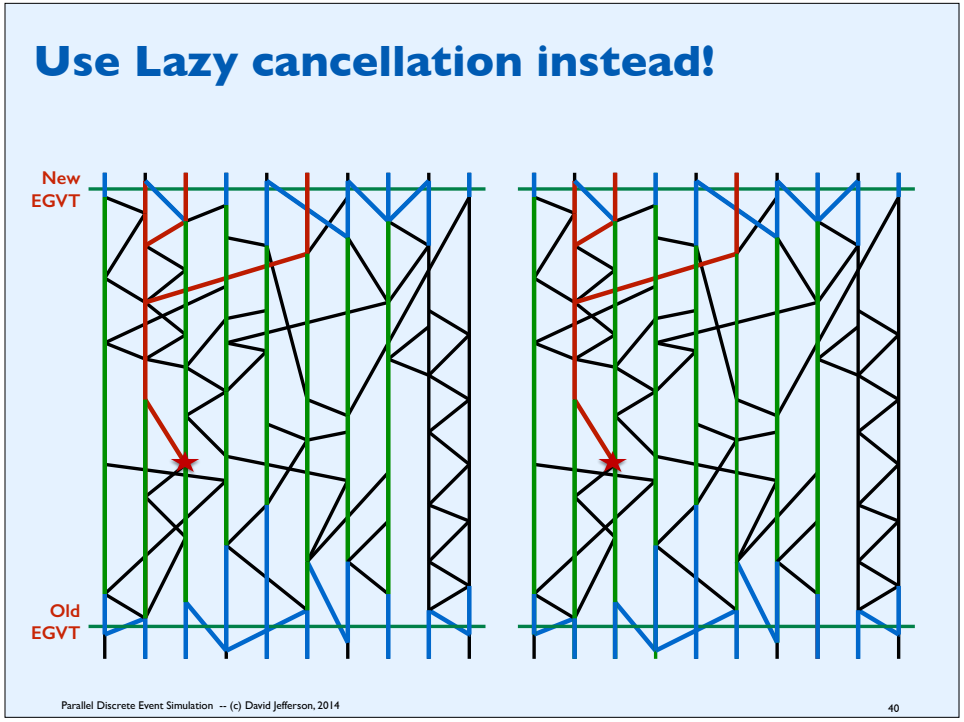
Trace back from the states and messages that disagree between the two computations at the time of the New EGVT back to the last validated snapshot to identify all states and messages that have causal paths to the known good states and messages at Old EGVT. In this diagram all of the **orange** and **red** events, states, and messages are suspect, and the fault is somewhere included among them. That is the portion of the computation that has to be re-done. Even if the fault was in the OS or runtime system, as long as it was transient, redoing the red computation will correct it. (But it will not necessarily correct the effects of *bugs* in the OS or runtime system.)

To correct the faults we roll back all of the objects in both computations that lead to suspect events. In this case it is the leftmost 8 objects in both computations that must roll back. When we roll back the 8 leftmost objects, we proceed to re-execute forward using *lazy cancellation*. That prevents us from resending messages that are the same as were generated the last time the event was executed before the rollback, and from re-doing more of the computation than is necessary to assure that the fault is corrected.

However, since we are not doing full state saves after every event, we have to trace backward from the bad outputs of the computation all the way back to the known good inputs. Any message, state, or event on a path from the known good inputs to one of the bad outputs is suspicious. In this diagram that includes both the red and orange arcs and nodes.



We must roll back all processes that are suspicious to a known good state, and start re-executing forward. As we re-execute forward we can use aggressive cancellation, lazy cancellation, or some other variation. Aggressive cancellation, however, is a poor choice because it leads to cancellation of many more messages than necessary, and hence much more re-computation than necessary. In this diagram the brown arcs and computation are not suspicious, but aggressive cancellation would cancel all of those messages anyway, and then the forward execution would regenerate and re-send them all. It would work, but is inefficient.



When we roll back the leftmost 8 objects (the suspicious ones) to the known good states and message queues and then re-execute forward with lazy cancellation, we end up canceling and re-executing a lot less computation than with aggressive cancellation. In this diagram the **red** and **green** arcs are those that have to be re-generated and re-transmitted. The **red** arcs were actually incorrect in at least one of the twin computations and of course they end up being recalculated. The vertical **green** arcs represent events and states that get re-executed just because we don't know that they are correct until we re-execute them all the way forward to final states at time New EGVT. But with lazy cancellation the events on the **green** paths recalculate the messages sent from those events, and because the outgoing messages from those events were correct the first time and are then regenerated the same as the first time the event was executed, there is no need to cancel them or resend them those messages -- that is the way lazy cancellation works. Thus, there are no **green** message arcs in this diagram, just **green** state arcs. With a little more logic, some of the **green** states would not have to be re-calculated either, because once we recalculate and event all of whose outgoing messages are already correct, if that happens in an object whose final state is known correct, then all of the intermediate states can be presumed to be so also.

End