Parallel Discrete Event Simulation Course #15

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-654663-DRAFT

Last 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!

Remember that none of the ideas in this week's lecture or last have been implemented and tried out in real applications. Lectures 14 and 15 are speculative in nature and are my personal best guesses of where virtual time technology might be useful.

Virtual Time for Most Large Scale Computations

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!**

Determinism

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
- Require deterministic assignment of virtual time values
- Then – It is impossible to write a nondeterministic virtual time program!
- This is true for both conservative and optimistic implementations.

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 generally can't be without duplication somewhere.

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

But as we will see, transient faults can be detected and corrected without restoring global checkpoints using virtual time.

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 10

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.

The general way to think about virtual time synchronization is this: define a synchronization constraint in terms of *real time* values. Then redefine it substituting *virtual time* values. The results is another synchronization primitive that is nonblocking and optimistic and in many cases can perform better than the real-time (conservative) primitive.

Mutual exclusion

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

The **real time** definition of mutual exclusion would go something like this: **P** and **Q** are mutually exclusive if the **real time** interval during which **P** executes does not overlap with the **real time** interval in which **Q** executes.

Or, equivalently, **P** and **Q** are mutually exclusive if one of them finishes executing at a **real time** moment earlier than the **real time** moment at which the other one starts.

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

The ~~real time~~ **virtual time** definition of mutual exclusion would go something like this: **P** and **Q** are mutually exclusive if the ~~real time~~ **virtual time** interval during which **P** executes does not overlap with the ~~real time~~ **virtual time** interval in which **Q** executes.

Or, **P** and **Q** are mutually exclusive if one of them finishes executing at a ~~real time~~ **virtual time** moment earlier than the ~~real time~~ **virtual time** moment at which the other one starts.

Atomic transactions

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

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
```

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 19

The **real time** definition of barrier synchronization is this: there is a barrier between parallel activities **P** and **Q** if **P** finishes at a **real time** moment earlier than the **real time** moment when **Q** starts.

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.

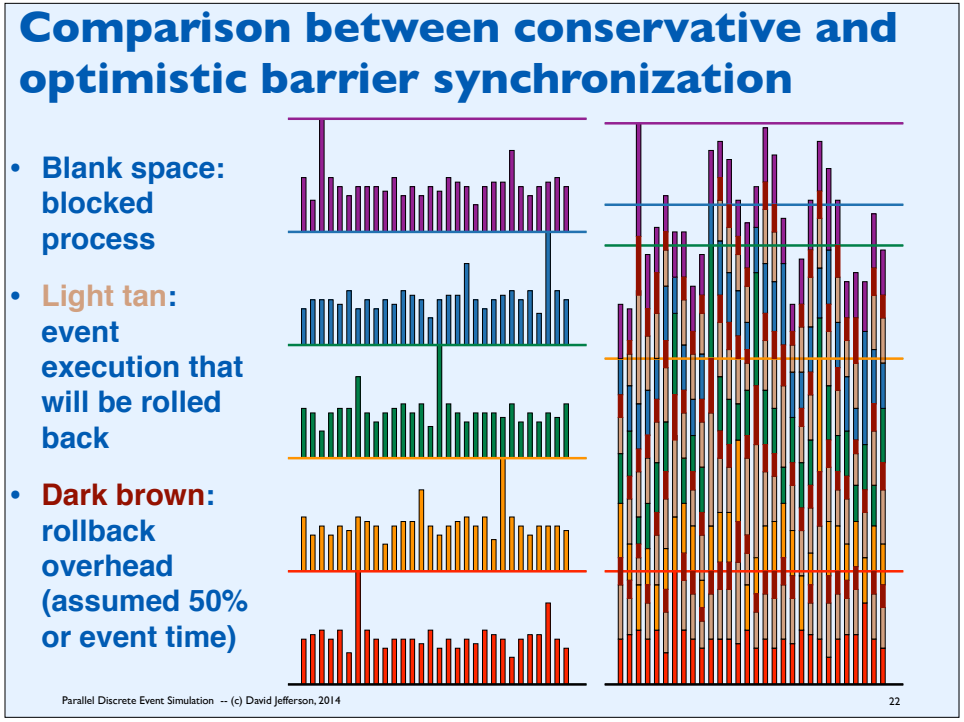
Optimistic Virtual Time Synchronization

- **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!**

The ~~real time~~ **virtual time** definition of barrier synchronization is this: there is a barrier between parallel activities P and Q if P finishes at ~~real time~~ **virtual time** moment earlier than the virtual time momentwhen Q starts.



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

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

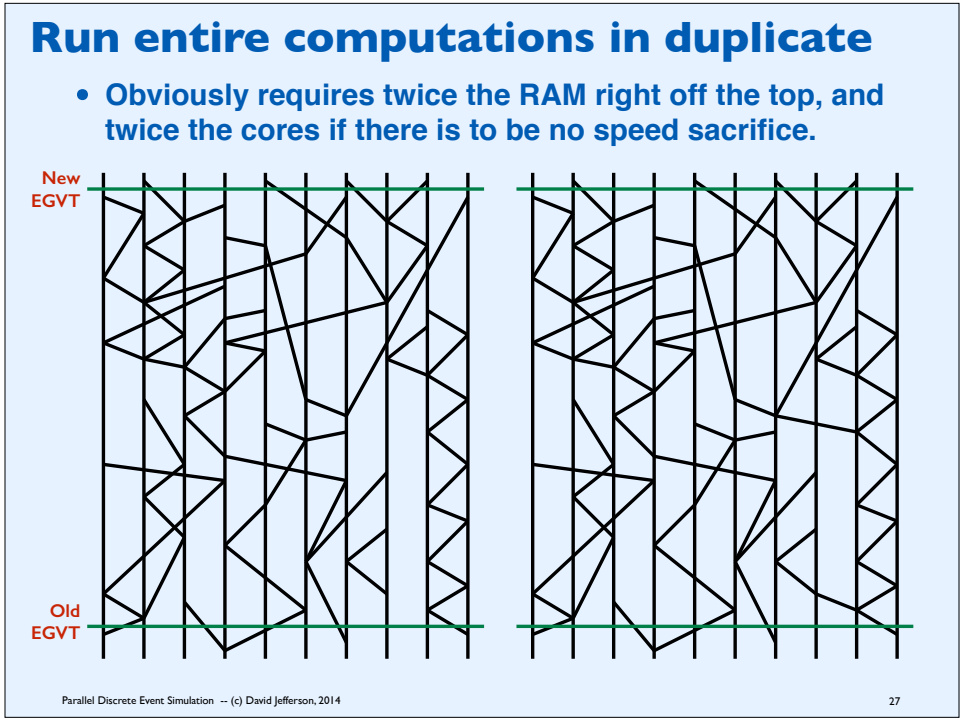
Notice that if we want to use rollback to recover from faults, we cannot use reverse computation to accomplish it. With reverse computation, if we start in state **S1** and execute $E^+()$ correctly then we get to **S2**, and if we then execute $E^-()$ correctly we get back to **S1**.

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

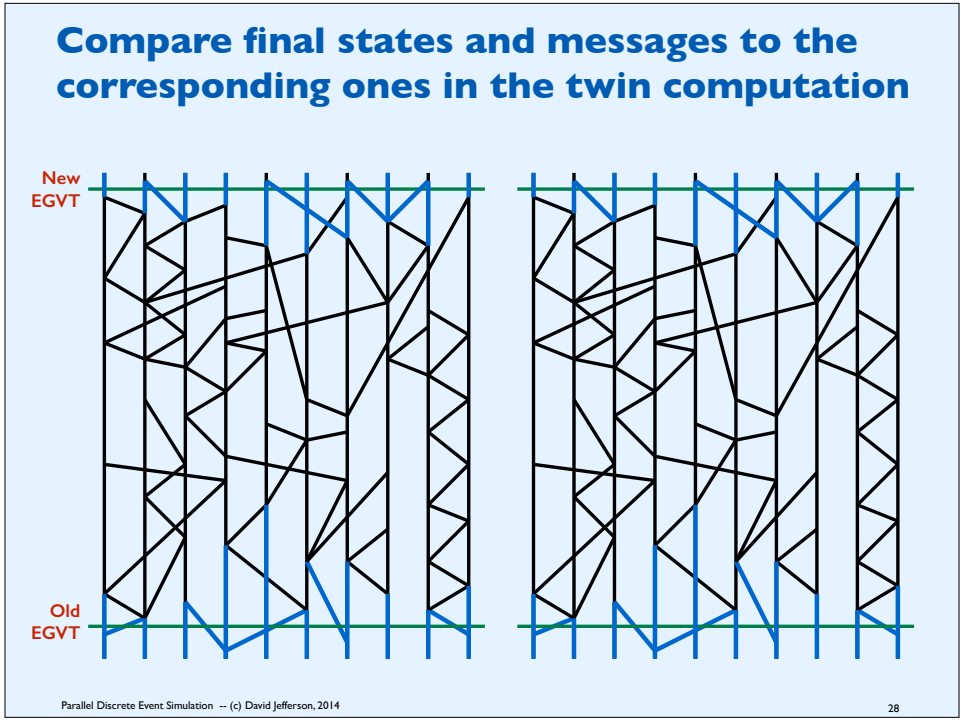


Because this is a deterministic virtual time computation, both of the space-time graphs will look identical.

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.

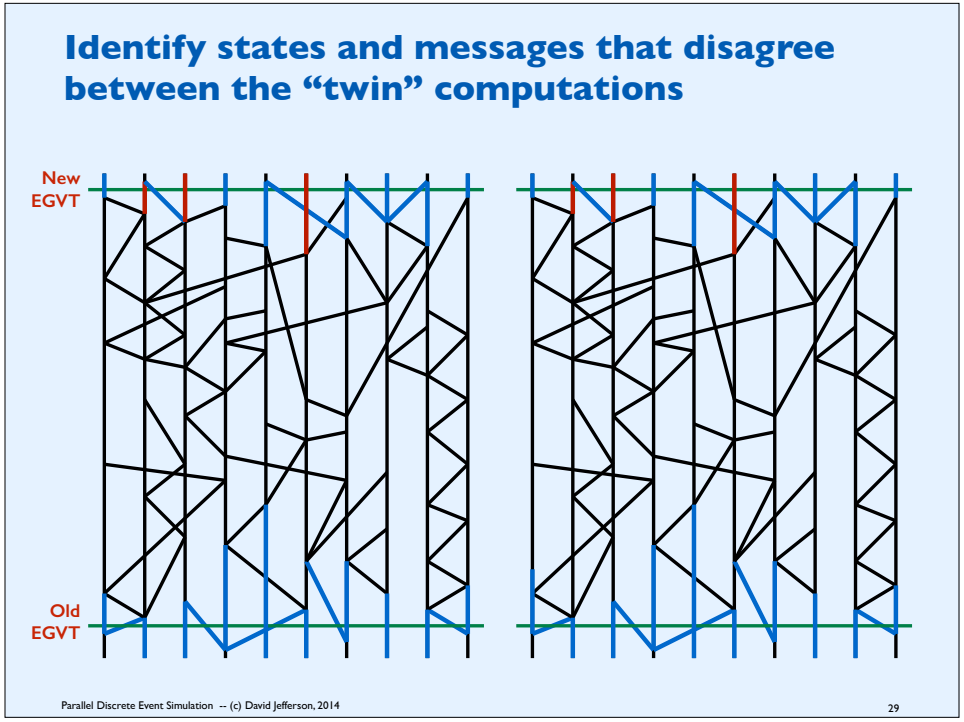
The upper green line represents the newly calculated EGVT. At this point we are about to commit to this new EGVT and discard all older state and message information. But first, as part of commitment, we have to make sure it is correct, i.e. that the new states and messages we are going to save and from 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.

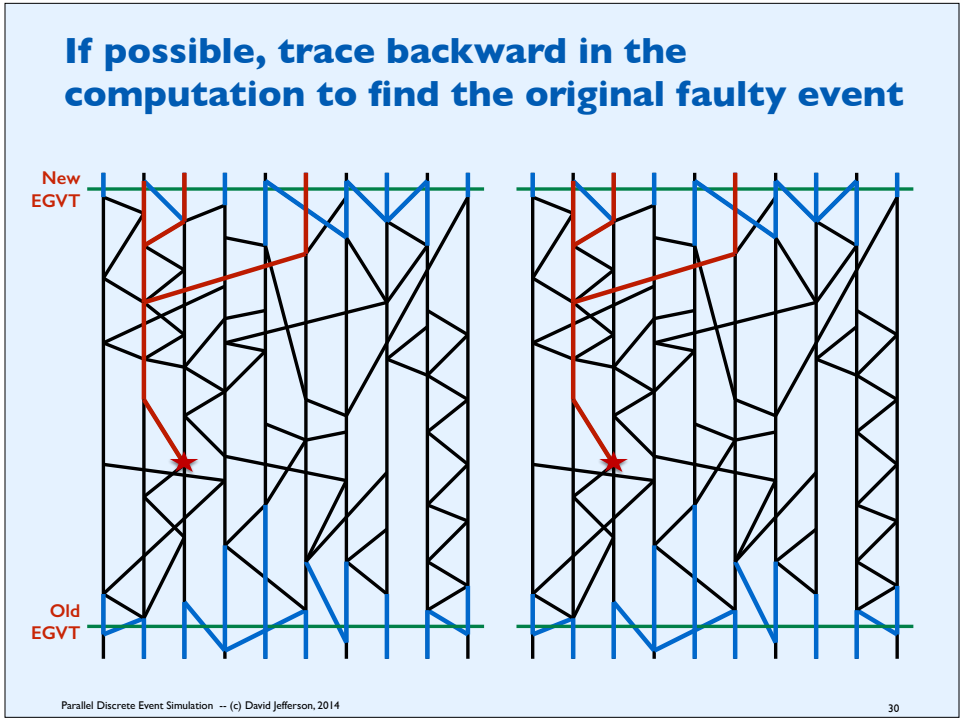
The first step is to compare these states and messages from one computation to those of the other twin computation. If they are all identical, then the commitment can continue, and all but the blue states and messages can be discarded.



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.

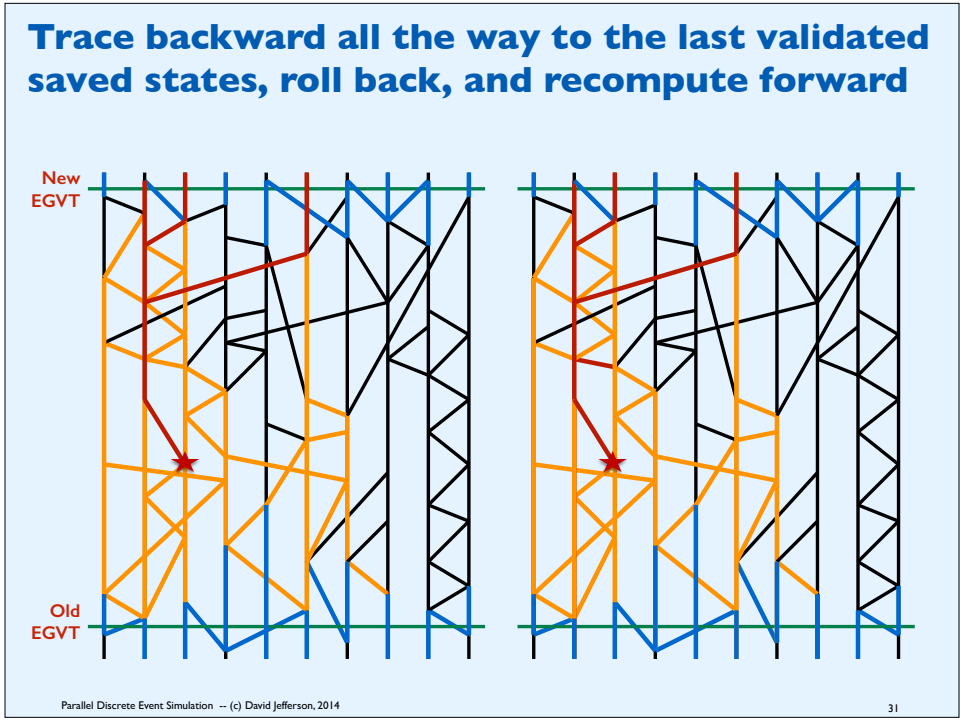
We don’t know at this point which ones are correct and which are wrong — indeed in principle they could all be wrong, though that should be extraordinarily unlikely. We also don’t know if these discrepancies represent the results of multiple faults, or just the spreading pollution of one original fault.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

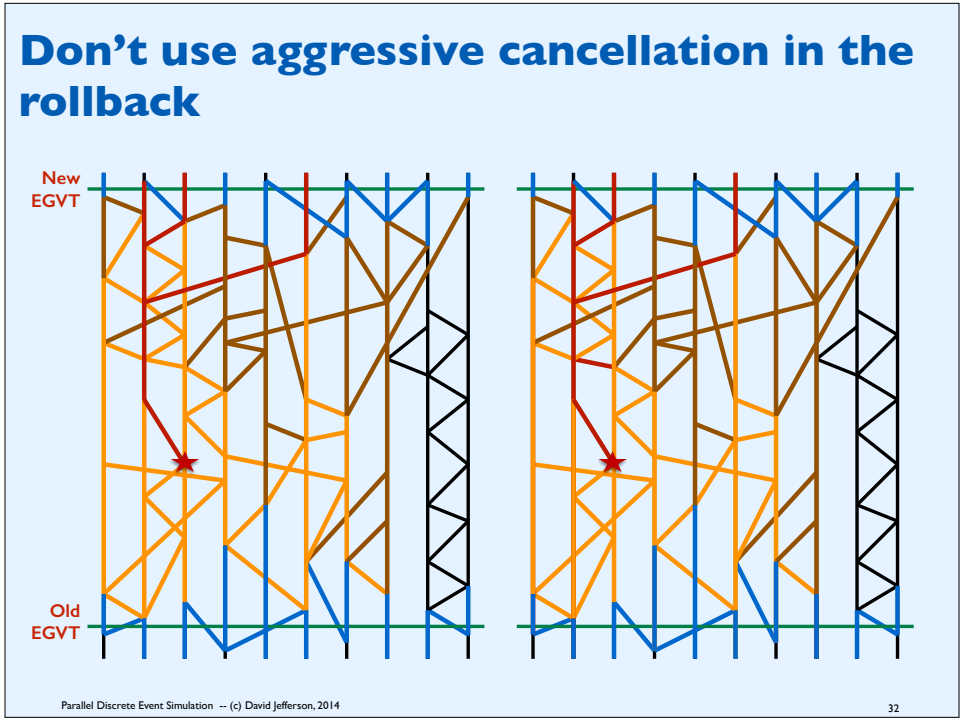
If there is a way to reconstruct this state reliably in both computations (e.g. because we are doing full up front state saving (snapshotting) between every two events) then we could just restore those states to the object in question in both computations and let them execute forward again from there 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.



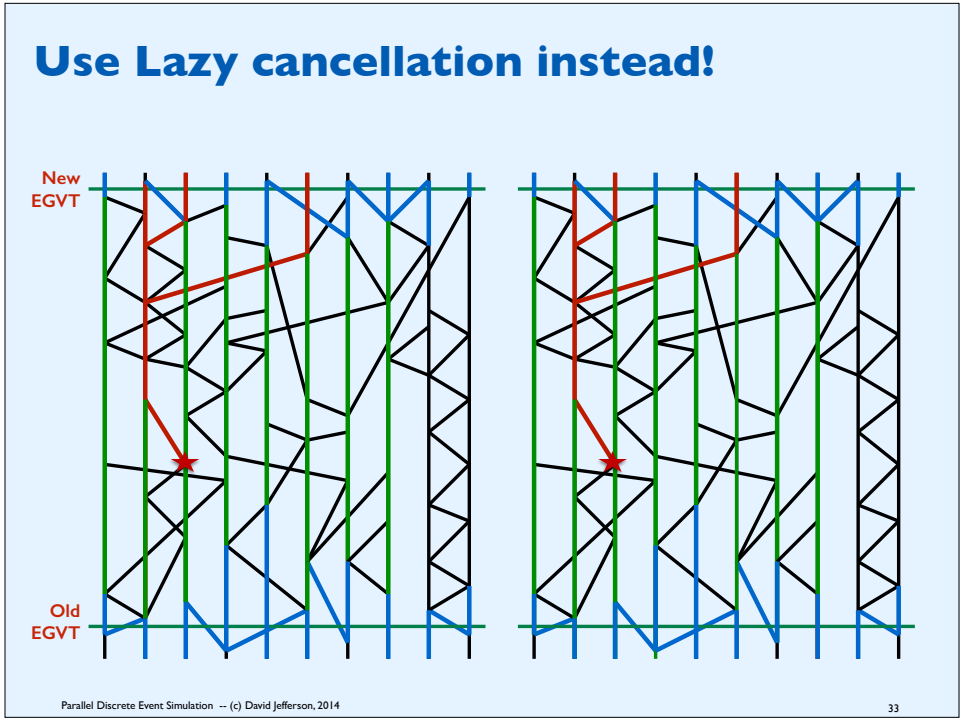
Trace back from the states and messages that disagree between the two computations at the time of the New EGVT back to the last validated snapshot to identify all states and messages that have causal paths to the known good states and messages at Old EGVT. In this diagram all of the **orange** and **red** 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.

Dynamic Configuration Management

Dynamic Configuration Management — DCM

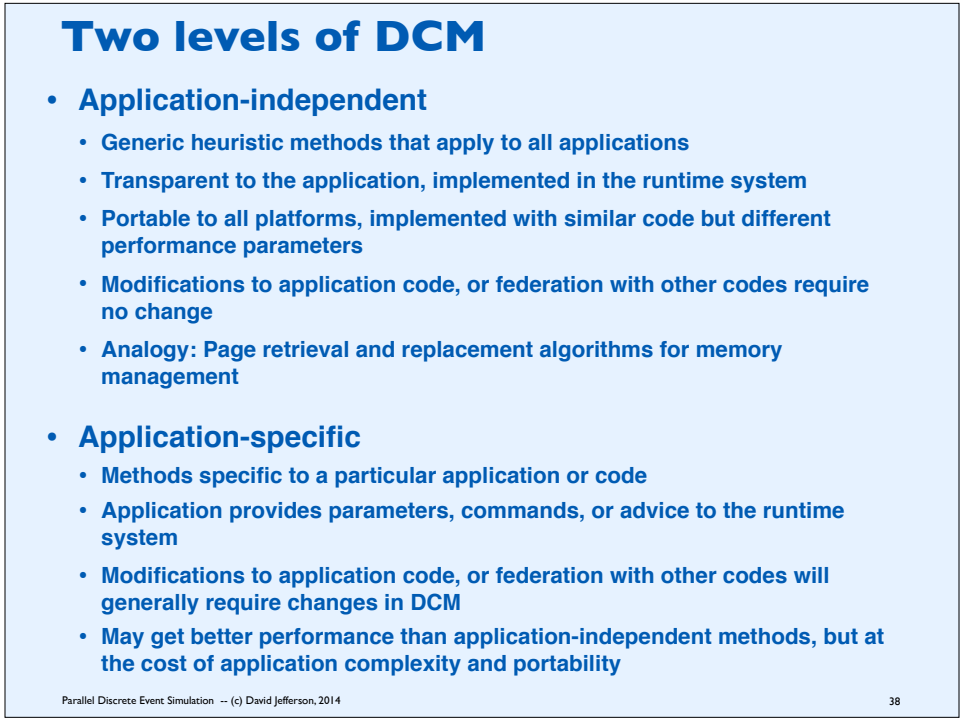
- Dynamic load balancing —> *dynamic configuration management*
  - includes dynamic processor (core) load balancing, balancing communication channel loads, and reducing communication latencies
  - The general view is to dynamically reconfigure the computation so that progress on the *critical path* is fastest
- In an SPMD application the natural technique for managing load is moving element / region boundaries, and splitting of joining of elements, and migrating *data*.
- But in an MPMD application like PDES, that does not work

DCM by object migration

- MPMD applications must be dynamically managed by migrating processes / objects from one platform node to another
- Overdecompose the application into small, concurrent, migratable units
- Instrument to estimate what parts of the computation and communication are on the critical path
- Migrate objects in such a way to speed execution of the critical path
- Adjust message routing maps dynamically so that messages in transit and subsequent messages are correctly routed to the new location of the migrated object

Ideal method of DCM

- Imagine an oracle that can instantly compute metrics for the future of the computation, with all arcs and nodes labels with real execution and message transmission times
- Have the oracle calculate the lengths of all future paths from the current states of all objects to the end of the computation.
- Schedule all event execution so that if there are `n` cores available, then *always* the lowest virtual time events on the `n` longest (critical) paths are being executed.
  - every microsecond that we are not making progress on the critical path(s) is a microsecond of delay in the final completion time.
- Propose object migrations that make longest near-term critical paths shorter:
  - If computation is the bottleneck make sure that long paths do not compete for cores
  - If some processors are faster than others, migrate critical objects to the faster ones
  - If communication bandwidth is the bottleneck, make sure that long paths do not compete for communication links
  - If communication latency is the bottleneck, make sure that objects that communicate *bidirectionally* on the critical path are close to one another, preferably on the same node.



Application-independent methods of DCM make one fundamental assumption: that the recent past dynamic behavior of an application is best predictor of the near future behavior. When that assumption is not true, as when the computation goes through frequent phase changes, then it is necessary to fall back on application-specific methods if they are available.

The general question for which we need new research is: To what extent can we make do with application-independent methods alone, so that they can be packaged in the runtime system and application programmers don't have to think about DCM? Demand paging has been an extraordinarily successful method of managing virtual memory, to the point tat there are very few circumstance in which an application programmer bothers to think about virtual memory management. The classic cases when it ***is*** necessary are when deciding how to walk the elements of a two dimensional array, and when writing memory management libraries (e.g. for **`malloc()`** and **`free()`**). Can we design application independent DCM algorithms that are as successful as demand paging algorithms?

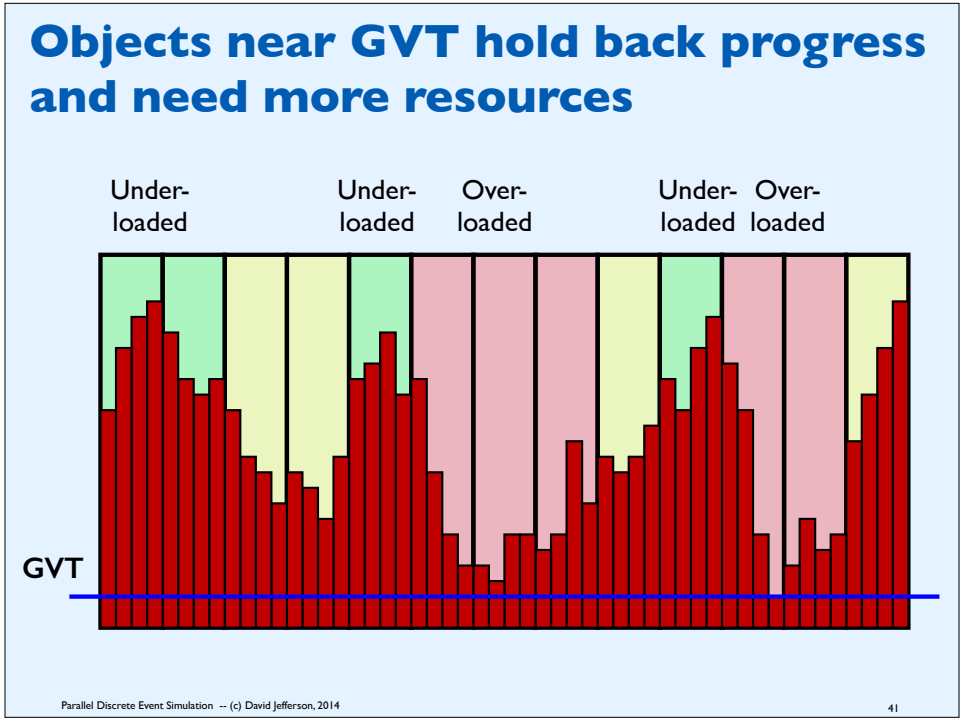
Virtual Time is useful for Application Independent DCM

- Dynamic load balancing has traditionally been driven by estimates of the "load" on the system processors and attempting to dynamically "balance" it.
- But that is not theoretically the correct approach, even for compute-bound executions, except in special cases
- DCM should ideally be driven by estimates of the future critical path.
  - Computation and communication *on the future critical path* should get more resources or lower latency
  - … at the expense of those off the critical path
- Current measure of processor or core loads tells you nothing about the future critical path
- Virtual time does!

Virtual time is an estimator of where the critical paths are

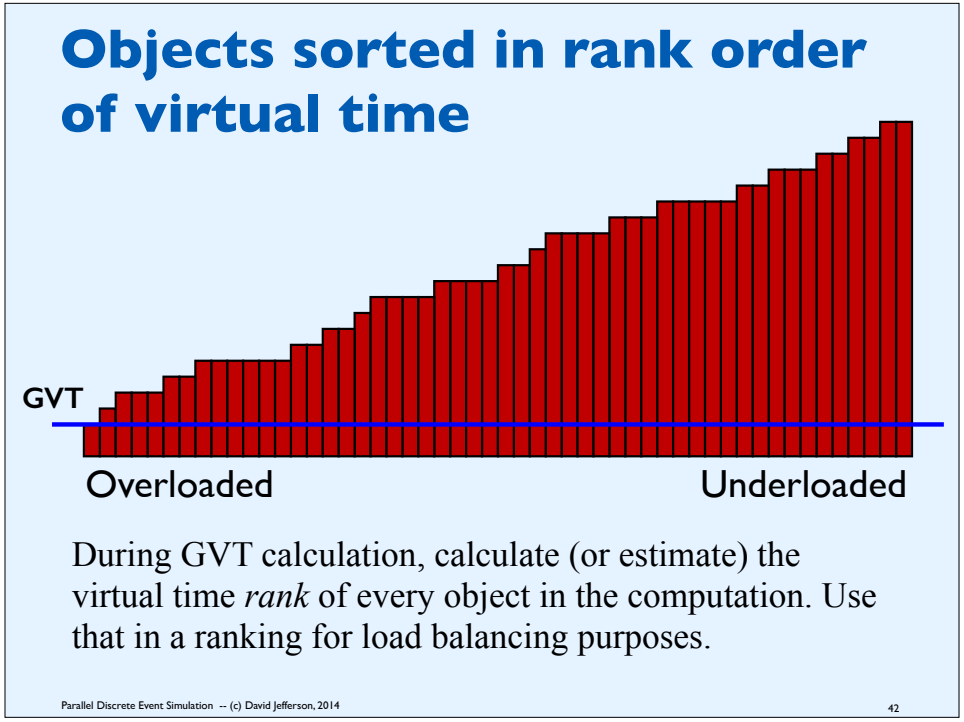
- At any given moment, virtual time is an estimator of what is on the critical path
  - lowest virtual times represents bottleneck objects — likely to be executing on the critical path
  - high (or ∞) virtual times are not a bottleneck, and likely to be doing speculative computation — unlikely to be advancing a critical path
- Rank order in virtual time indicates where more resources are needed, and where they should be taken from
- "Velocity" in virtual time as another useful statistic
  - Ratio of committed progress in virtual time Δv to committed progress in real time Δt

$$\Delta v/\Delta t$$



The red vertical bars represent how far in virtual time the objects in the simulation have progressed as of the time a new value of GVT has been calculated. Notice that GVT is defined by the lowest virtual time to which any object has progressed, whereas most of the objects are ahead of GVT, and some far ahead. (This description is not strictly true, of course, since messages in transit can also determine GVT and we are not considering that here. Also not considered is that not all of the objects in a computation will be polled for the contribution to GVT or informed of the new GVT value at exactly the same time, and so this diagram, which purports to be an instantaneous snapshot of the progress of all of the objects, really cannot be. But I am ignoring those points.)

The background regions above the groups of bars (representing groups of four objects each located in the same platform node) are colored pink, yellow or green according to whether the farthest behind object on that node is far behind in virtual time, among the midrange, or far ahead in virtual time. Those platform nodes that are far ahead (green) can be considered to be underloaded in comparison with the others. Those that are far back (pink) can be considered to be overloaded. This suggests that a reasonable strategy for ordinary load balancing would be to migrate load from pink to green nodes.

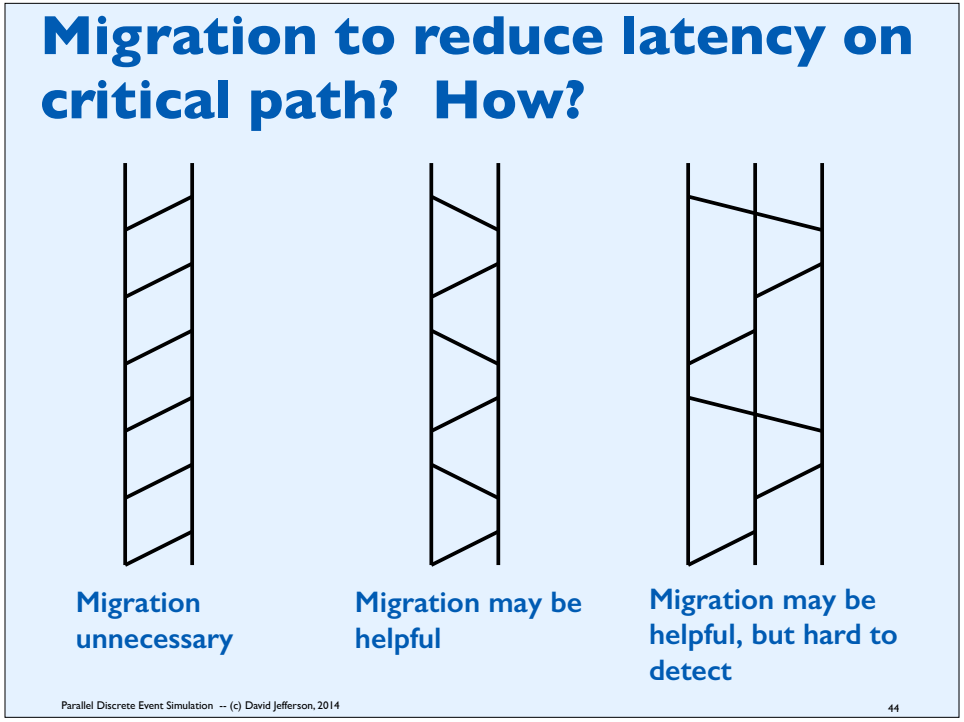


In this slide we have taken the data (red bars) from the previous slide and arrayed it in sorted order

In order to apply the strategy outlined on the previous slide, it is necessary for each node to be informed, at the time a new EGVT value is promulgated, of its *rank* in virtual time compared to all other nodes. That way a node knows whether it is overloaded or underloaded with respect to other node. The nodes armed with that information may collectively apply algorithms and heuristics to decide which objects are to be migrated where. This rank information does not fully determine the load balancing algorithm, but it definitely helps inform it.

Energy management as a potential tool for “load balancing"

- Assumptions
  - We want to power as little of the hardware circuitry as possible at all times
  - The runtime system can reduce or increase power usage on a node-by-node basis by reducing or increasing the clock frequencies on the entire node
  - Changes in node-level power allocation are low-overhead
- Approach
  - Send maximum power to nodes that are behind in virtual time, causing them to execute at max speed
  - Reduce power to nodes that are ahead in virtual time, causing them to slow down (relatively)
  - Energy management for short term adjustments
  - Migration for longer-term adjustments



DCM also should be interpreted is including the migration of objects for purposes of getting those the communicate frequently to be located near one another in the architecture of the machine so as to reduce latency, and if possible to get the located on the same node so that communication can be through shared memory.

However, knowing when migration to reduce is appropriate or not is very tricky. First, of course, it is only worth reducing communication latency *if that communication is on a critical path* of the future computation. But even if that is perfectly known, it is still not obvious how to instrument to determine when to migrate and when not.

For example, in the leftmost diagram the communication pattern is that one object is sending frequent messages to the other. But there is no return traffic — the communication is entirely one way. This is a case in which the two objects are essentially stages in a pipeline, and there is very little to be gained in migrating them closer together. If a million messages were sent, and they were moved closer together so that the communication latency were, say 1 usec instead of 10 used (a latency reduction of 90%) the execution time for the entire computation would only be shortened by 9 usec, *not* 9 million usec.

On the other hand, in the middle diagram where there is traffic in both directions it is possible for every message latency to be on the critical path of the entire computation. In that case reducing latency from 10 used to 1 used would save 9 used for each message in the critical path. If there are a million messages exchanged, and all are on the critical path (because the application is communication bound) then that saves 9 million usec in of total runtime end to end.

In the third diagram messages are circulating among three objects. This is like the middle case in that if all of the messages are on the critical path, then it will definitely pay to migrate the three of them closer together, or even just two of them. The problem is that it is not clear how to instrument in order to detect this pattern, since no one node or pair of nodes has enough information to recognize it.

New Research Needed in DCM

- Better estimates of critical path in compute-bound computations
- Estimates of critical path in latency-bound computations
- Even having a good estimate of the critical path, how do we suggest particular migrations?
  - What objects should be moved from where to where?
- How do we consider the costs of DCM in a cost-benefit analysis?
  - The cost of the instrumentation, and the migration decisions
  - The cost of migration itself
- Unified theory of DCM
- Unification of DCM with energy management

Space-Time Symmetry

Symmetries in Time Warp

- Attention to symmetry allows for
  - simple explanations
  - clean APIs
  - clean implementations
  - deeper understanding
  - unexpected variations and generalizations
- The semantics of optimistic methods have been carefully designed to exhibit many symmetries
  - input message / output message symmetry
  - forward / reverse message transmission
  - event message / state symmetry
  - message / antimessage symmetry
  - forward / backward in time symmetry

The Time Warp optimistic method for implementing virtual time was deliberately designed with numerous symmetries in mind. Whenever there was a "near symmetry", we asked "what happens if you make the symmetry perfect, and in every case the system was improved and new insight gained.

Extended analogy between virtual memory (demand paging) and virtual time (time warp)

virtual memory	virtual time
virtual address	virtual time
page	event
page out of memory	event in the past
page in memory	event in the present or future
page fault (caused by addressing a page out of memory)	rollback ("time fault" caused by addressing an event in the past)
pure demand paging (retrieve page only when page out of memory is referenced)	pure demand rollback (roll back only when a virtual time in the past is referenced)
thrashing: too many page faults	thrashing: too many rollbacks
cure: allocate more real memory to hold pages	cure: allocate more real time for events (i.e. slow down!)

The term "virtual time" was chosen partly because the term "simulation time" suggested that it was only for simulations, whereas we intended wider applicability, and partly because of this extended analogy to virtual memory, which we consciously used to clarify the ideas of virtual time.

Examples of unexpected innovations inspired by symmetry

- Message sendback — symmetric to process rollback
- "Jump forward" rollback optimization — symmetric to lazy cancellation
- Antistates and anti-objects — inspired by antimessages

There were several unexpected innovations in the Time Warp algorithm that were directly inspired by symmetry considerations.

Message sendback, used for flow control and memory management in general, was directly inspired by object rollback. By asking the question "Why can we roll back computation but not communication?" the answer turned out to be "We can!", and it was a very important discovery.

Lazy cancellation as discovered early. But it was a long time before the symmetry between states and messages became clear enough that we could ask the question "Is there a state analog to lazy cancellation for messages?". The answer turned out to be yes, and it was dubbed the "jump forward optimization". We have not talked about that in this course, but it is present in the literature.

Anti-states and anti-objects were directly inspired by antimessages and the question "Because of the strong symmetry between states and messages, since there are two kinds of messages, + and -, why aren't there also two kinds of states?". The answer was that of course you could define that, and when you do out pops the notion of anti-objects. If you run two objects, P and anti-P in the same computation, the results is that anti-P exactly cancels all of the side effects if P, and it does so asynchronously and concurrently.

Object / anti-object Symmetry Group

- **Define NOOP:**
  - Object with *empty state* that does *nothing* when it receives an event message
- **Let || be a binary operation on objects that means to execute them *at the same virtual location*.**
  - Two objects at the same virtual location both receive and process the same incoming messages using their respective separate states.
- **If P is an object, define −P as its anti-object.**
  - −P is in the same virtual location as P and starts in the state that is the antistate of the one P starts in (i.e. same state but the opposite "sign").
  - Whenever P sends an event message M, −P computes and sends the antimessage −M
  - When P saves a state S, −P saves the antistate −S, and erases it
- **Then || over the space of all objects forms an Abelian group operation, with NOOP as its identity**

```
P || (Q || R) == (P || Q) || R          associativity
P || Q        ==  Q || P                commutativity
P || NOOP     ==  NOOP || P     =  P    identity
P || -P       ==  NOOP                  inverse
```

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 50

In this slide P, Q, and R are objects. P ||Q means to execute P and Q in parallel and *at the same virtual location!* What does it mean to execute two objects at the same virtual location? It means that whenever any event message is sent to P is also sent to Q at the same time. Put another way, at the point in virtual space-time where the event occurs, there are *two states* (one from each object) and *one event message*. If that sound bizarre, just remember that we have already had to deal in the early part of this course with the fact that we sometimes have two event messages arriving at the same time at the same location. We have called this *tie*, and we needed a *tie-breaking rule* to determine what to do in that case. But since messages and states are supposed to be symmetric to one another (and we want them to be) then we need to be able to consider and handle the events in which there are two states and one event message (and more generally, **m** states and **n** messages). That is what we are doing here. And what we mean by executing an event with two states and one event message (our state-tie-breaking rule) is that both objects separately process the event message in the context of their separate states, each making its own state changes and each sending out new event messages as calculated. But, since they both are at the same virtual location, they both send event messages with the same sender and send time. Note that the "sender" is not really the "name" of the object, since "P" and "Q" have different "names"; rather the sender of a message is more properly the "virtual location" of the sending object, and side P and Q have the same virtual location, the messages they send have the same sender field. (This is important because when P and -P are at the same virtual location, we want the messages they send to be true anti messages of one another, which they would not be if they had different senders.)

We define an antistate to be the same thing as a (positive) state except that when it processes a (positive) event message, any event messages it sends as a result are negative, not positive, and the copies of outgoing messages that it saves in its output queue are positive, not negative.

The antiobject of object P, denoted as -P, is identical to P in that it has the same event methods and the same initial state, except its initial (and all subsequent states) are negative instead of (the normal) positive states.

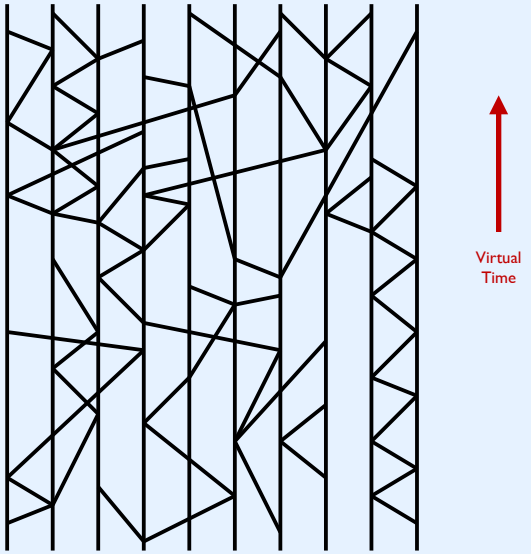
These definitions lead to the consideration that the space of all objects and anti-objects forms an *Abelian group* with respect to *parallel composition at the same virtual location*. (Strictly speaking the elements of the group are not the objects, but object *equivalence classes* — two distinct objects that generate the same sequences of states and event messages when driven by the same sequence of incoming event messages are equivalent.)

(This exposition can be continued to consider state - antistate annihilation, symmetric to message-antimessage annihilation, but the margin of this slide is to small to develop this subject.)

What is the point of objects and anti-objects. Well I am not sure what the practical application of this would be, but I think it is pretty neat that in the virtual space-time formalism you can actually exactly nullify all of the side effects, direct and indirect, of literally *any* object, just by launching its anti-object at the same virtual location. And the nullification takes place totally asynchronously!

Anti-objects

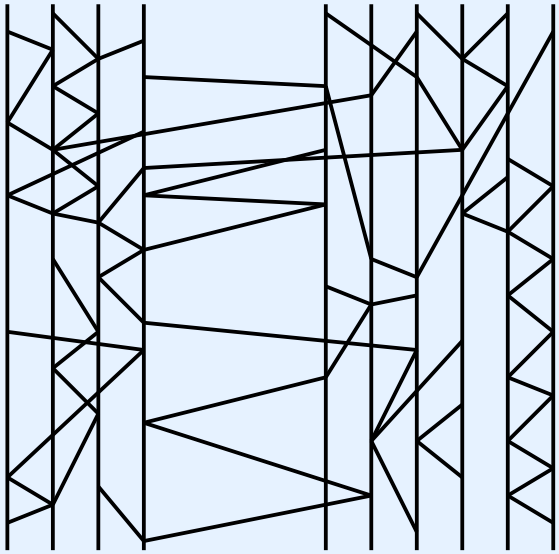
- Objects **P** and **−P** (anti-**P**) start in states **S** and **−S** (anti-**S**)
- **P** and **−P** receive exactly the same inputs
- However, **−P** sends antimessages to the same places **P** sends positive messages
- All of **P**'s side effects on other objects are exactly canceled, as if **P** did not exist.



Consider this set of objects interacting as shown. Virtual time goes upward.

Anti-objects

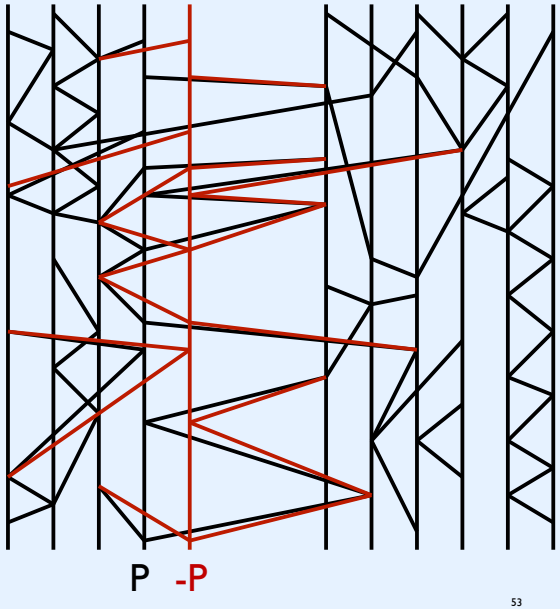
- Objects P and −P (anti-P) start in states S and −S (anti-S)
- P and −P receive exactly the same inputs
- However, −P sends antimessages to the same places P sends positive messages
- All of P's side effects on other objects are exactly canceled, as if P did not exist.



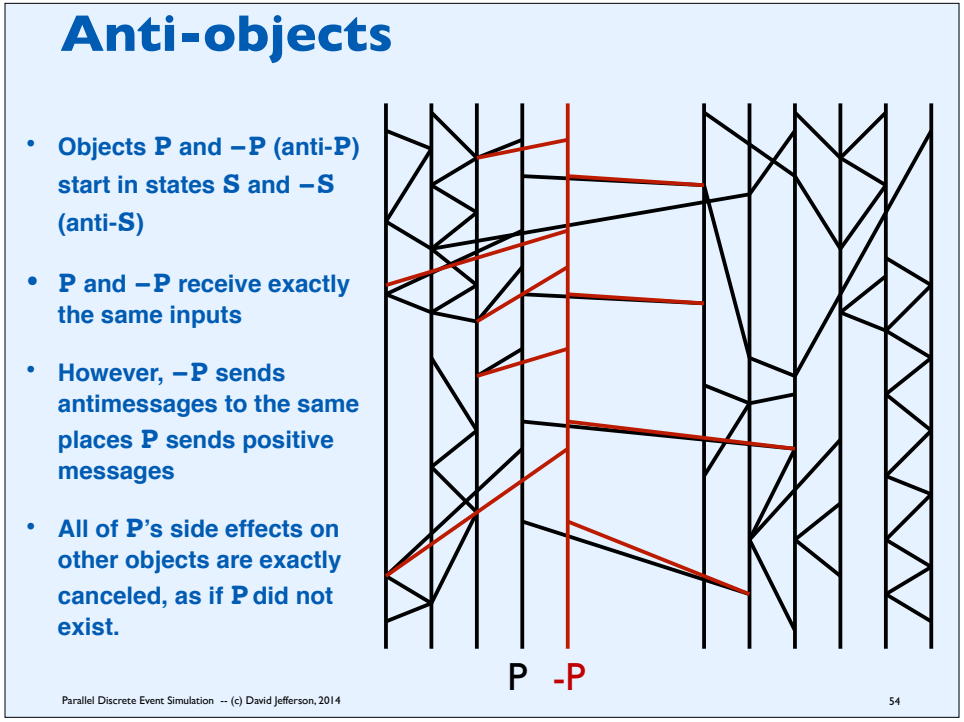
Same object interaction diagram — just stretched out in one place.

Anti-objects

- Objects **P** and **−P** (anti-**P**) start in states **S** and **−S** (anti-**S**)
- **P** and **−P** receive exactly the same inputs
- However, **−P** sends antimessages to the same places **P** sends positive messages
- All of **P**'s side effects on other objects are exactly canceled, as if **P** did not exist.



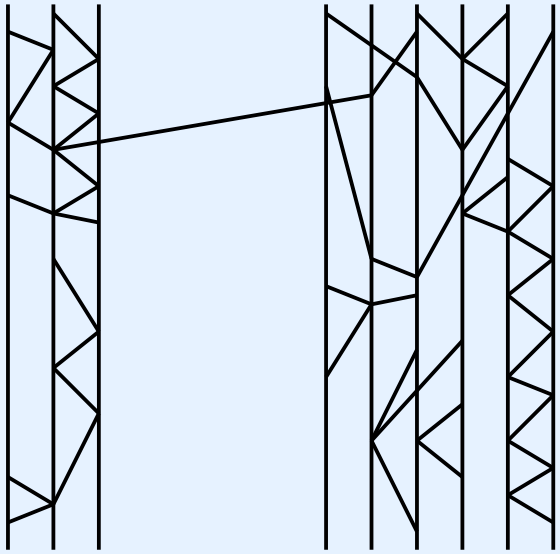
Call the 4th object **P**. We have added **-P** in **red** next to it. **-P** gets all the same input messages as P, but instead of sending positive event messages it sends negative ones.



In this diagram we have removed all of the messages output by both **P** and **-P** because of course they cancel one another and thus have a net null effect. We are still showing the messages inout to **P** and **-P** because nothing prevents other processes from sending messages to them. We should note, however, that **P** and **-P** will not get the same messages as **P** would alone, of course, because some of the messages sent to **P** (and duplicated to **-P**) are affected by or prompted by the messages output from **P** (when not cancelled by **-P**).

Anti-objects

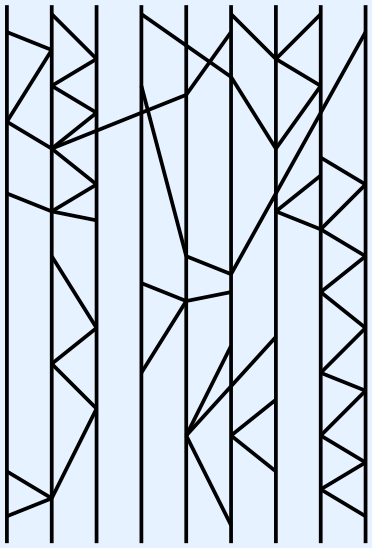
- Objects P and −P (anti-P) start in states S and −S (anti-S)
- P and −P receive exactly the same inputs
- However, −P sends antimessages to the same places P sends positive messages
- All of P's side effects on other objects are exactly canceled, as if P did not exist.



Despite all of the messages sent to P and **-P**, they produce no output event messages that get committed, so they act effectively as a kind of event message sink. Since they have no effect on the computation, it is as if they are not there at all.

Anti-objects

- Objects **P** and **−P** (anti-**P**) start in states **S** and **−S** (anti-**S**)
- **P** and **−P** receive exactly the same inputs
- However, **−P** sends antimessages to the same places **P** sends positive messages
- All of **P**'s side effects on other objects are exactly canceled, as if **P** did not exist.



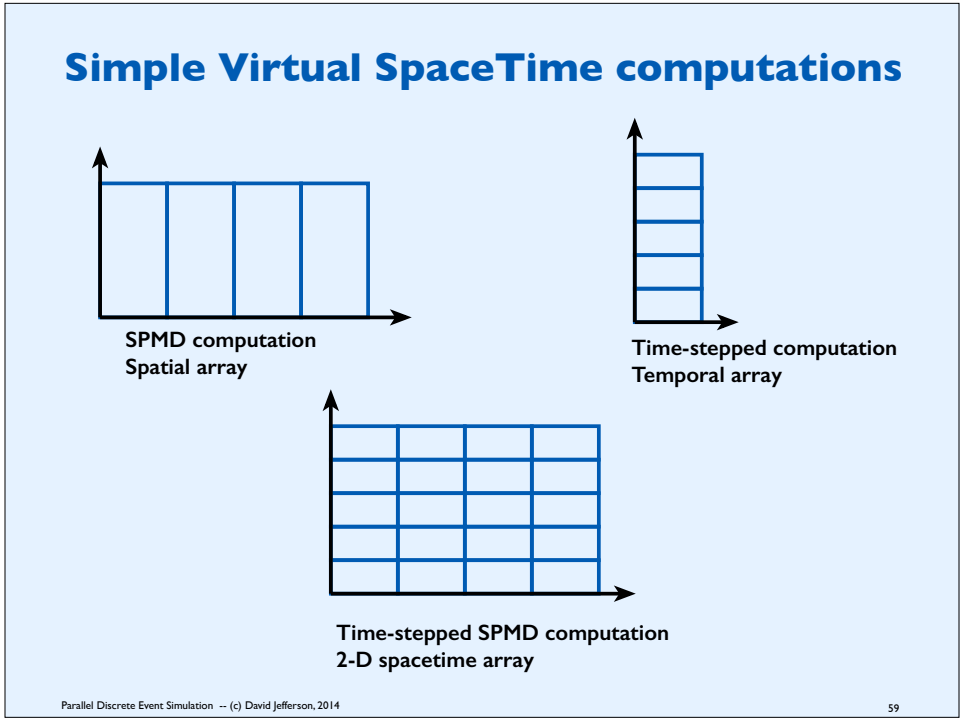
This is the same slide as the previous one, but with the "gap" closed up.

Virtual SpaceTime

Dualities in space and time?

Spatial notions	Temporal notions
array	fixed length time series
SPMD parallelism	time-stepped, for-loop
struct	sequential composition
allocation / allocator	scheduling / scheduler
pointer / address	virtual time value
memory location	broadcast (instantaneous in virtual time)
atomic action	encapsulated data

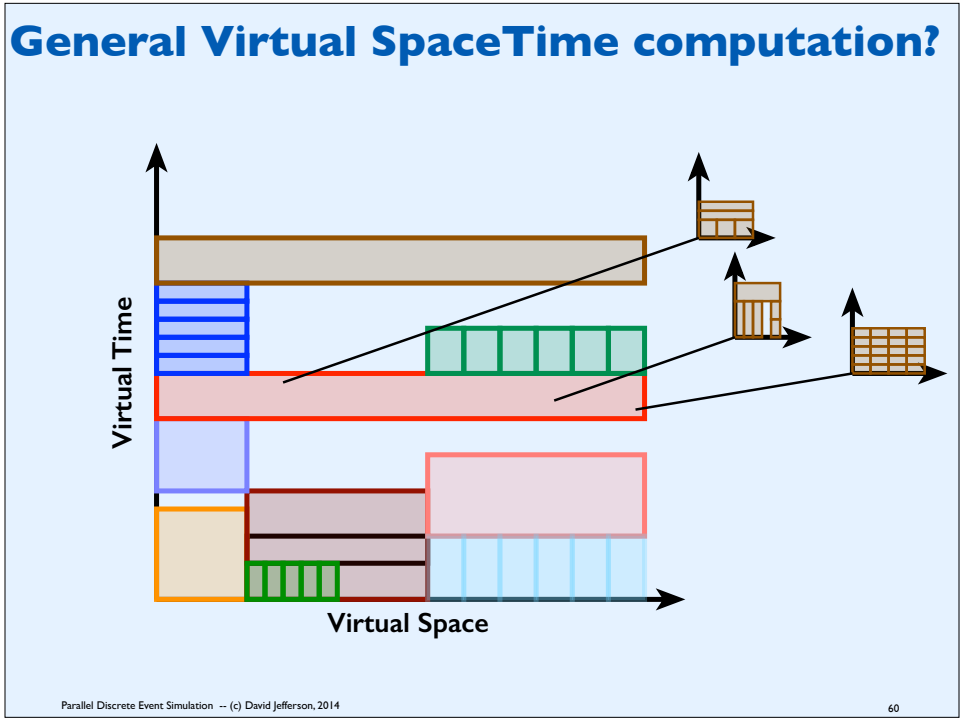
Among the symmetries considered is virtual space-virtual time symmetry. But that is not a perfect symmetry, since space and time are not generally interchangeable with one another — there is still a directionality and connection to causality in time that space does not have. Still, there is at least a partial duality between them, and some spatial programming notions have natural duals in time. This is the beginning of the idea that virtual space-time may be the semantic foundation of a new parallel programming paradigm.



One way to look at parallel computation is as a process unfolding in virtual space-time. We can view an SPMD computation, for example, as an *array*. It is not an array of data elements, however, but an array of communicating computations. Thinking of it as an array is appropriate because the elements are indexed by integers (*rank*, in MPI), because that the number of elements is generally constant (as with data arrays), and because all of the elements are of the same *type* (i.e. share the same code), as with data arrays.

If we "rotate" an SPMD computation 90 degrees in virtual space-time, and struggle reinterpret what was broadcast communication is now memory (of the past, but also of the future) and what was memory is now instantaneous broadcast, then we can see that the space-time dual of an array is a time series; and if the objects in the time series are computations instead of just date, then a temporal array is a time stepped computation, i.e. a big for-loop (with a parallel body).

And a time-stepped SPMD computation can be viewed as a two-dimensional array, one dimension of virtual space and one of virtual time.

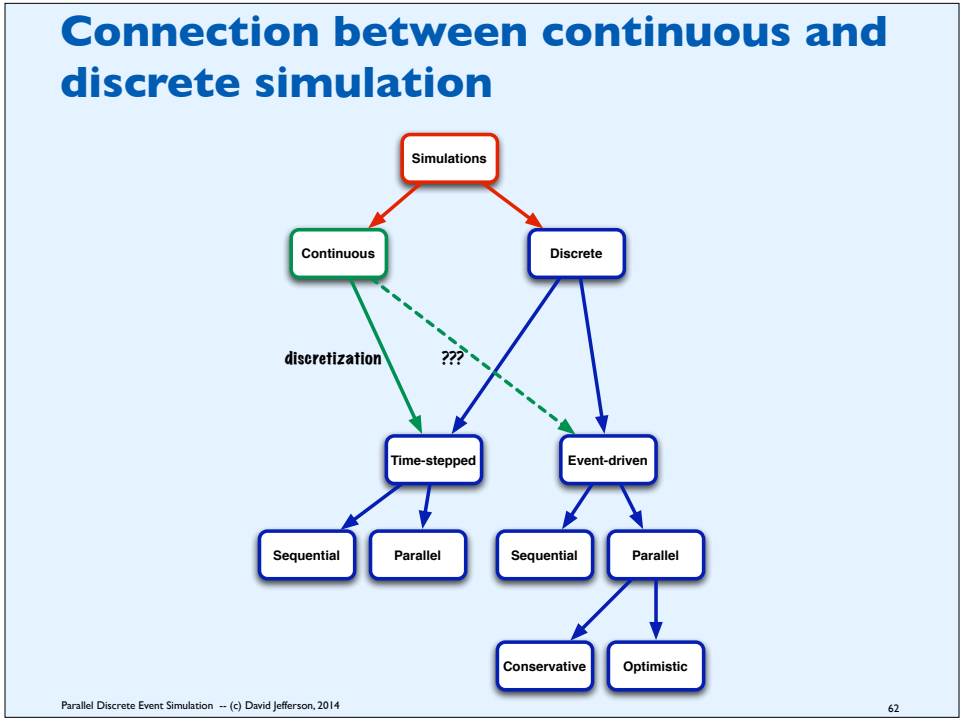


Can we imagine a new computational paradigm based on virtual space-time? In this diagram we see a computation composed of many smaller ones composed in space (horizontal), and composed in time (vertical). Some of the component computations arrays, either temporal or spacial.

And some events (points in spacetime) may invoke whole new space-time computations that begin and complete during the course of that one event. The three small brown space-time diagrams in the upper right represent that. In effect those are subcomputations, making this in effect a multiscale computation.

In this illustration, some of the virtual space-time is not filled with any computation. That does not necessarily mean that resources are wasted, however. Just as using widely separated parts of virtual memory address space without using the address space in between does not result in any wasted resources, the same is true of virtual time.

Event-driven Methods for Continuous Simulation



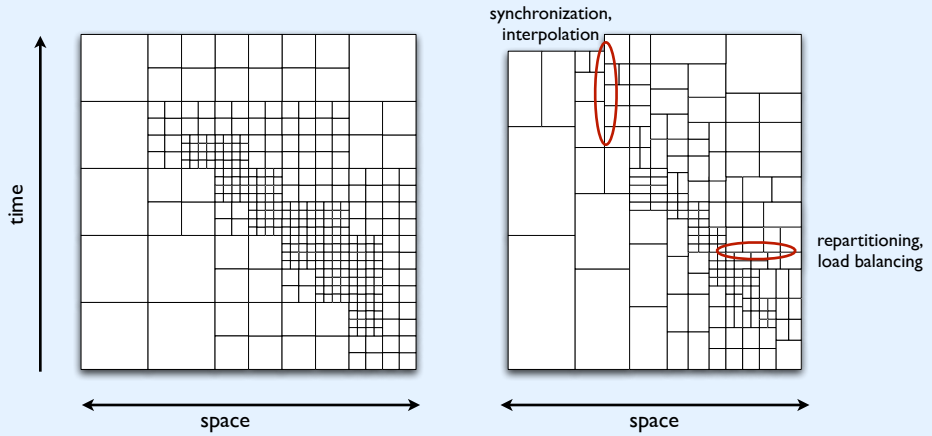
Discretization of a continuous model is the transformation from equational form into a discrete model. Virtually always this is a time-stepped model.

Grand challenge: Develop techniques for transforming continuous models into discrete event models. No one knows how to do that in general, although there are hints coming. Would help solve the general problem of multiscale simulation, in time at least.

Event-driven methods for continuous simulation?

- Can we discretize continuous models into event driven form instead of time-stepped?
- Event-driven continuous simulation
  - Should work well when there are very high derivatives and/or chaotic behavior that "looks like" unexpected discontinuities
  - Could it improve upon AMR?
- Grand Challenge: Unify continuous and discrete event simulation
  - DEVS formalism?

Time stepped vs. event-driven continuous simulation



Adaptive Mesh Refinement

hypothetical event-driven algorithm

Future Basic Research in Parallel Discrete Event Simulation and Scalable Computation

Research Issues

- Performance
  - dynamic configuration management theory
  - low latency migration and message forwarding mechanisms
  - dynamic energy management
  - mixed conservative / optimistic methods
  - hardware support for virtual time and rollback
- Reverse computation
  - multiple programming languages
  - more sophisticated methods
  - languages designed with reversal in mind

Research Issues

- Software engineering technology and tools for model construction
  - coupling of simulations
    - spatial coupling (one-way and two-way boundary relationships)
      - temporal coupling (one simulation creates initial conditions for (part of)p other)
    - multiscale systems
      - objects as simulations
      - events as simulations
  - virtual machines in simulations
  - discrete methods for continuous problems
    - discretization of ODEs and PDEs directly into event-driven form
    - mixed discrete and continuous simulation
    - coupling of continuous and discrete event simulations
  - unification of rollback methods for synchronization and for backtracking
- Virtual Time for Simulation and Big Data
  - file systems
  - databases

End