Parallel Discrete Event Simulation Course

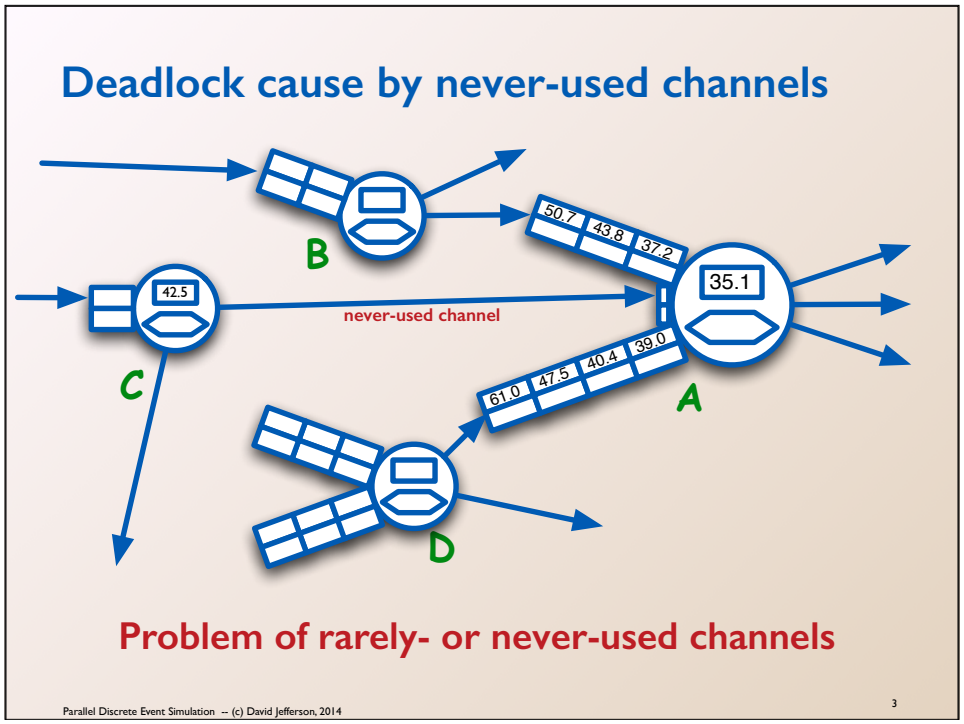
#6

David Jefferson
Lawrence Livermore National Laboratory

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-651651

Reprise

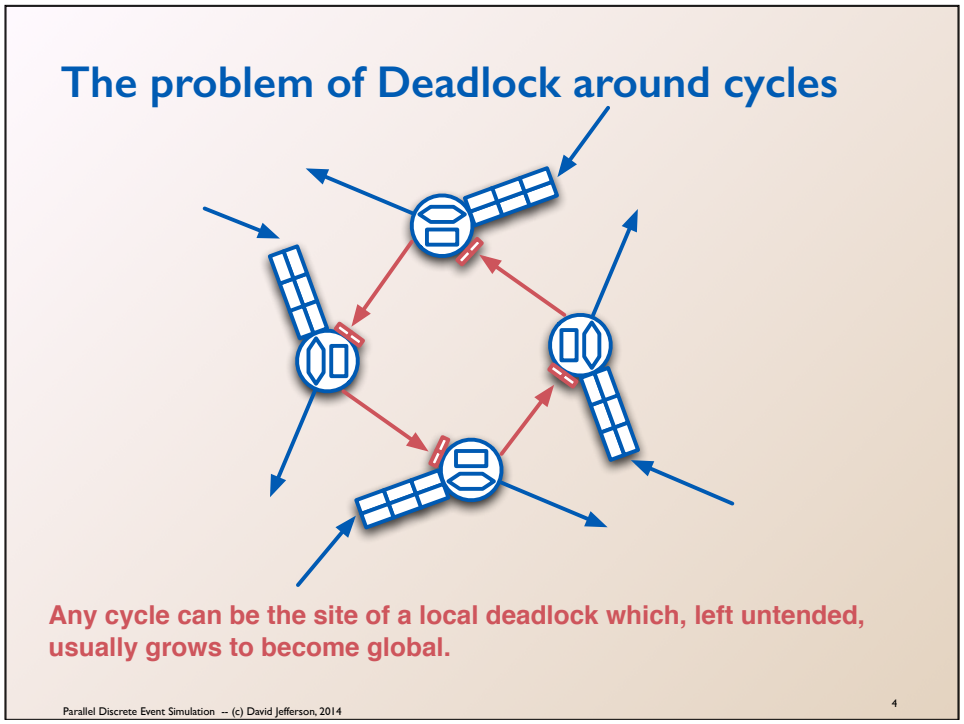


Here is one serious problem with the *naïve* conservative algorithm. What happens if object C has a channel to A but *rarely or never sends event messages* along that channel?Then the input queue associated with that channel at A is almost always, or always, empty! Hence, A is blocked most of the time, or even permanently, though it has plenty of events queued for execution. C may have progressed to a time (42.5) well ahead of the time of the next event that A will execute, so it cannot possibly send an event that would cause A to receive a massage in the past. But A does not know this, and thus stays blocked! That leads to A being at the very least badly delayed or even permanently blocked. And that condition spreads to other objects that A is supposed to send messages to, but can't. It can also lead to the blocking of processes that send to A (i.e. B and D) as A runs out of memory buffering their messages and then B and D block for flow-control. In effect, blocking propagates both forward and backward along directed arcs outward from A. Hence, except in unusual cases (such as a graph with disconnected components) if a channel is never used, the situation devolves into global deadlock!

For this reason, one cannot get around the requirement of a static interaction graph just by deciding to choose as the complete graph connecting all objects to one another in both directions. Besides requiring O(n**2) storage, that would leave most queues empty most of the time, and the system would thus be deadlocked.

Note a highly unusual peculiarity of the conservative, graph-oriented PDES algorithm: An object is slowed down or stopped when work is *not* sent regularly along one of the channels to it! An object only progresses smoothly when events are regularly sent to it along every one of its input channels. If it does not get enough work to do from *all* of its suppliers, then it blocks!?! *Usually failure to give a server work from one of its clients just makes things go all the faster for the others. But not in this case.*

This property is responsible for all of the difficulties with conservative algorithms. It is not shared by optimistic algorithms. I know of no other protocol in CS with this property -- it seems downright perverse!



The most worrisome hazard for a conservative PDES is that any directed cycle of queues in the graph can become a local deadlock if all become simultaneously empty. And the deadlock grows inexorably as the queues to which the objects in the deadlock should be sending messages eventually become empty and thus block still more objects. Meanwhile long backups of messages may grow in the nonempty queues, causing a flow control problems and further blocking. A local deadlock such as this, left untreated, will quickly grow to become global.

One might try to catch that deadlock while it is still local, and try to prevent or break it somehow. But the fact is that even detecting a "local" deadlock is way too costly. A deadlock involving only 2 or 3 objects out of a million is computationally way too expensive to monitor for, especially since they may reside in different places among the thousands of hardware nodes of the underlying cluster. (The deadlock is "local" in the graph sense, but it is not necessarily "local" in the sense of all vertices being located on one physical node!). However, the cycle does not have to be small. A cycle of empty queues involving very large numbers of objects can just as easily happen.

If the deadlock becomes global (which it will if the graph is connected), that condition is easily detected by periodically counting the number of objects that are not blocked, and when that total declines to 0, a global deadlock is detected. A global deadlock is breakable. It is always the case that the object(s) with the lowest timestamped message globally can safely execute even if it has some empty input queues. But it cannot do so until the deadlock becomes global and is detected! In the mean time, before the deadlock if fully global, more and more objects block, and the degree of parallelism declines to zero. So performance sucks!

In fact, we should note that under the Naïve Conservative Algorithm, the any simulation with a cycle (which is practically all) actually starts in or near deadlock (!) with most queues empty. This is why I call the Naïve Conservative Graph-oriented PDES algorithm "naïve". A new idea has to be injected to make this work. That idea, of course is *lookahead*.

How to avoid deadlock

- **If any channel is starved of messages (event messages and null messages) then a deadlock will inevitably spread from around the receiving LP on that channel.**
  - **Peculiar that the *lack* of traffic on a channel is what causes the problem**
  - **This is why we cannot simply adopt the complete graph of with arcs between all nodes**
- **A simulation is *deadlock free* ⇔**

  every channel **transmits a *never-ending sequence of messages*** **(event and null) messages with *increasing time stamps***
- **Most null message algorithms are variations on this theme:**
  - **Whenever an LP simulation clock increases, send updated null messages out all channels.**
  - **Send additional null messages when better lookahead information is**

Deadlock-free Null Message Algorithm

```
while (true) do {
    simTime = ∞;
    for (all input queues Q) {
        if ( Q.empty() ) {
            wait for message to arrive in Q;
        }
        if ( Q.head().timestamp < simTime ) (
            q = Q;
            simTime = Q.head().timestamp();
        }
    }

    if ( simTime > StopTime ) break;

    if ( !q.head().nullMessage() ) {
        executeEvent( q.head() );
    }

    for (all output channels C) {
        C.sendNullMessage(simTime + lookahead);
    }

    q.removeHead();
}
```

Find input queue **q** with lowest timestamp event message or null message across all input queues, waiting for any empty queue to be nonempty.

Update simTime for null messages just as for event messages

Termination test

Execute event messages, but not null messages

Send updated lookahead info out *all* channels, in response to both event messages and null messages. No deadlock if eventually lookahead > 0

Discard the message

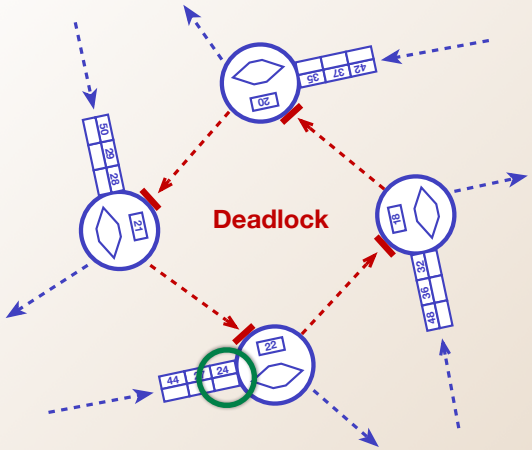
Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 6

The red lines are those added to the "naïve" algorithm on a previous slide to make it deadlock free.

Note that the input queues now contain a mixture of null messages and real event messages, (sorted within each queue of course). And either kind of message can be the one with the lowest time stamp and can cause the simulation clock to increase.

This algorithm is just a clean, compact model algorithm. Many variations are possible, especially regarding when null messages are sent and what the lookahead values are.

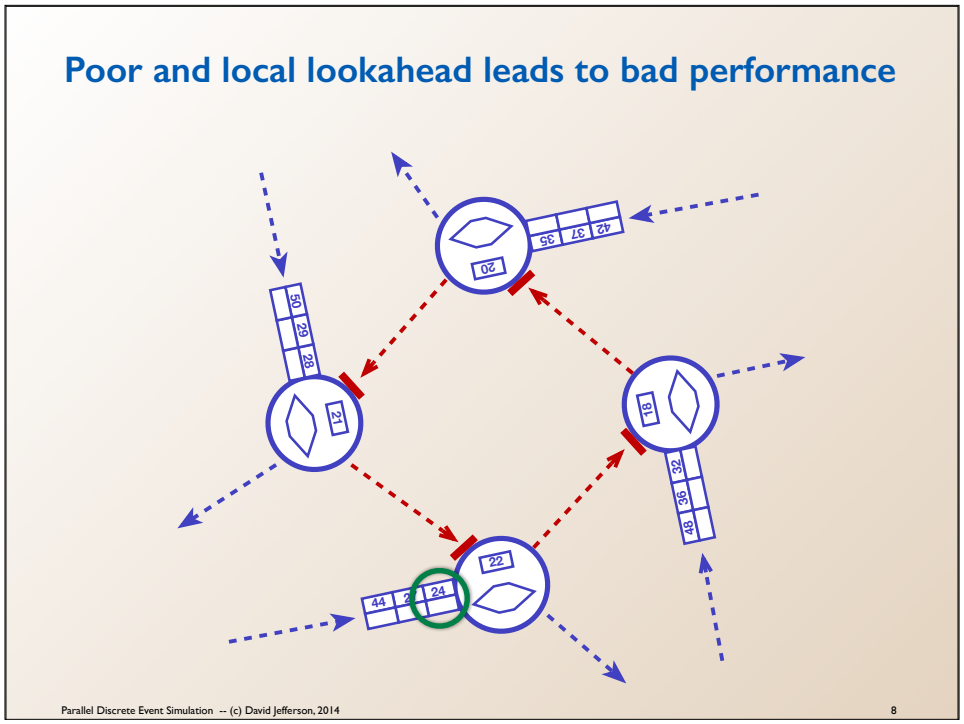
Poor lookahead leads to bad performance



Consider lookahead of 0.1 at each object

Allow null messages to prompt other null messages.

Send null message out all channels output as often as possible



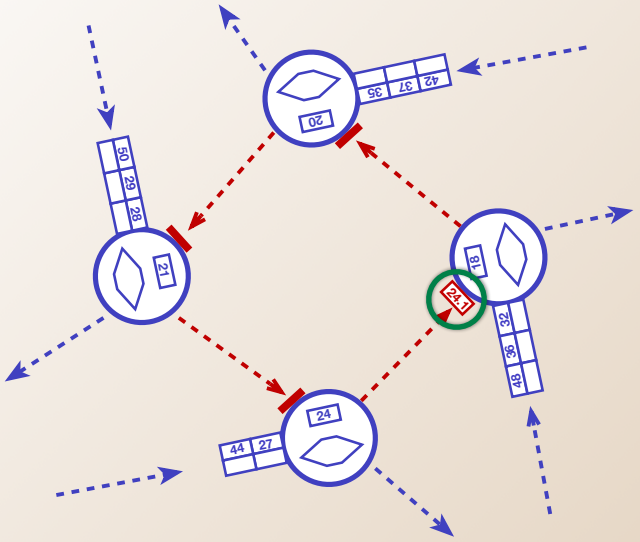
Let's assume we can break this deadlock initially by identifying one safe event (by some means) -- the event message that is circled in an input queue of the South process.

From now on, real event messages will be **blue**, and null lookahead messages will be **red**.

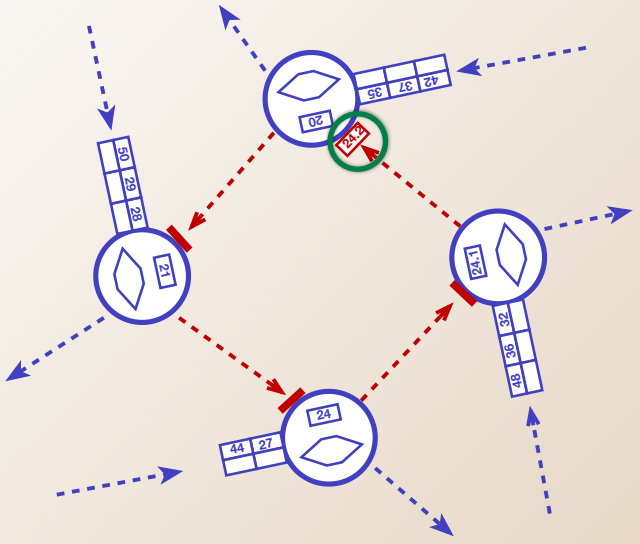
Each following slide will represent all 4 processes either processing one event or updating its lookahead information by sending a null message out all of its output channels.

(Note that sending null messages out all output channels, in response to an incoming null message, will cause an exponential blow-up. But even ignoring that (as we are in this sequence) it still would not always work well.

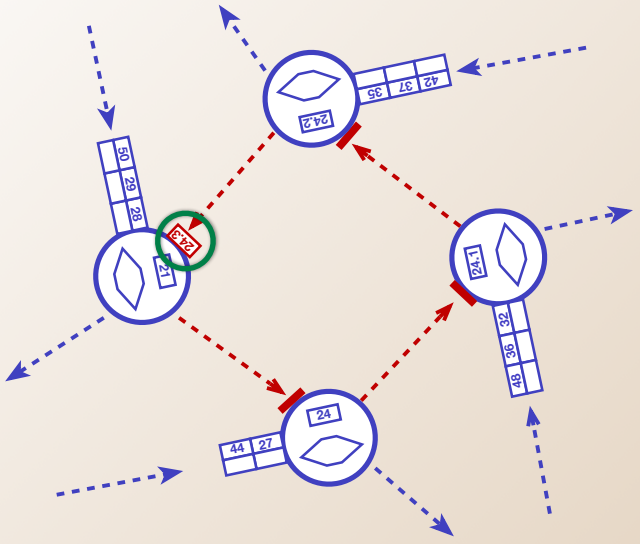
Poor and local lookahead leads to bad performance



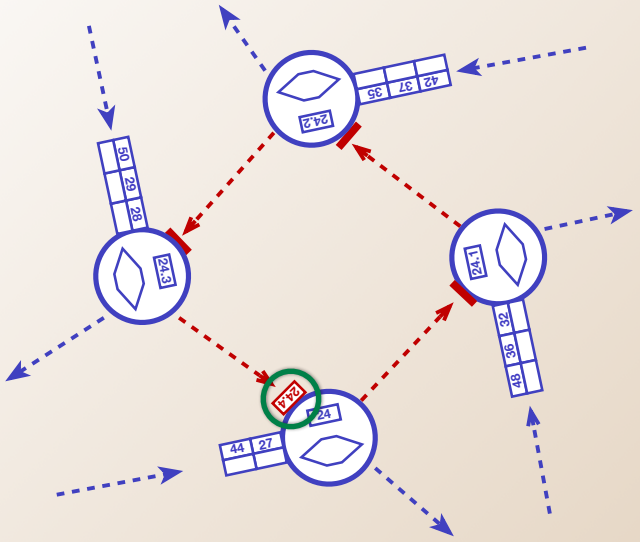
Poor and local lookahead leads to bad performance



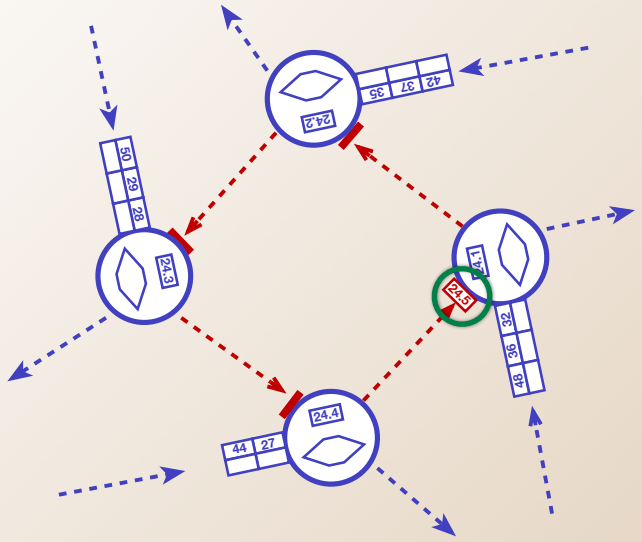
Poor and local lookahead leads to bad performance



Poor and local lookahead leads to bad performance



Poor and local lookahead leads to bad performance



Sources of lookahead information in a model

- **node service time**
  - plane arrives at airport and cannot possible depart in < 0.8 hours
  - incoming packet has to be processed for a minimum of, say, 20 msec before a reply can be sent
- **node refractory time**
  - any two planes leaving the airport must be separated by, say, > 3 minutes
  - any two outgoing packets must be separated by at least, say, 10 µsec of overhead time
- **link travel time**
  - plane departs from airport and cannot possibly arrive at destination in, say, < 3 hours
  - packet latency across the continent through Internet cannot be less than, say, 10 ms.
- **clock-based scheduling**
  - no plane is not scheduled to depart from airport until, say, 13:05
  - no packet can depart on a shared TDMA link (statically time sliced) at least until the sender's next time slice, which is, say, 600 µsec in the future

End Reprise

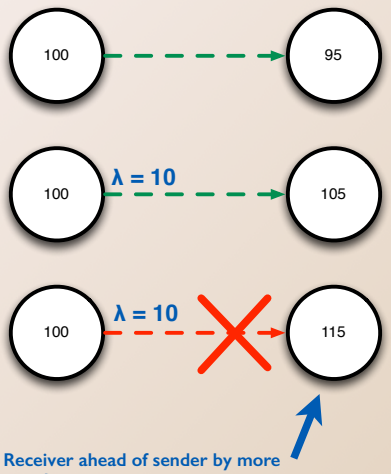
Dynamic object and channel creation

- **Dynamic object creation**
  - Easy when there are channels between objects. (Not so easy otherwise, e.g. Bounded Lag.)
  - Just instantiate with simTime in creator's future
  - Register its lookahead
- **Dynamic channel creation is more complex**
  - Must establish LookAhead value λ for sender wrt receiver
  - If, at the time of creating the channel,

    $T_{sender} + \lambda < T_{receiver}$

    then the receiver must be blocked until

    $T_{sender} + \lambda >= T_{receiver}$
  - Sender must not be able to send an event message with a timestamp less than $T_{receiver}$



Dynamic object creation is not trivial when there are not discrete event message channels involved, as there aren't with the Bounded Lag protocol.

With Bounded Lag:

1) We must establish new object's lookahead "distance" to and from all other objects, and broadcast it.

   Note that "to" and "from" may be different because D(p,q) != D(q,p)

2) We must be sure that the new object is not too far in the future of any other object in its "w-neighborhood".

   * It must block until it obeys the lookahead inequalities wrt all other senders in its w-neighborhood

   * All objects that act as receivers from the new object must block until new object advances to the point where it obeys the lookahead inequality wrt to them

   Note that these two conditions are not the same because D(p,q) != D(q,p)

3) Notice that the first of these is a global operation that in principle involves every node, and the second is a regional operation involving all objects in the w-neighborhood

With other protocols, different lookahead considerations arise, all boiling down to these two requirements:

It must be impossible for any other object to send an event to the new object arriving in its past.
It must be impossible for the new object to send an event to any other object arriving in its past.

Criticisms of conservative paradigm

- Static model restrictions of some kind are generally required for decent performance. Highly dynamic models, or those with port lookahead, are basically excluded.
- Additional synchronization logic required (lookahead) that is not needed for sequential execution or optimistic parallel execution.
- Lack of clean separation between the model and the simulator.
- Conservative simulation performance is not robust in the face of small changes to the model.
- Conservative simulators do not generally achieve the maximum concurrency possible even when the computation is balanced.
- Somewhat surprisingly, graph-oriented conservative simulators are not space optimal either — they may require much more space than the sequential algorithm.
- Development and long term maintenance of conservative models, especially federated models, can thus be a pain.

Static restrictions on models, while not fundamental, are usually imposed by the simulator.
Except in special cases, conservative algorithms require additional logic (lookahead information and algorithms) not required for sequential execution, to avoid or break local deadlocks and to lubricate concurrency in general. The "special cases" are mostly cycle-free feed-forward networks, or else time-stepped simulations cast as event driven simulations.
The need for lookahead logic breaks the clean separation we like to have between the model and the simulator.

A model designed for one conservative simulator (one with null messages, for example) has to be modified algorithmically to work with a different one (say, a Bounded Lag simulator).

Two models designed for different conservative simulators cannot be easily coupled.

Modelers have to know too much about the simulation algorithm

Conservative simulations are not robust in the face of modest changes. A high performance simulation can be ruined by the introduction of an interaction that is in fact quite rare.
Conservative simulators do not achieve the maximum concurrency possible. For models well suited to them they do OK, but in general they spend too much time waiting for possible interactions that in fact rarely or never happen.
Somewhat surprisingly, conservative simulators are not space optimal. In worst case they can require MUCH more space than the corresponding sequential execution, although usually not.

Example of a slight alteration that ruins the performance of a conservative simulation:

> Suppose the airport example was extended with the possibility that while in flight an aircraft might have to make an emergency landing at the nearest airport. That is a rare event, but it has poor lookahead. It means that every airport can now interact with all airports along its flight path, albeit very rarely, and unpredictably (poor lookahead). The lookahead (warning interval in simTime) is quite small compared to the delay between two successive emergency landings.

Critical Path Theory

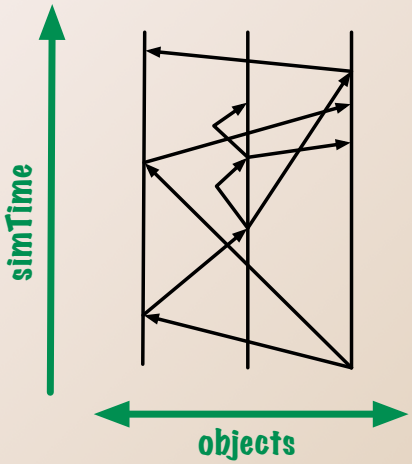
Critical Path Theory

How fast can a particular simulation run on a particular machine?

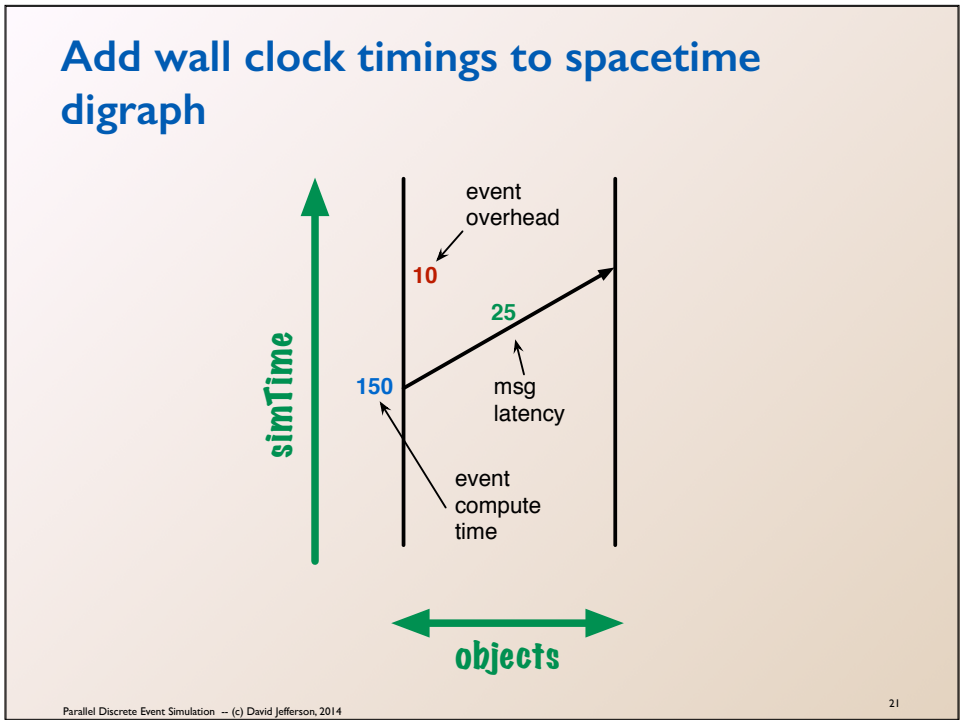
How much parallelism is present in the application?

Causality digraph and parallelism

- Every event (except initial) is scheduled by some other event
- Vertical arcs: successive state changes in one object
- Diagonal arcs: event messages
- Paths represent ***causality chains***
- Events connected by a path must (appear to) be executed sequentially
- Events not connected by a path can be executed in parallel, even if out of time order.



A PDES can be viewed as a digraph in simulation spacetime. In this diagram the horizontal axis is the space of objects, and the vertical axis is simulation time. The nodes in the graph are events, and the arcs are *causal connections* representing information flows. The vertical arcs (without arrow heads) represent the flow of an object's state from one event to the next in that object. The slanted arc represent event messages sent by one event to be received during another.



The numbers here represent wall clock times on a particular (real or idealized) platform.

The **blue** numbers on the events represent the wall clock time taken to execute an event. I have included a range of event execution lengths in this diagram, anywhere from 45 to 200 units of time.

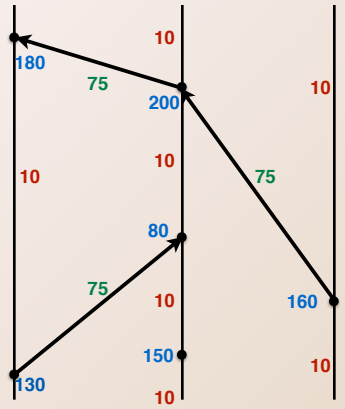
The **red** numbers on the arcs between events in the same LP represent event overhead, i.e. all of the computation that is not direct execution of events. It includes synchronization, storage management, etc. In this case I have made the arbitrary assumption that that is a constant 10 units of time everywhere.

The **green** numbers on the arcs that represent event messages indicate the message latencies, i.e. the wall clock time required to transport a message from one process to another. In this case I have made the arbitrary assumption that it takes 25 time units to send a message between two neighboring LPs, and more time if the communicating LPs are more distant.

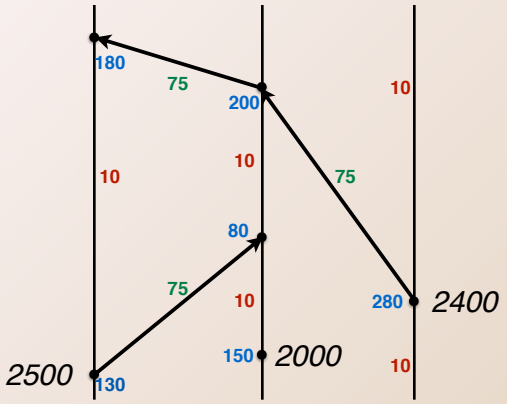
In each case these numbers should either be estimates of the minimum times each of these should take, or the average times. Use minimum times if you wish to get an actual lower bound on runtime of the entire simulation. Use average times (considering message traffic effects and LP scheduling effects) if you want a somewhat more accurate estimate that is not strictly a lower bound.

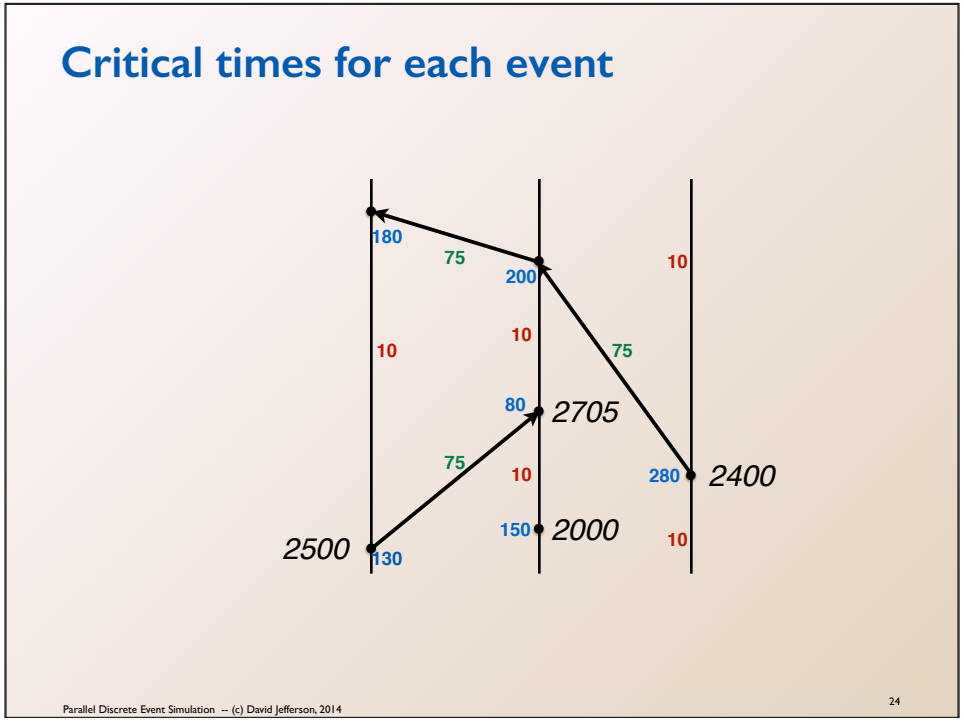
If the event overheads are negligible then you can set them to zero. If the message latency is negligible (e.g. in shared memory) then you can set them to zero. Etc.

Add wall clock timings to spacetime digraph



Critical times for each event



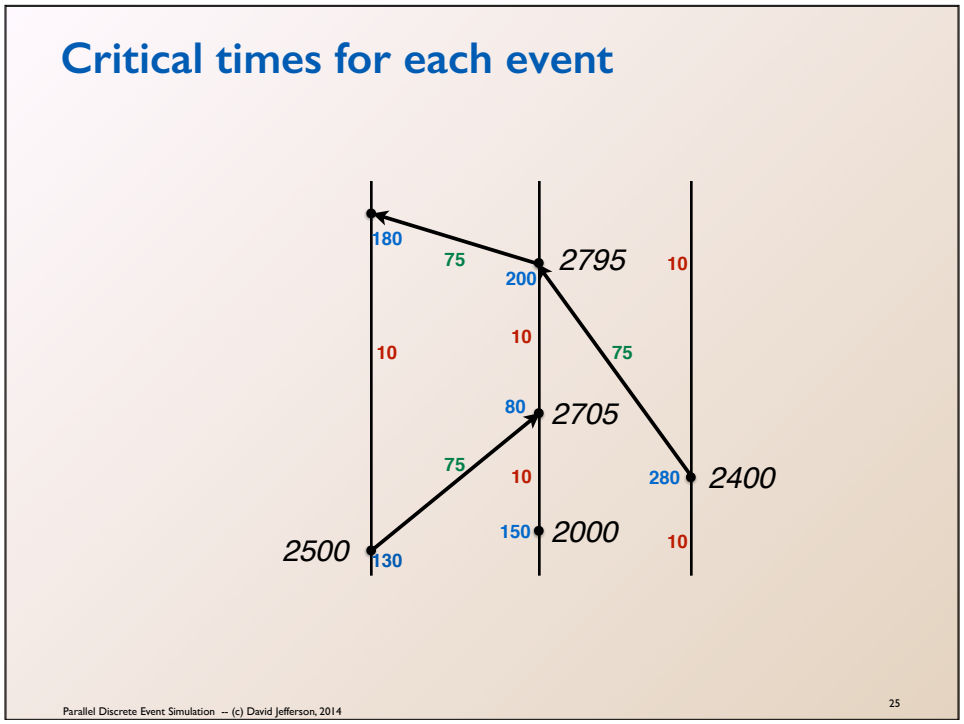


Label both nodes and arcs of this graph with real-time timings according to some model of the performance of the underlying hardware that the simulation will run on.

Graph nodes are labeled with the compute time required for the event it corresponds to.

Diagonal arcs are labeled with the message latency for that event message.

Vertical arcs are labeled with event overhead time.

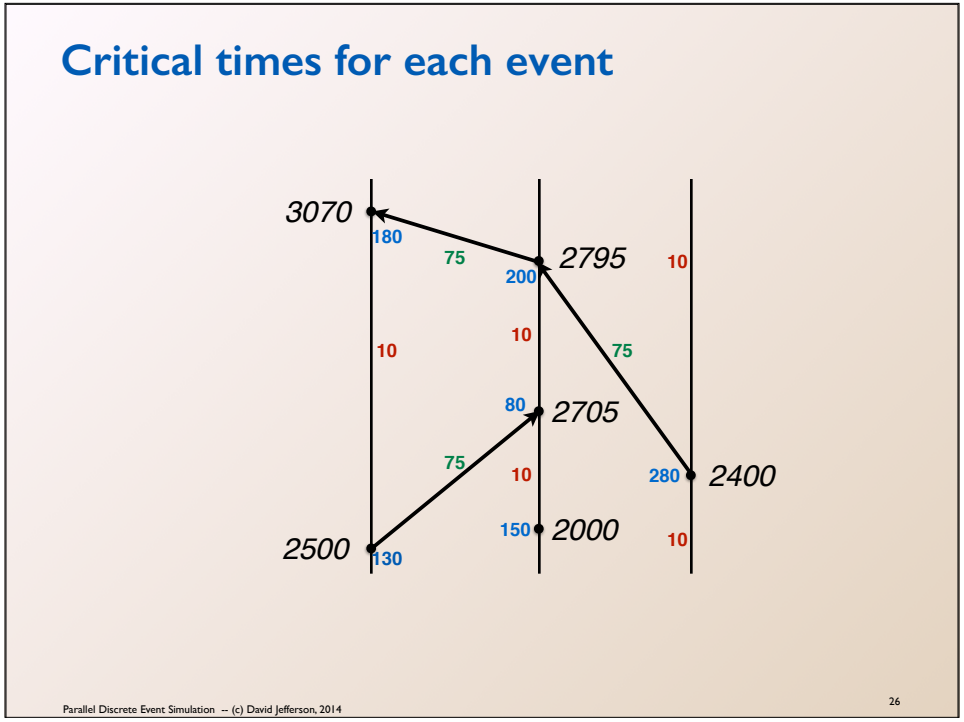


Label both nodes and arcs of this graph with real-time timings according to some model of the performance of the underlying hardware that the simulation will run on.

Graph nodes are labeled with the compute time required for the event it corresponds to.

Diagonal arcs are labeled with the message latency for that event message.

Vertical arcs are labeled with event overhead time.

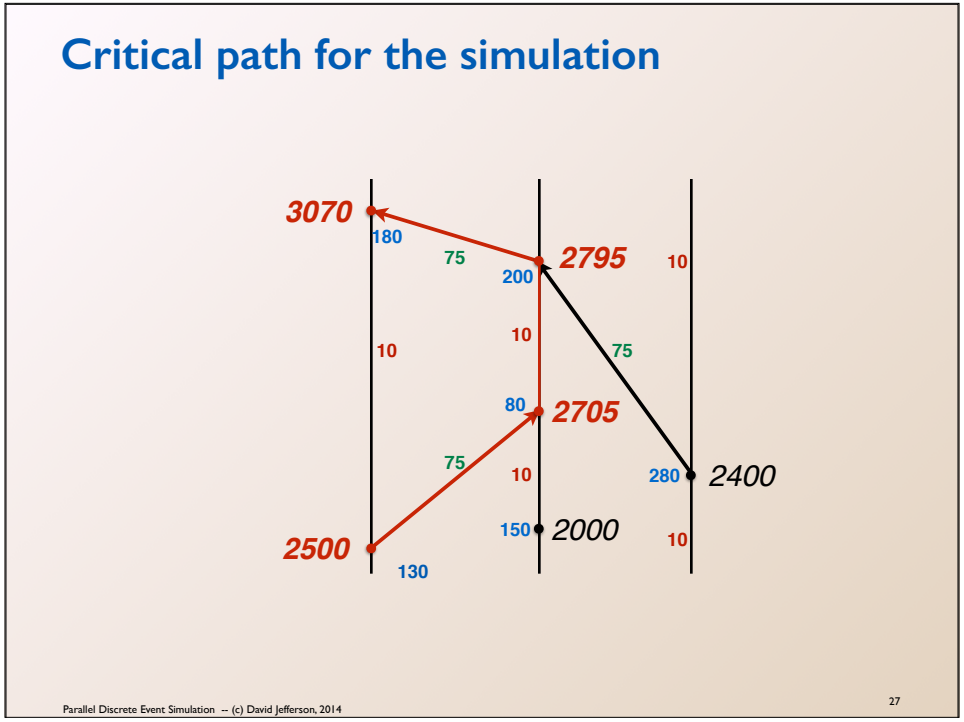


Label both nodes and arcs of this graph with real-time timings according to some model of the performance of the underlying hardware that the simulation will run on.

Graph nodes are labeled with the compute time required for the event it corresponds to.

Diagonal arcs are labeled with the message latency for that event message.

Vertical arcs are labeled with event overhead time.

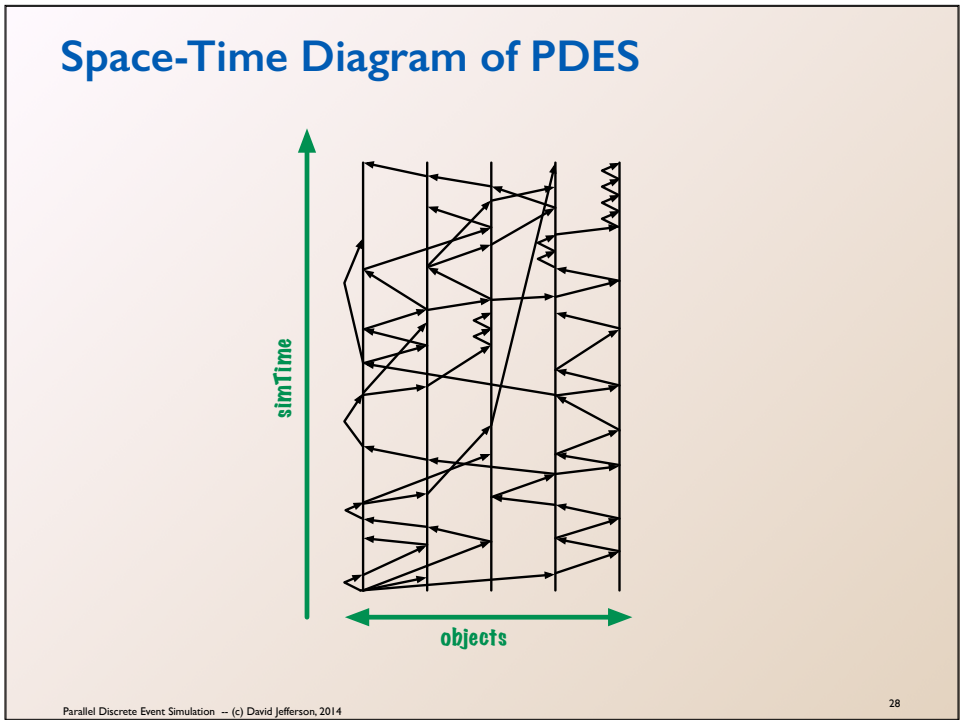


Label both nodes and arcs of this graph with real-time timings according to some model of the performance of the underlying hardware that the simulation will run on.

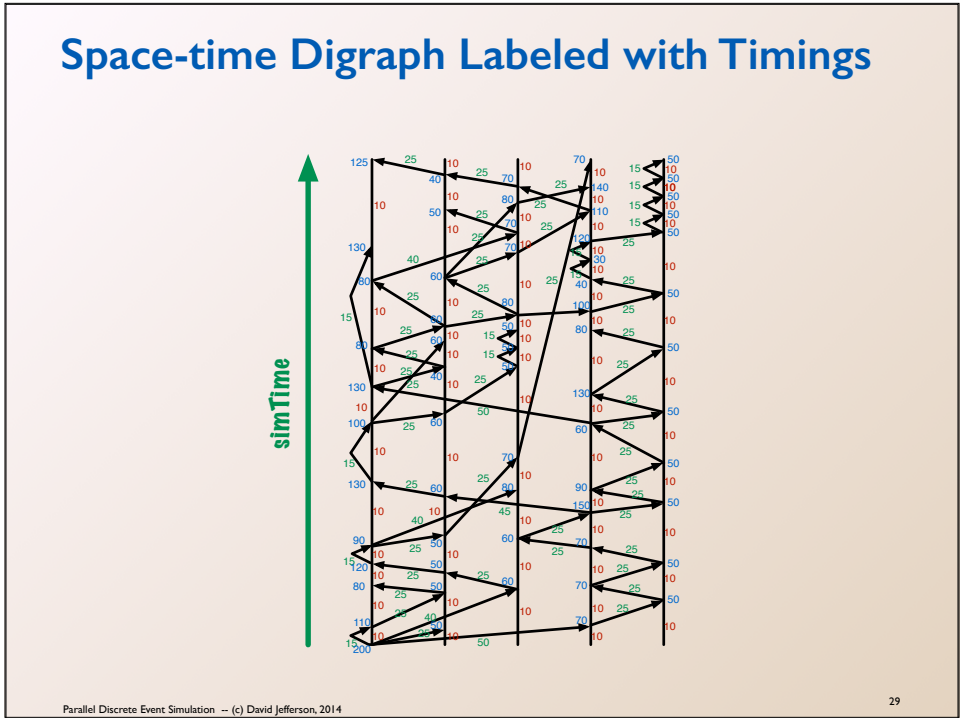
Graph nodes are labeled with the compute time required for the event it corresponds to.

Diagonal arcs are labeled with the message latency for that event message.

Vertical arcs are labeled with event overhead time.



This space-time graph depends only on the code and the input initialization of the simulation. The events executed, the simulation times at which they occur, and the causal relations among them are all deterministic. So this graph is the same regardless of timing, synchronization, platform or runtime configuration.



The numbers here represent wall clock times on a particular (real or idealized) platform.

The **blue** numbers on the events represent the wall clock time taken to execute an event. I have included a range of event execution lengths in this diagram, anywhere from 45 to 200 units of time.

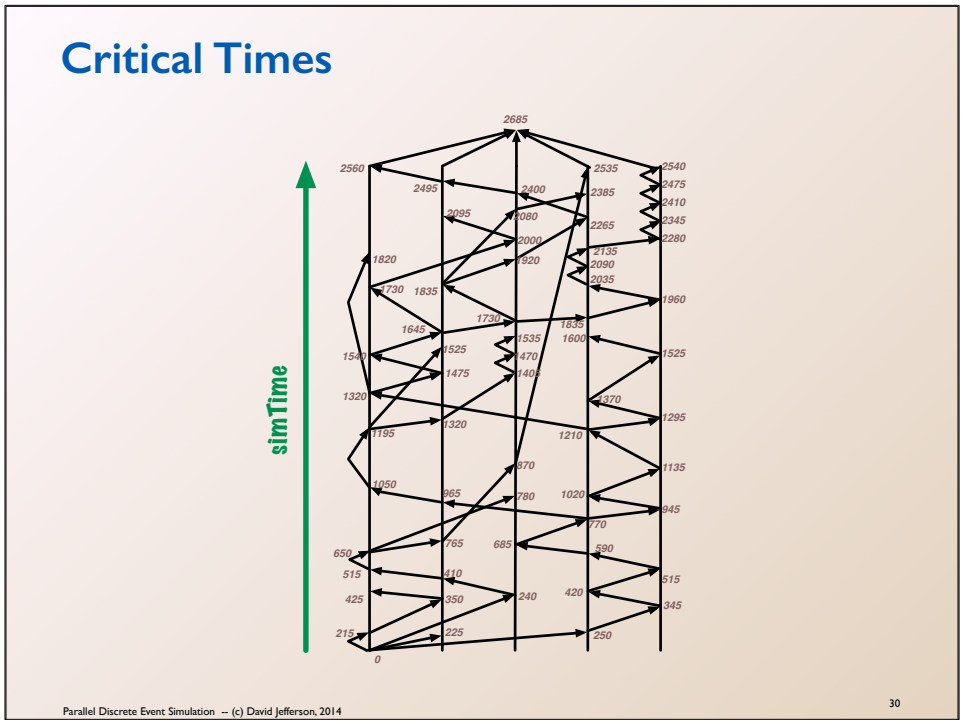
The **red** numbers on the arcs between events in the same LP represent event overhead, i.e. all of the computation that is not direct execution of events. It includes synchronization, storage management, etc. In this case I have made the arbitrary assumption that that is a constant 10 units of time everywhere.

The **green** numbers on the arcs that represent event messages indicate the message latencies, i.e. the wall clock time required to transport a message from one process to another. In this case I have made the arbitrary assumption that it takes 25 time units to send a message between two neighboring LPs, and more time if the communicating LPs are more distant.

In each case these numbers should either be estimates of the minimum times each of these should take, or the average times. Use minimum times if you wish to get an actual lower bound on runtime of the entire simulation. Use average times (considering message traffic effects and LP scheduling effects) if you want a somewhat more accurate estimate that is not strictly a lower bound.

When you consider execution on a particular platform with a particular (static) assignment of objects to nodes, then the actual event timings, event overheads, and message latencies are introduced, based on the best case performance on that platform and in that configuration.

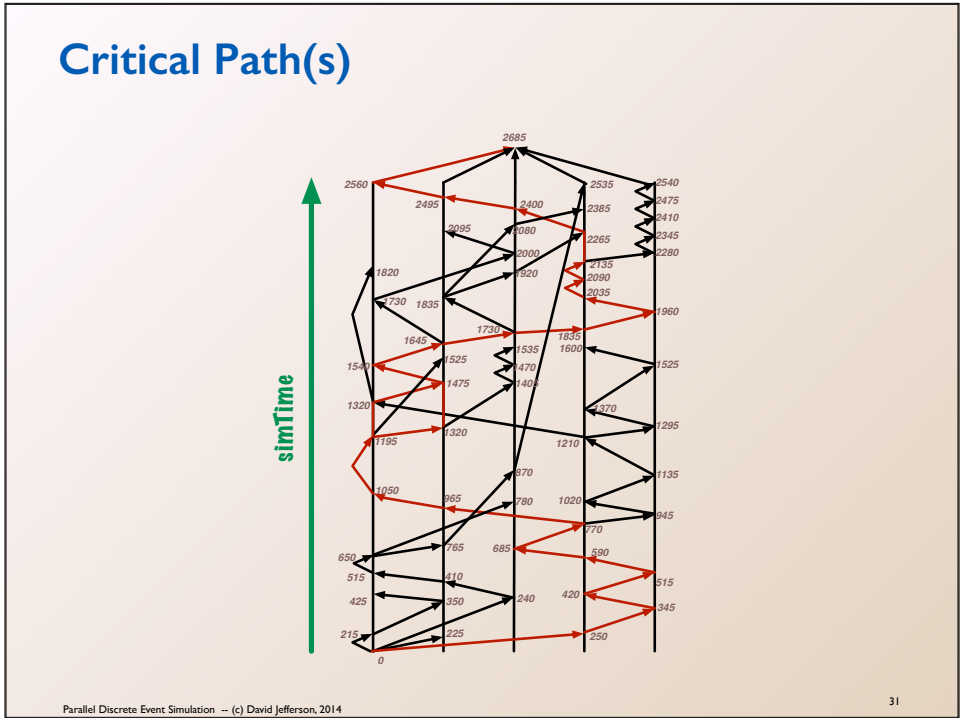
Note that this applies to STATIC configurations. If there is dynamic load reconfiguration of any kind, then this diagram does not capture such effects.



These critical times are for the start of the events that they are attached to. Thus, if an event has critical time 650, than it cannot possibly start execution until wall clock time 650.

We assume that when an event sends one or more messages, it does so at the end of its execution. Th

Note that in the diagram we have added a final fictitious termination event.

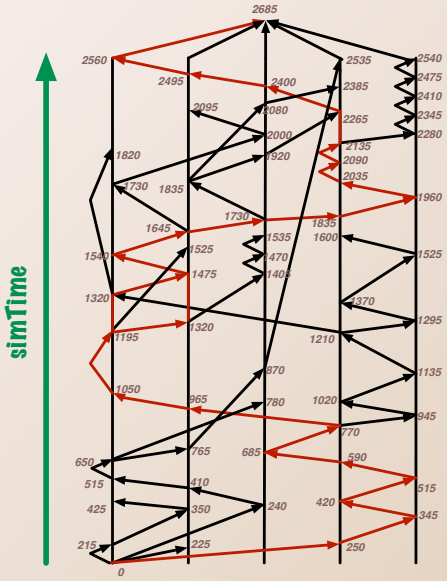


The critical path is not necessarily unique. There may be several, or many, paths that tie as the longest path. This diagram shows two paths of equal length. More generally, there may be many paths of near-equal length, so that they are all co-critical.

The critical times and critical path can be calculated on the fly, during the simulation, in time linear in the length of the simulation, i.e. a constant overhead per event. So it can and should be done routinely during PDES simulations.

Co-critical Path(s)

- Many critical and near-critical paths -- co-critical paths -- in a large simulation.
- Performance improvement requires shortening all co-critical paths -- without lengthening others to criticality.
- Critical times and co-critical paths can be computed during a run with linear overhead
- In improving performance, consider:
  - code shared among many co-critical paths -- improve that shared code
  - objects shared among co-critical paths -- improve those objects
  - critical latencies between pairs of objects -- shorten the distances between those objects



Speed-up from parallelism

n	number of objects (LPs) in the simulation
p	number of processors used
T_{seq}	sequential execution time
T_{crit}	critical path length
T_p	actual measured time on p nodes of cluster
$S_{actual(p)}$	actual measured speedup from parallelism using p nodes
$S_{potential}$	potential speedup from parallelism
$S_{fraction(p)}$	fraction of optimal speedup achieved
$E_{procs(p)}$	processor efficiency; fraction of processor time used for (committed) event execution

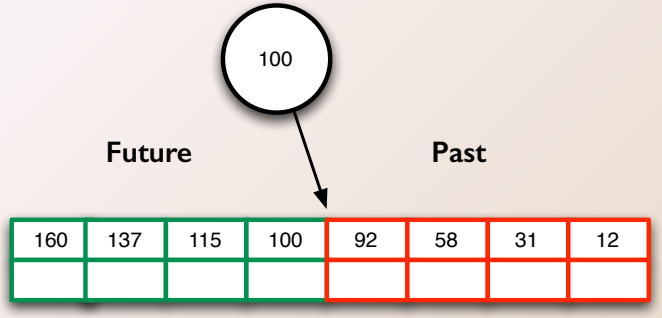
n	= 5		
p	= 5		
T_{seq}	= 4725		
T_{crit}	= 2685		
T_p	= 3200 (suppose)		
$S_{actual(p)}$	= T_{seq} / T_p	= 4725/3200	= 1.48
$S_{potential}$	= T_{seq} / T_{crit}	= 4725/2685	= 1.76
$S_{fraction(p)}$	= $S_{actual(p)}$ / $S_{potential}$	= 1.48/1.76	= 0.84
$E_{procs(p)}$	= T_{seq} / (T_p * p)	= 4725/(3200*5)	= 0.295

Optimistic
Parallel Discrete Event Simulation
Algorithms

Optimistic Paradigm

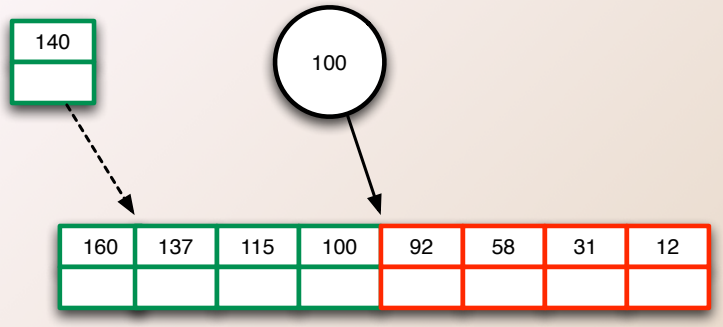
- **No static restrictions**
  - Any object can send an event message to any other
  - No graph structure or "channels" required, just a flat name space
  - Order preservation during message transport not required
  - Dynamic object creation and destruction permitted
  - No lookahead information required
- **Events are considered reversible -- they can be undone (rolled back).**
- **Rollback is the fundamental synchronization primitive, not process blocking.**
- **Optimistic PDES first introduced with the Time Warp algorithm in 1984 (RAND Corp., JPL)**
- **Many variants now, all descended from TW**

Fundamental idea



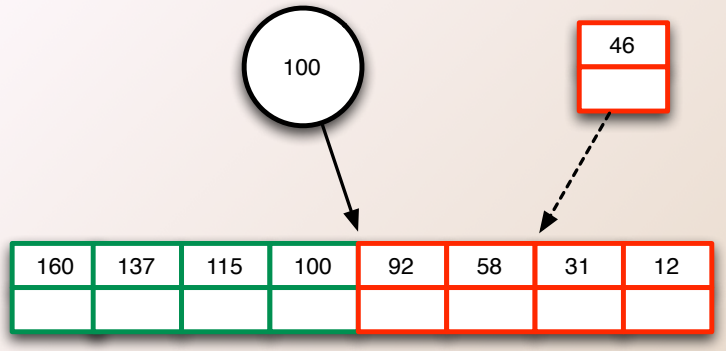
- Assume infinite storage (for now)
- Save all event messages, both processed and unprocessed
- Object simTime is index into message queue
- Process all events in simTime order, blocking *only* when all events in the queue have been processed

Fundamental idea



If an event message arrives in the "future", just enqueue it in sorted order and continue processing events

Fundamental idea



If an event message arrives in the "past":

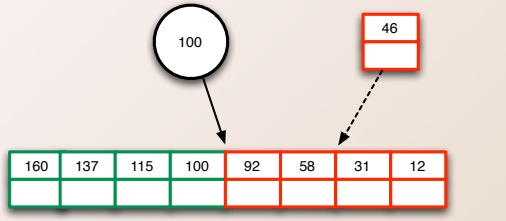
1) Interrupt processing of the current event (if any)

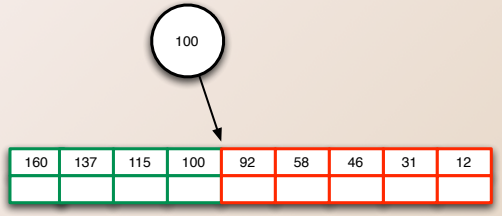
2) Enqueue event message in sorted order

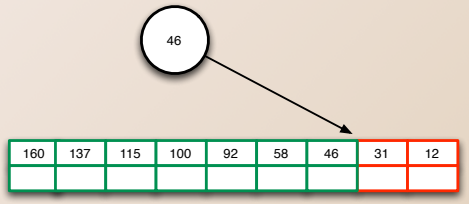
3) Roll back to "before" the events that should not have been done, and continue from there

Synchronization using

If an event message arrives in the "past", roll back to "before" the events that should not have been done, and restart from there







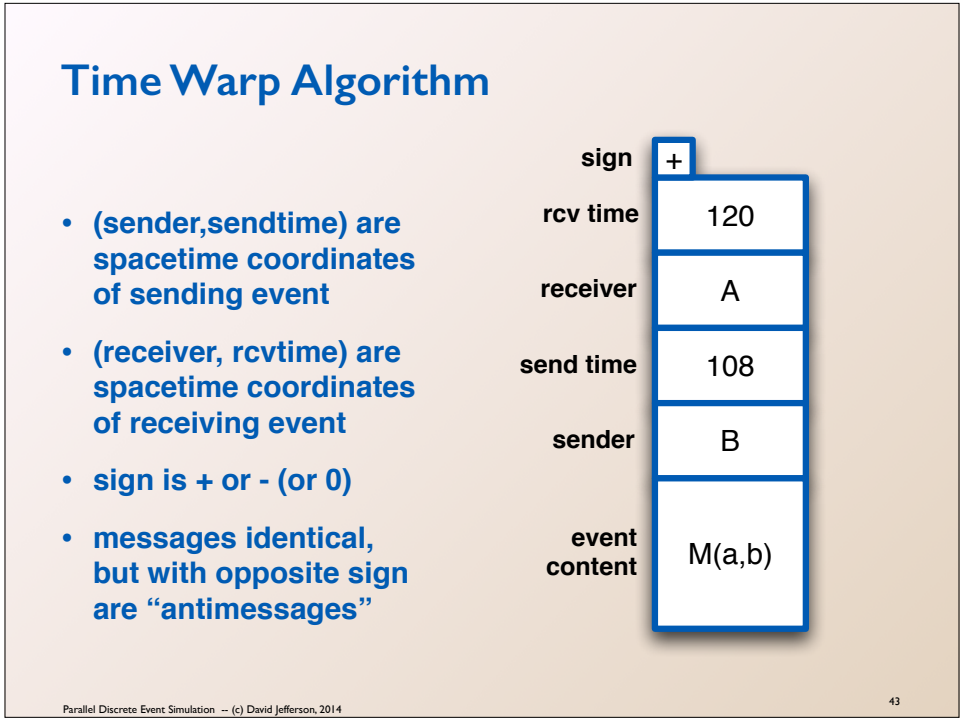
This shows the list of challenges we have to overcome for the Time Warp algorithm, or any optimistic PDES algorithm, to be practical. Most people with a background in asynchronous distributed computation who have not seen optimistic PDES algorithms are inclined to believe that doing this is either literally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

The Time Warp Algorithm

Brief History of Time Warp

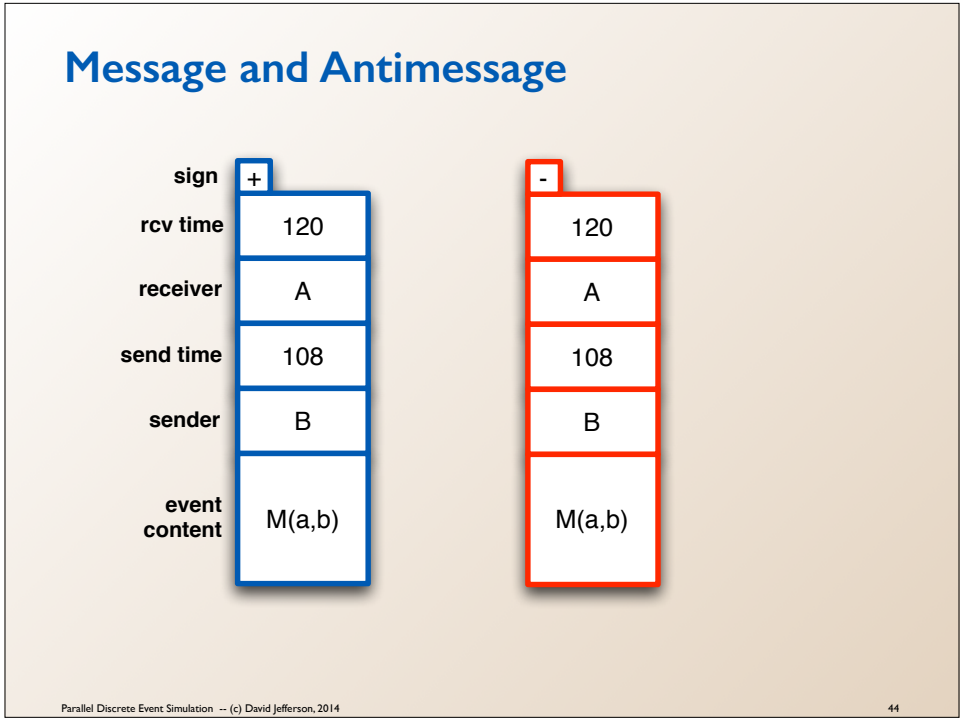
- Invented by myself and Henry Sowizral at the RAND Corp. in 1983
- Implemented and studied at JPL on Caltech Hypercubes from 1985-1991
- Many other contributors in the early years (Brian Beckman, Peter Reiher, Anat Gafni, Orna Berry, Richard Fujimoto, …)
- First journal publication:

  Jefferson, David, "Virtual Time", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 7, 3, pp. 404-425, July 1985



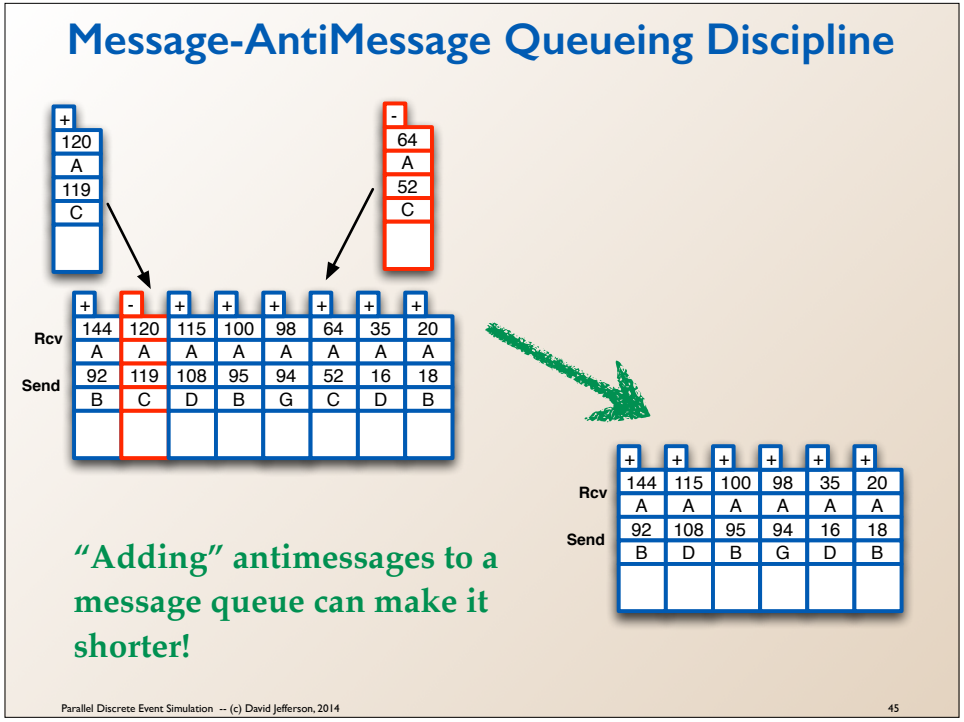
An event message in the TW algorithms holds the spacetime coordinates of the sending event (B,108) and the spacetime coordinates of the receiving event (A,120). It also has as sign, + or -, whose purpose will become apparent.

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



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

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



This is a "before" and "after" diagram. The message queue before (which happens to be an input queue since it is sorted by receive time) has 8 messages in it. (One of those messages is negative--that can happen, although it is infrequent).

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

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

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

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