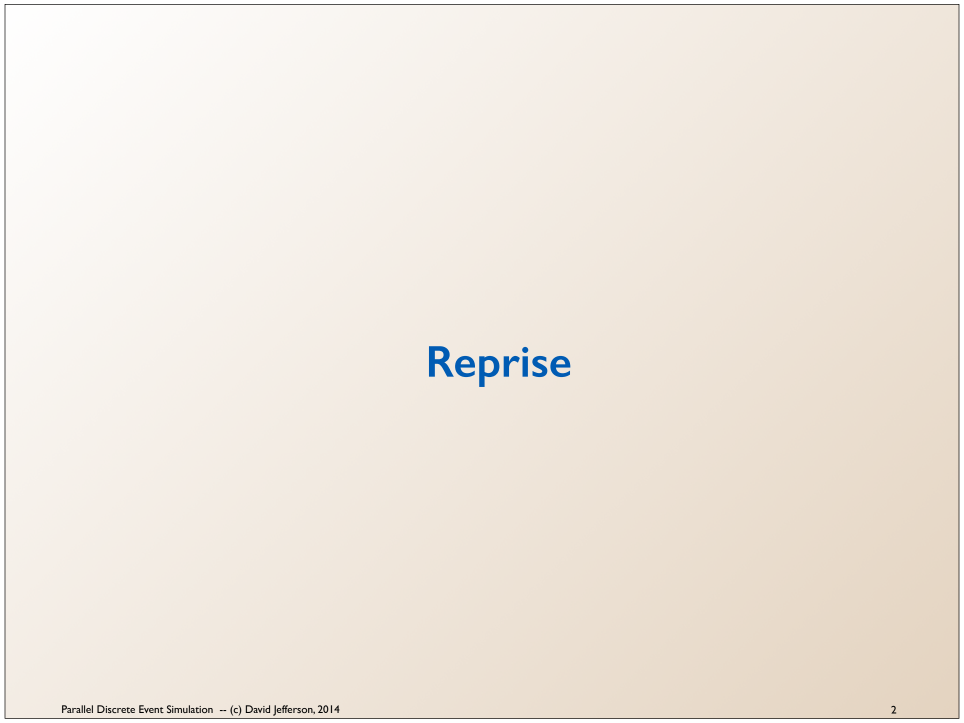
Parallel Discrete Event Simulation Course

#5

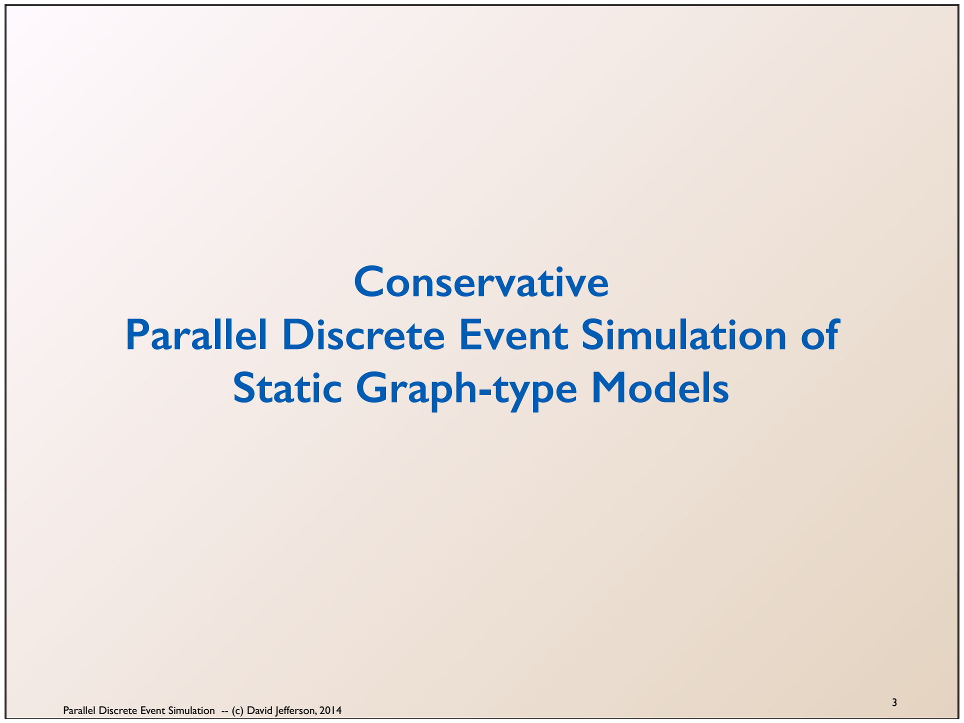
David Jefferson
Lawrence Livermore National Laboratory
2014

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

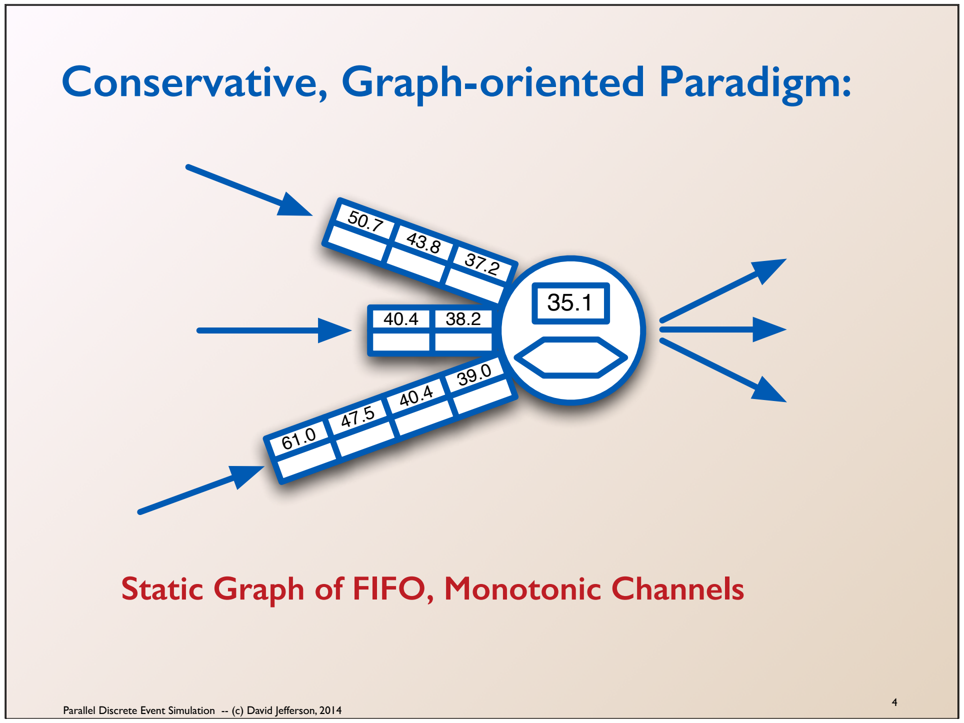
Release Number: LLNL-PRES-650779



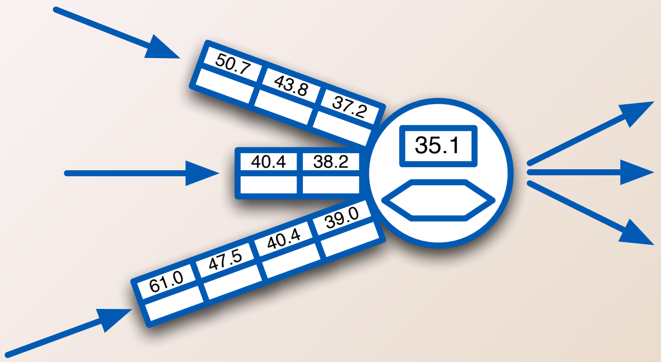
Reprise



Conservative
Parallel Discrete Event Simulation of
Static Graph-type Models



Conservative, Graph-oriented Paradigm:



Static Graph of FIFO, Monotonic Channels

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

4

Core conservative graph-oriented algorithm:

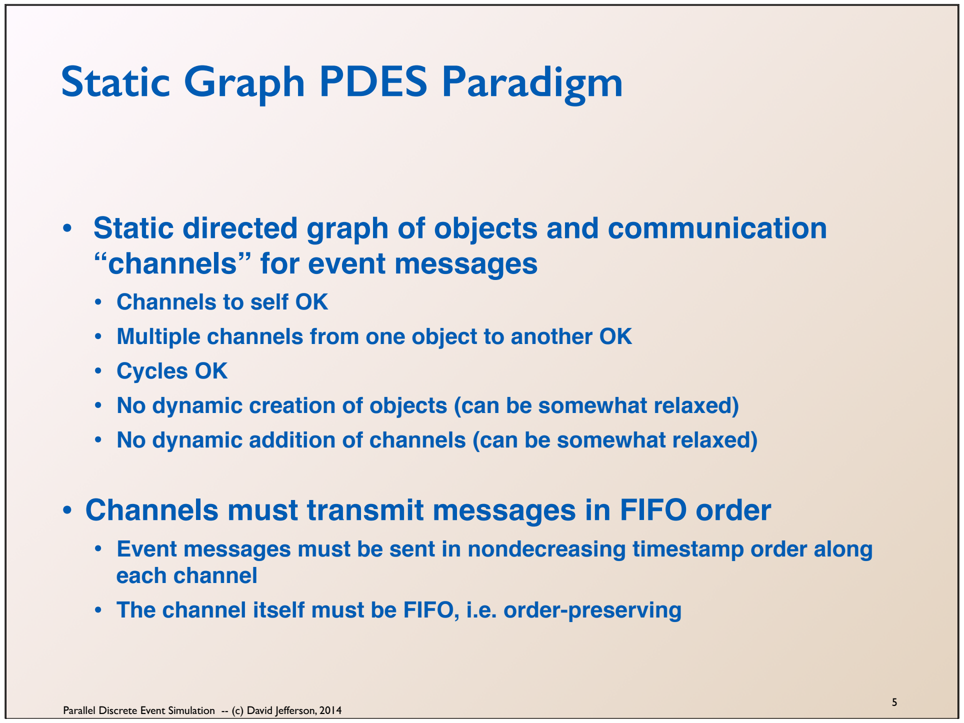
1) Each object has a static number of input queues, one for each incoming arc in the static communication graph, i.e. one for each other object that ever sends event messages to it.

a) Messages from an object to itself require an input channel
b) An object can have two or more channels to another object or itself -- that is permitted. But the number of such channels must be static (for the time being).
c) Neither new channels nor new objects can be created. (This condition can be partially relaxed later.)
d) Channels can be effectively *deleted* with no problems by the sender using the simple trick of sending a null message with a time stamp of ∞. That is a guarantee that no further events will be sent along that channel, so it might as well be destroyed.
e) Whole objects are effectively deleted if all of their incoming channels are deleted.

2) Because of the FIFO property in each channel and the restrictions that the sender must send event messages in nondecreasing time stamp order in each channel (i.e. no timestamp inversions within a channel) we can generalize as follows:

As long as all incoming queues are nonempty, then the unprocessed message with the lowest time stamp that will *ever* be received by this object is at the head of one of the input queues, the one with the minimum time stamp. (Of course, there may be ties, so adjust accordingly.)

3) So the naïve algorithm is:
a) choose the minimum timestamp message from across all of the input queues and execute it
b) block if any queues are found to be empty until they are all nonempty

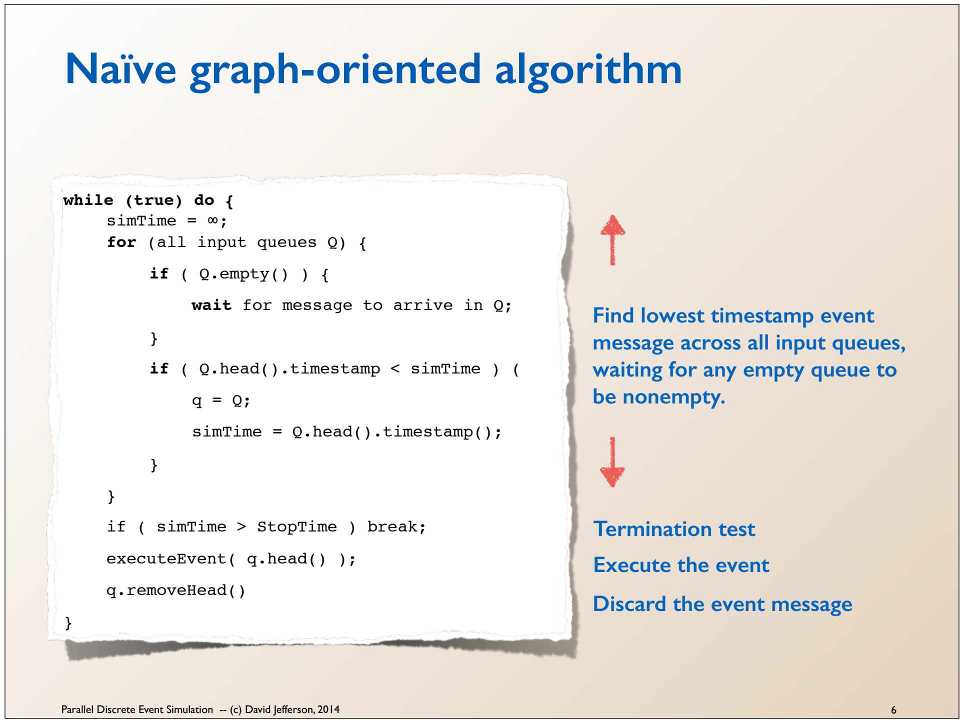


Static Graph PDES Paradigm

- Static directed graph of objects and communication "channels" for event messages
  - Channels to self OK
  - Multiple channels from one object to another OK
  - Cycles OK
  - No dynamic creation of objects (can be somewhat relaxed)
  - No dynamic addition of channels (can be somewhat relaxed)
- Channels must transmit messages in FIFO order
  - Event messages must be sent in nondecreasing timestamp order along each channel
  - The channel itself must be FIFO, i.e. order-preserving

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

5



Naïve graph-oriented 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;
    executeEvent( q.head() );
    q.removeHead()
}
```

Find lowest timestamp event message across all input queues, waiting for any empty queue to be nonempty.

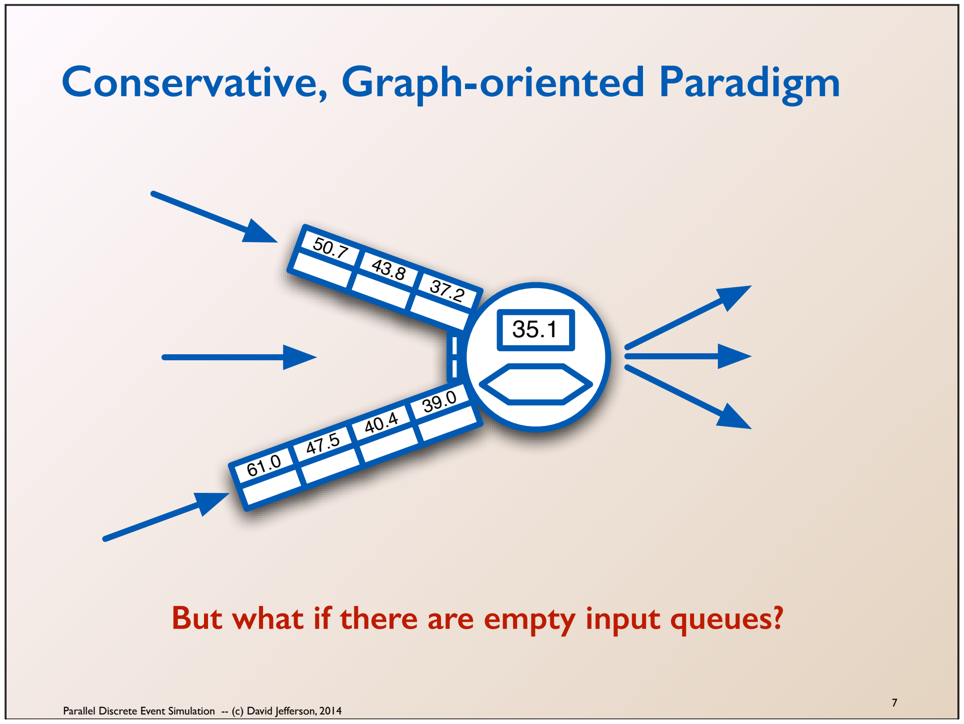
Termination test

Execute the event

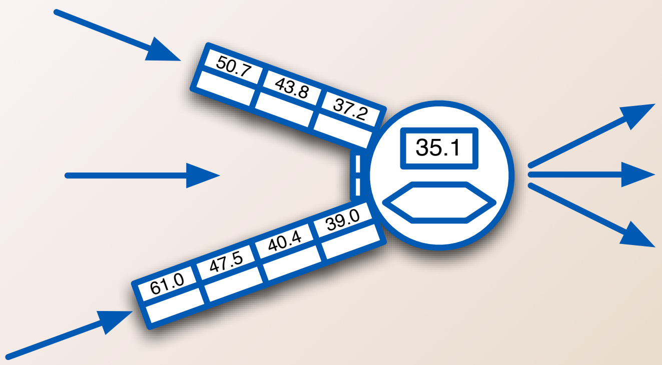
Discard the event message

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

This algorithm is “naïve” because it does not use any lookahead information and is subject to deadlock.



Conservative, Graph-oriented Paradigm

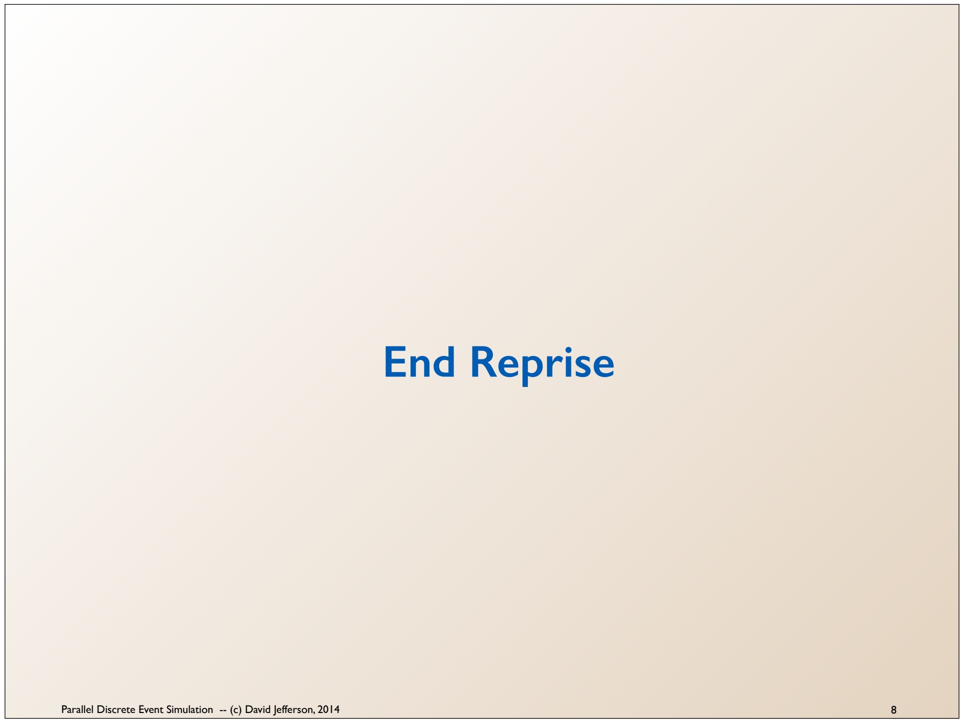


But what if there are empty input queues?

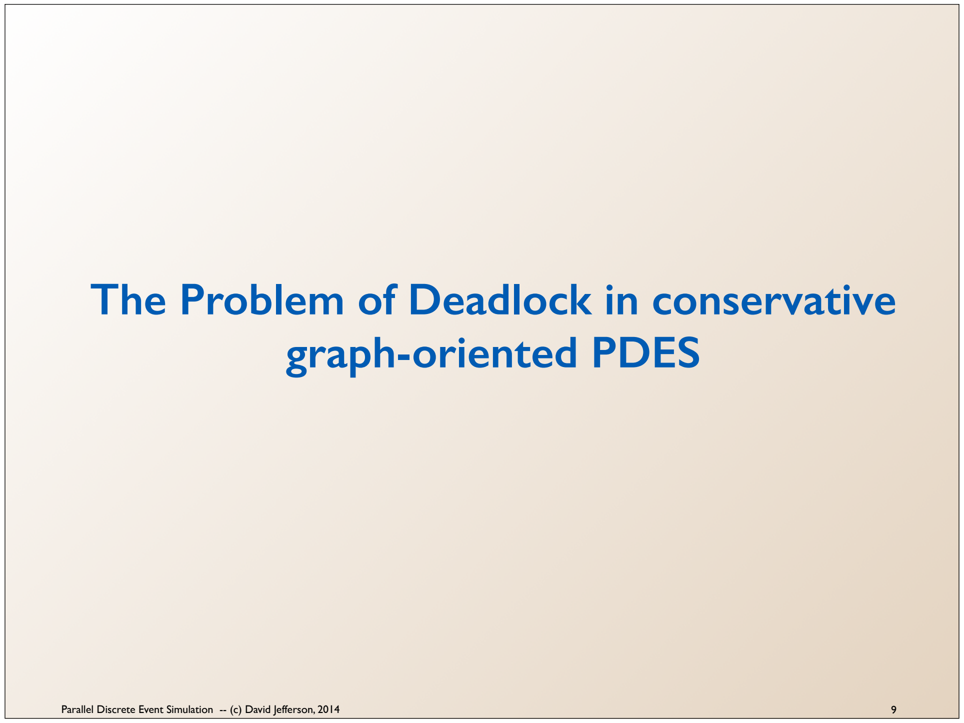
Core conservative graph-oriented algorithm:

What if one or more of the input queues is empty? Then there is no way to know whether the sender at the other end of that/those channel(s) will send a message with a smaller time stamp than those on the non-empty channels.

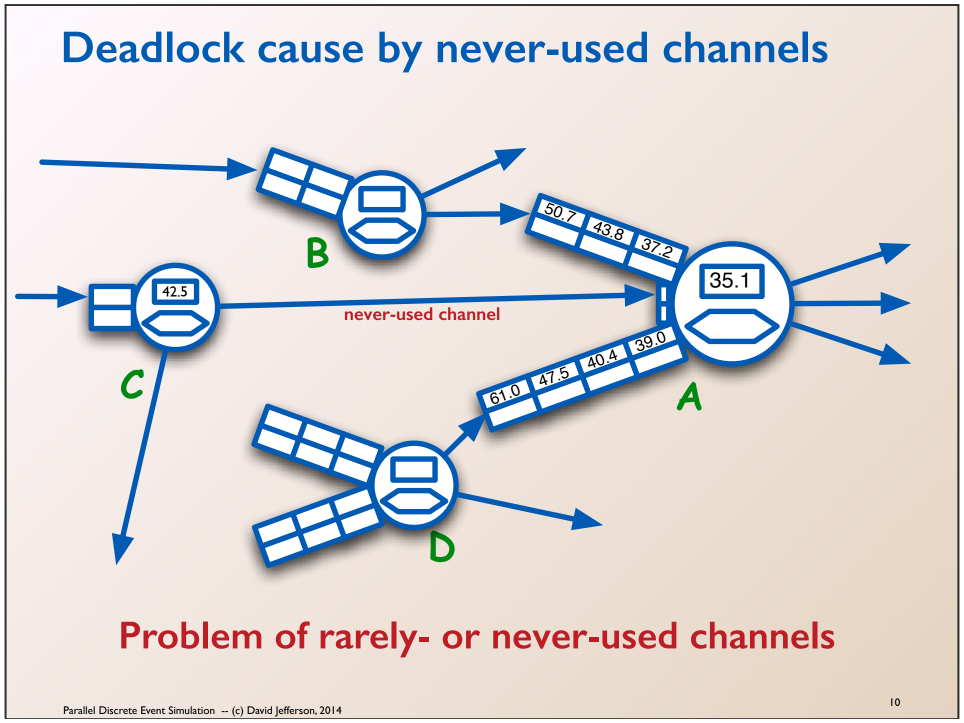
In that circumstance the simulator must block this process/object until such time as at least one more message arrives to make all the empty queue(s) nonempty.



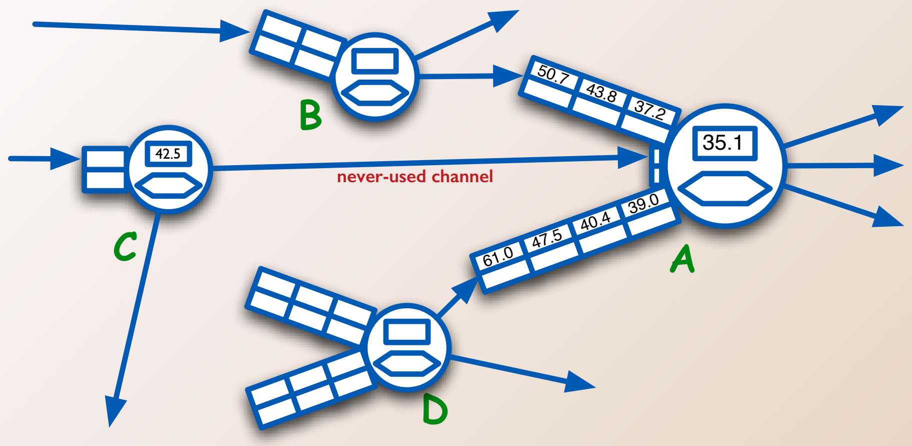
End Reprise



The Problem of Deadlock in conservative graph-oriented PDES



Deadlock cause by never-used channels



Problem of rarely- or never-used channels

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

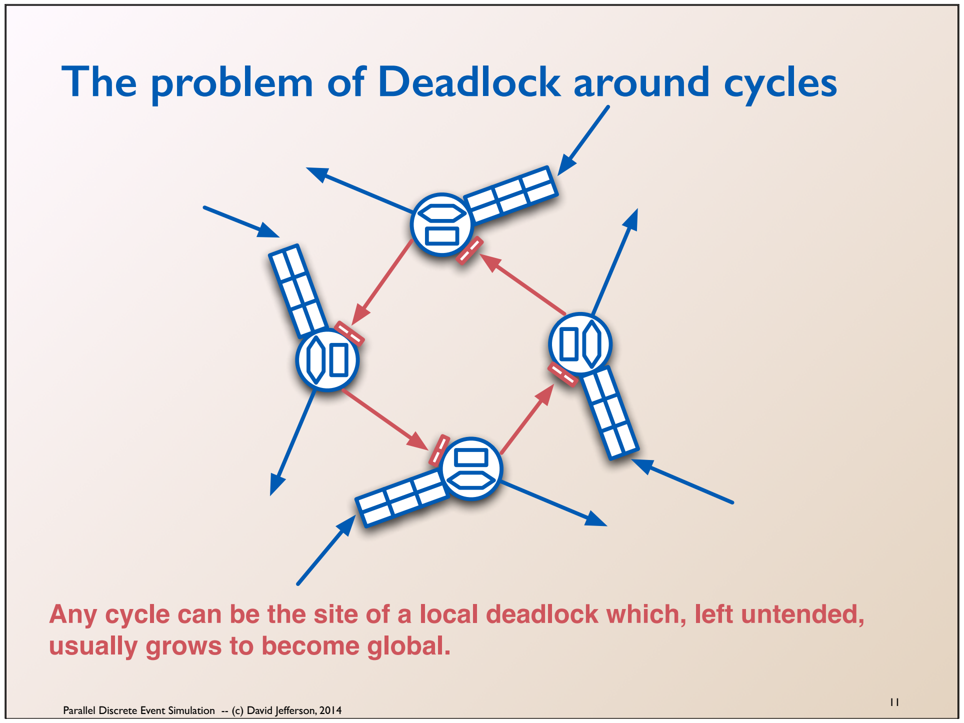
10

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!

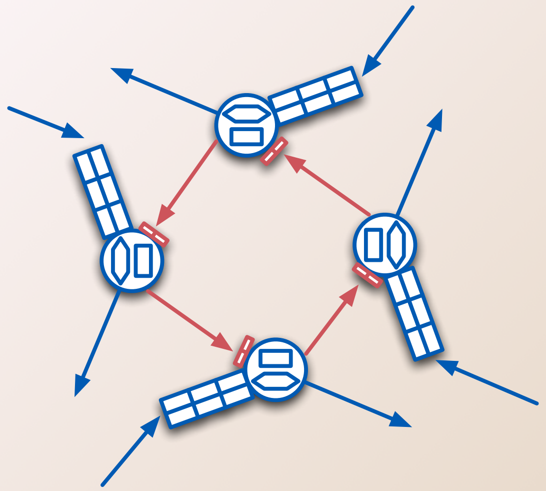
For this reason, one cannot get around the requirement of a static interaction graph just by deciding to choose as the complete graph connecting all objects to one another in both directions. Besides requiring O(n**2) storage, that would leave most queues empty most of the time, drastically increasing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

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 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 problem of Deadlock around cycles



Any cycle can be the site of a local deadlock which, left untended, usually grows to become global.

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

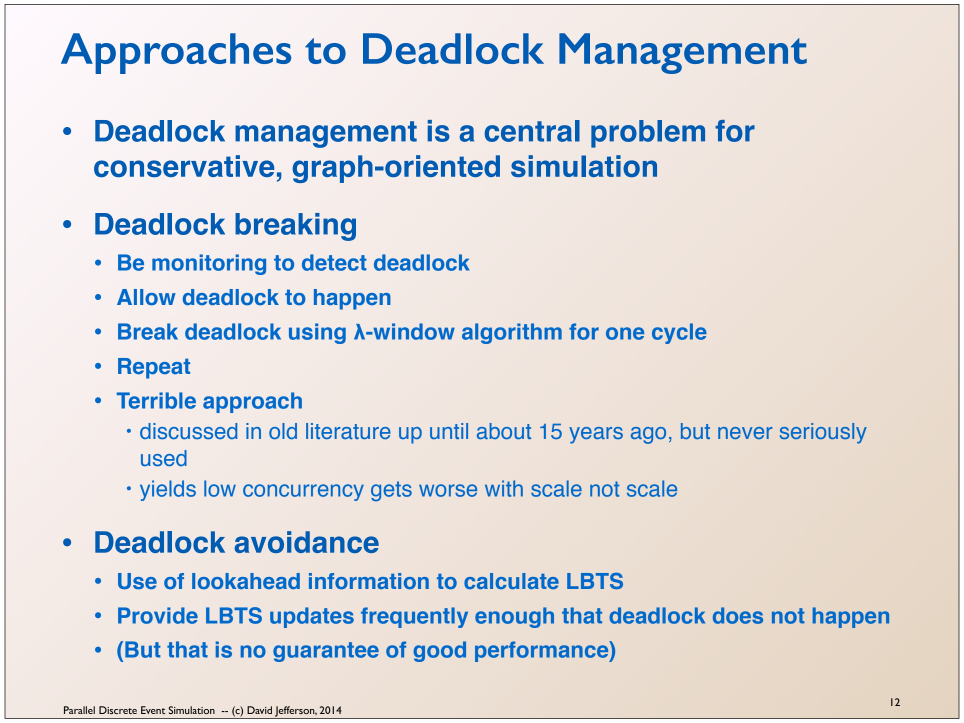
11

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*.

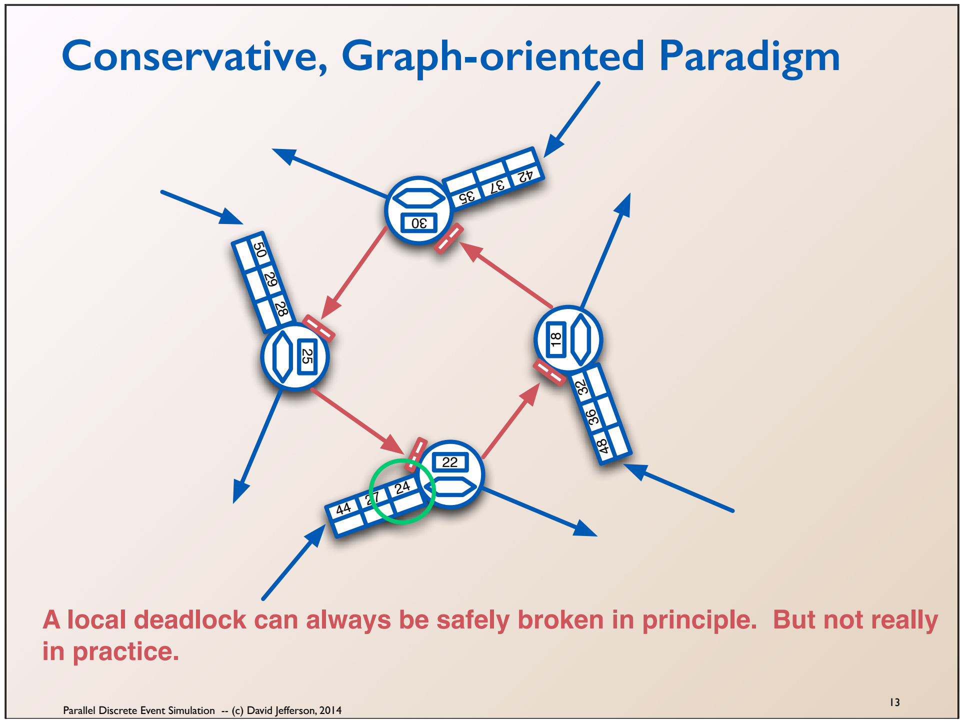


Approaches to Deadlock Management

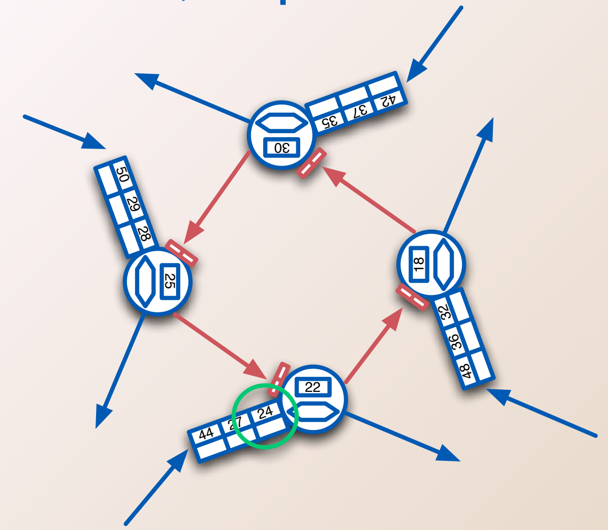
- **Deadlock management is a central problem for conservative, graph-oriented simulation**
- **Deadlock breaking**
  - **Be monitoring to detect deadlock**
  - **Allow deadlock to happen**
  - **Break deadlock using λ-window algorithm for one cycle**
  - **Repeat**
  - **Terrible approach**
    - discussed in old literature up until about 15 years ago, but never seriously used
    - yields low concurrency gets worse with scale not scale
- **Deadlock avoidance**
  - **Use of lookahead information to calculate LBTS**
  - **Provide LBTS updates frequently enough that deadlock does not happen**
  - **(But that is no guarantee of good performance)**

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

12



Conservative, Graph-oriented Paradigm



A local deadlock can always be safely broken in principle. But not really in practice.

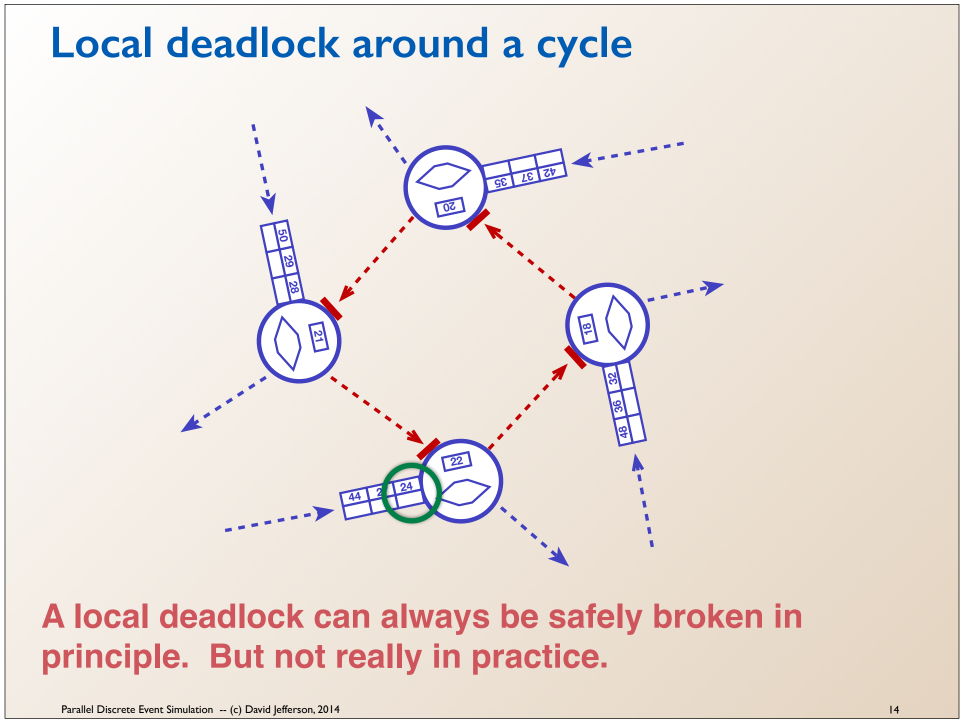
Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

13

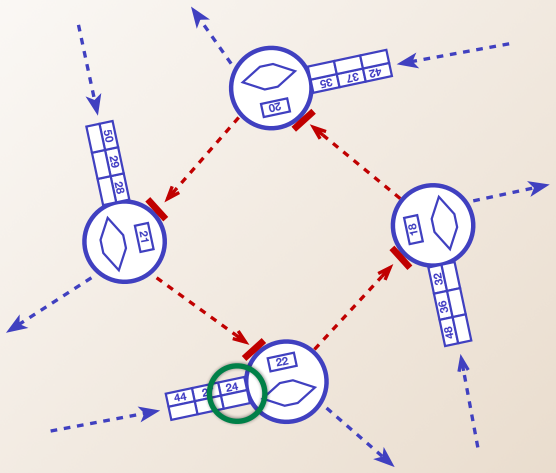
A cycle of empty event queues is only a deadlock according to the naïve paradigm in which an object always blocks until it all of its queues are nonempty. In general a not-so-naïve simulator can in principle break such a local deadlock safely by just selecting the lowest time stamped event message among all of the objects involved in the cycle, and allowing the object it belongs to to processing it. The message with time stamp 24 can be safely processed in this example, in spite of the fact that another queue in the same object is empty.

The problem is that the simulator cannot generally recognize a local deadlock when it happens. There is no fast, scalable algorithm for this. (For any given model, however, one might create such an algorithm as a special case.)

The fact that it is safe to allow an object to execute even though it has an empty queue associated with a channel-to-self is just a special case of this more general point. Since that case is trivially detectable based on local information only, it was worth singling out.



Local deadlock around a cycle



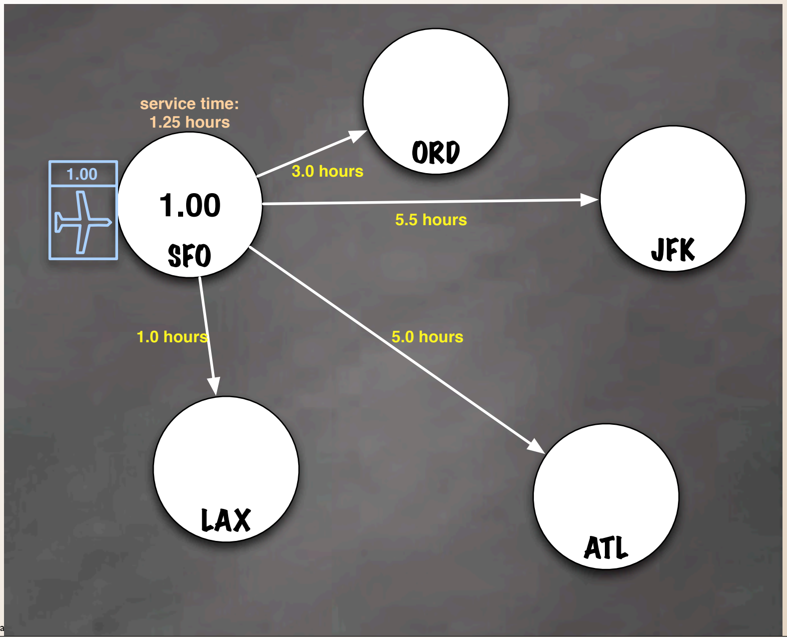
A local deadlock can always be safely broken in principle. But not really in practice.

A cycle of empty event queues is a local deadlock according to the naïve conservative paradigm in which an object always blocks until it all of its queues are nonempty. In general a not-so-naïve simulator can in principle break such a local deadlock safely by just selecting the lowest time stamped event message among all of the objects involved in the cycle, and allowing the object it belongs to to processing it. The message with time stamp 24 can be safely processed in this example, in spite of the fact that another queue in the same object is empty.

The fact that it is safe to allow an object to execute even though it has an empty queue associated with a channel-to-self is just a special case of this more general point. Since that case is trivially detectable based on local information only, it was worth singling out.

The problem is that the simulator cannot generally **recognize** a local deadlock when it happens. There is no fast, scalable algorithm for this in general. (For any give model, however, one might create such an algorithm as a special case. But that amounts to building a special-purpose simulator.)

Airplane Service Event at SFO

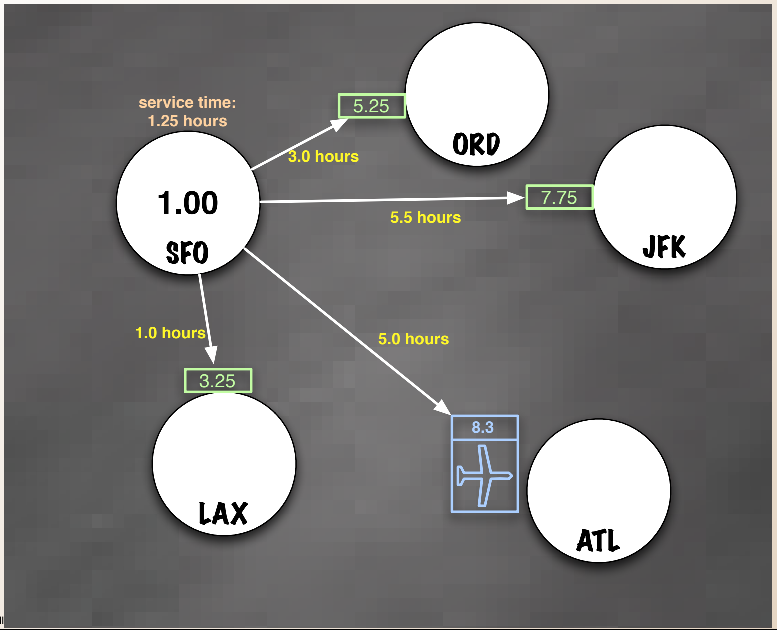


An Airplane_Service event is processed by the SFO object at 1.00 hours simTime.

It takes **at least 1.25 hours** to turn a plane around on the ground (unloading and loading passengers and baggage, fueling, ground checks).

It take **at least 3 hours** for ***the fastest plane*** to fly to ORD, and **at least** the listed times to fly to the other airports. (Note that a particular plane may fly slower.) So we can calculate the **lookahead** for all 4 other airports by summing minimum Service Time and minimum Travel Time.

Lookahead and subsequent events



Parall 16

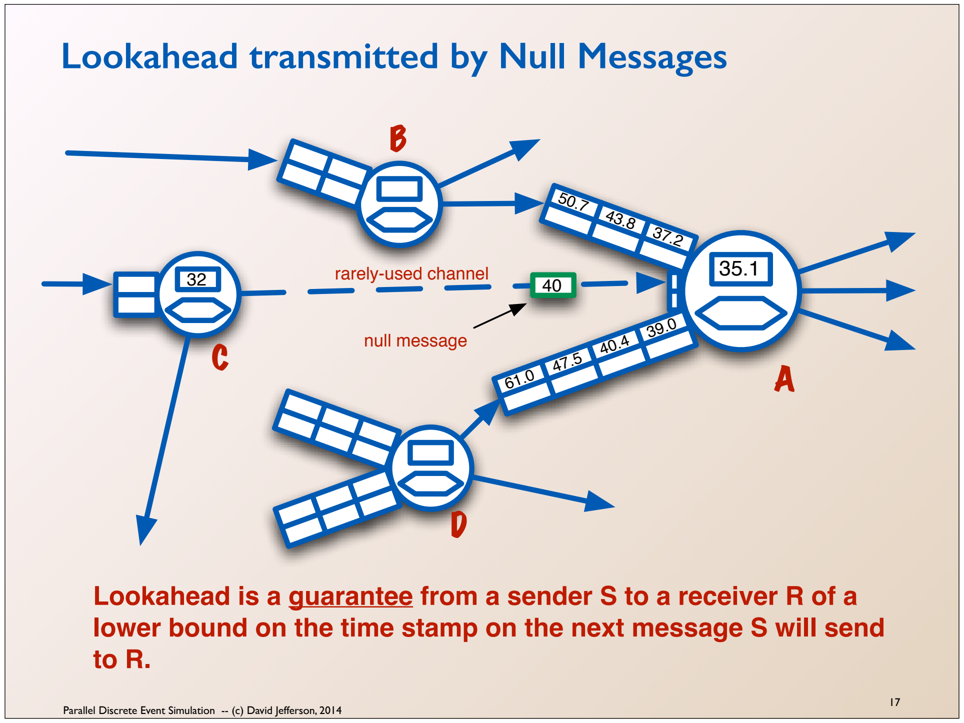
It turns out that the plane at SFO was heading out to the ATL airport.

The processing of the event at SFO leads to sending null messages to ORD, JFK, and LAX indicating the **minimum** time at which they can possibly get another plane arriving from SFO.

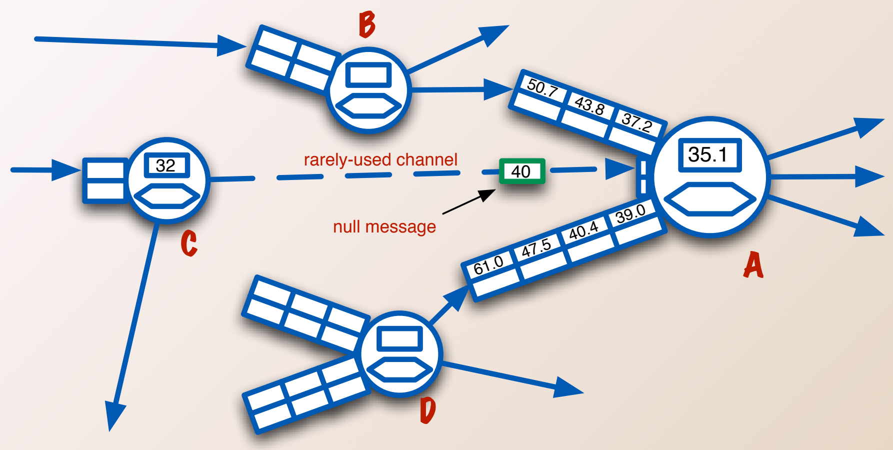
It also leads to scheduling an airplane arrival event at ATL for simTime 8.3 hours.

Note that this is **later** than the minimum possible time for the plane to arrive at ATL, which would be at 1.00 + 1.25 + 5.0 = 7.25. The processing of the event at SFO accounted for some unexpected (random) ground delays and flight time delays of over another hour.

A complication arises if it is possible in the model for two planes, P1 and P2 to both leave SFO for ATL, with P1 leaving before P2 but arriving after P2. That is an example of an "inversion" and the modeler either has to send the event message for P2 down a different channel (i.e. requiring at least two event message channels from SFO to ATL) , or it has to delay sending the event message for P1 until after it sends the event message for P2. Neither solution is very elegant. Both require that when P1 is ready to depart the model anticipates that another plane P2 may depart later but arrive earlier, i.e anticipates the possibility of inversion. This is not a very clean modeling situation, so this example exposes a weakness in the graph-oriented synchronization paradigm: even though it technically can handle a bounded number of inversions, inversions mess up the logic. And if a second channel is chosen as the way to deal with the inversion, then it will probably be rarely used, but a stream of null messages has to be sent down that channel to keep the queue for it in ATL from going empty.



Lookahead transmitted by Null Messages



Lookahead is a guarantee from a sender S to a receiver R of a lower bound on the time stamp on the next message S will send to R.

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

17

The fundamental concept needed to deal with the problems of rarely- or never-used channels, and including cyclic local deadlocks, is "lookahead". Lookahead is a guarantee from a Sender to a Receiver (on a per-channel basis if there is more than one channel between them) of a **lower bound** on the time stamps on future event messages that will ever be sent down that channel. The sender is basically saying "I don't know when or if I will send another message down this channel. But if I ever do, I guarantee it will not have a time stamp lower than **t.**"

A message that carries only lookahead information is often called a **null message**, because it is like an event message in the way it is queued and the synchronization effects it has, but it does not actually call for any event method to be executed. It contains no information except a simTime value.In the above diagram, C is sending lookahead info in a null message to A, guaranteeing that C will never in the future send an event message to A with a time stamp less than 40. Note several things in this example:

1) C is sending this null message with a time stamp of 40, even though C is only at time 32 itself. C is thus "looking ahead" and making a guarantee about its own future behavior.

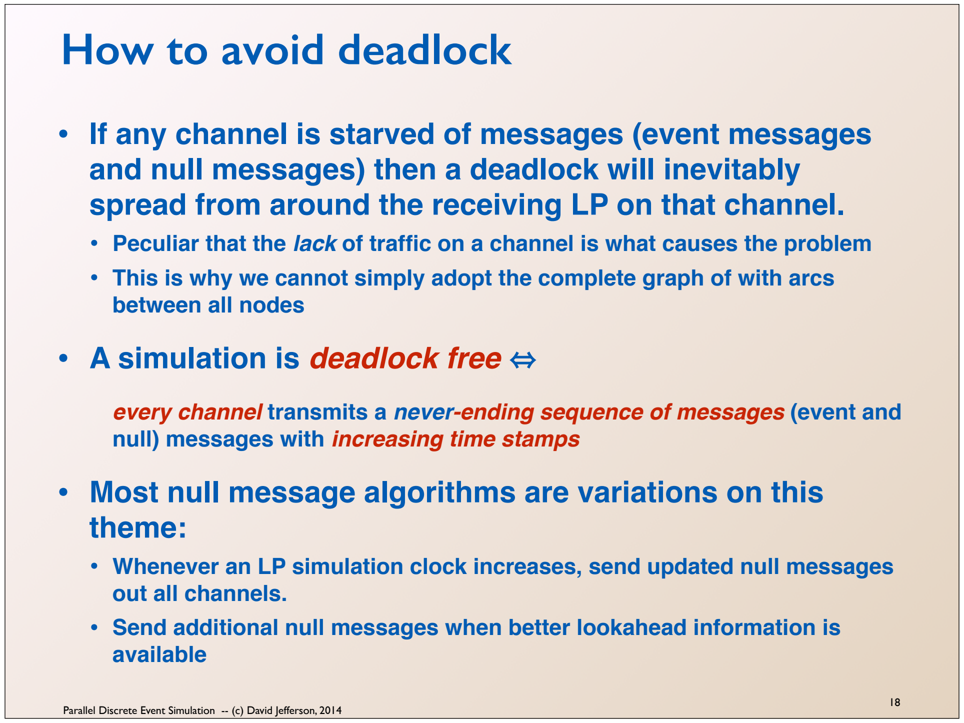
2) C can always send a null message with time 32 (its current simTime), but that is not useful "lookahead" information. In general, we want to C send the strongest lookahead information it can, i.e. a guarantee extending as far into the future as possible, and we want C to send it as early as possible in real time during the computation. That way A can proceed with processing its event messages as smoothly as possible with minimal synchronization pauses.

3) Lookahead information is not needed for *sequential* execution of any model. Nor is it needed for *optimistic* execution. It is, however, a practical requirement for all conservative parallel executions. And it imposes a burden on the programmers of conservative models to not only process and send event messages, which they have to do anyway, but also to explicitly calculate when they will ***not*** send event messages, and for how long in the future. This requires thinking about model behavior in a way that is not necessary with other styles of PDES execution.

4) A's queue does not have to be empty for C to send a null message; indeed C ordinarily will not know whether A's input queues are empty or not. If for some reason C sends multiple null messages in a row, then only the one with the highest time stamp matters, and the others can be thrown away when the latest one arrives.

5) Null messages can be sent on a "push" or "pull" basis. C can decide when to send them, e.g. whenever it is able to make a new, better estimate of its lookahead guarantee ("push"). Or, whenever the simulator finds A with an empty queue, the simulator can send a query to C to get the best lookahead information available from C at that moment ("pull").

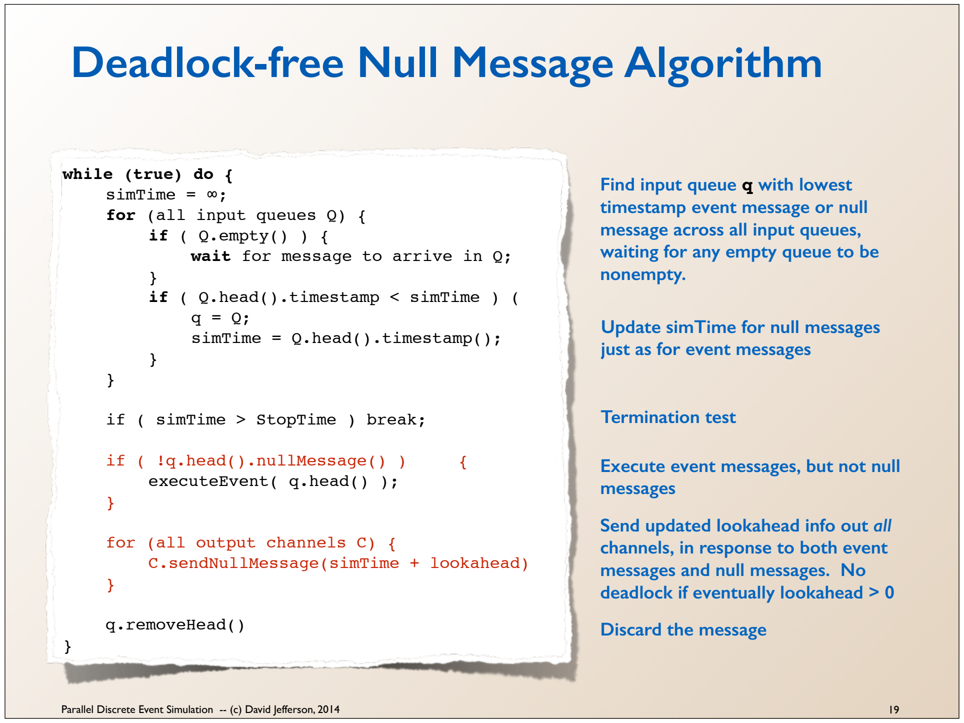
6) There are many other variations in the protocols for the exchange of lookahead information. Sometimes an object can choose a static interval in the future as its lookahead guarantee, and never change it, whereas other times the lookahead value can change with every event the sender executes. Sometimes the lookahead guarantee is the same for all output channels that C sends to, and sometimes it differs on different output channels. And sometimes, in a part of the computation that is way off the critical path, one can get away without sending lookahead information at all and just tolerate the fact that that part of the model will be slow because, being way off the critical path, it does not matter to overall performance. (This last strategy is risky, however. What is on and what is off the critical path of a simulation may change from run to run and from version to version.)



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 available**



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

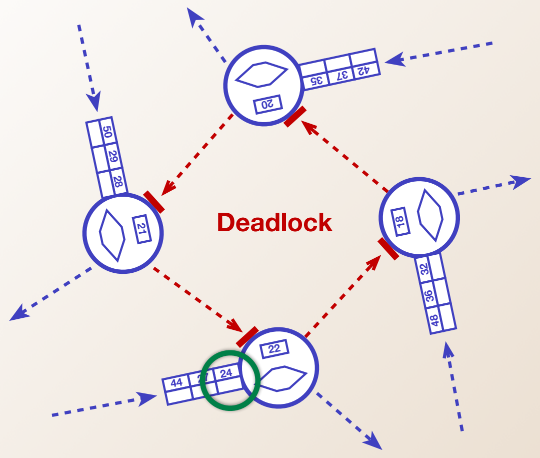
Discard the message

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.

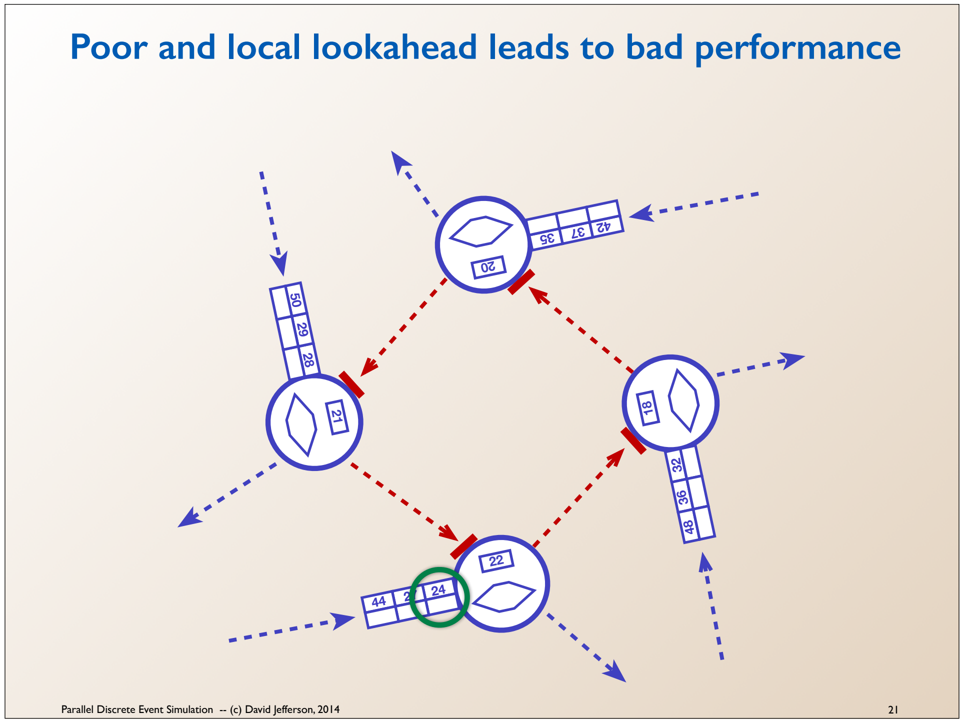
Poor lookahead leads to bad performance



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



Poor and local lookahead leads to bad performance

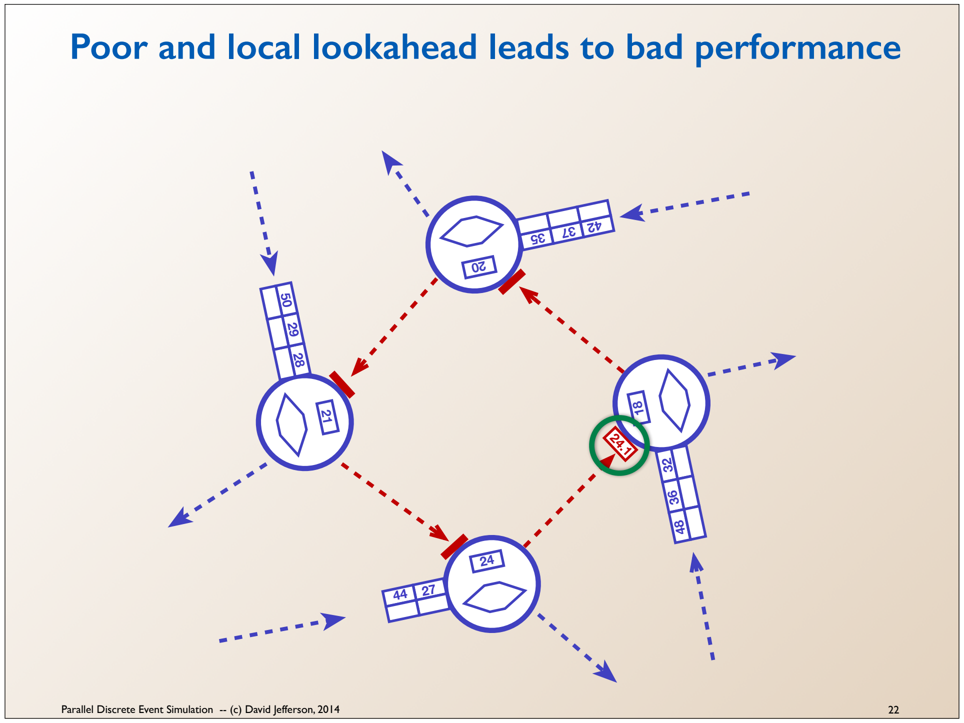
Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 21

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.

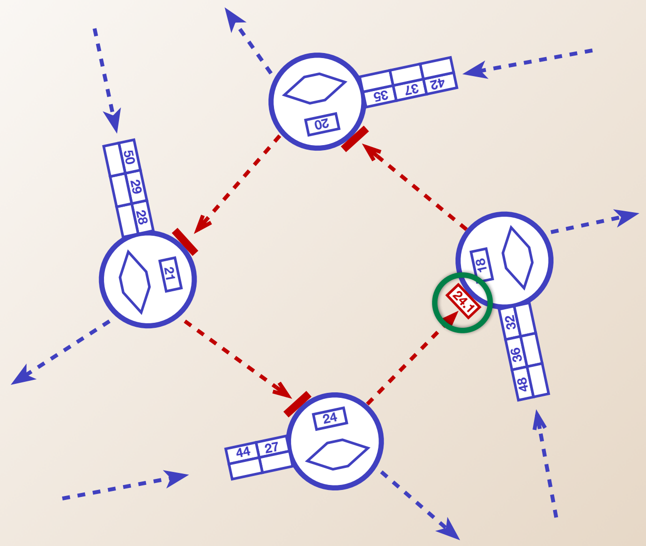
From now on, real event messages will be **blue**, and null lookahead messages will be **red**.

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.

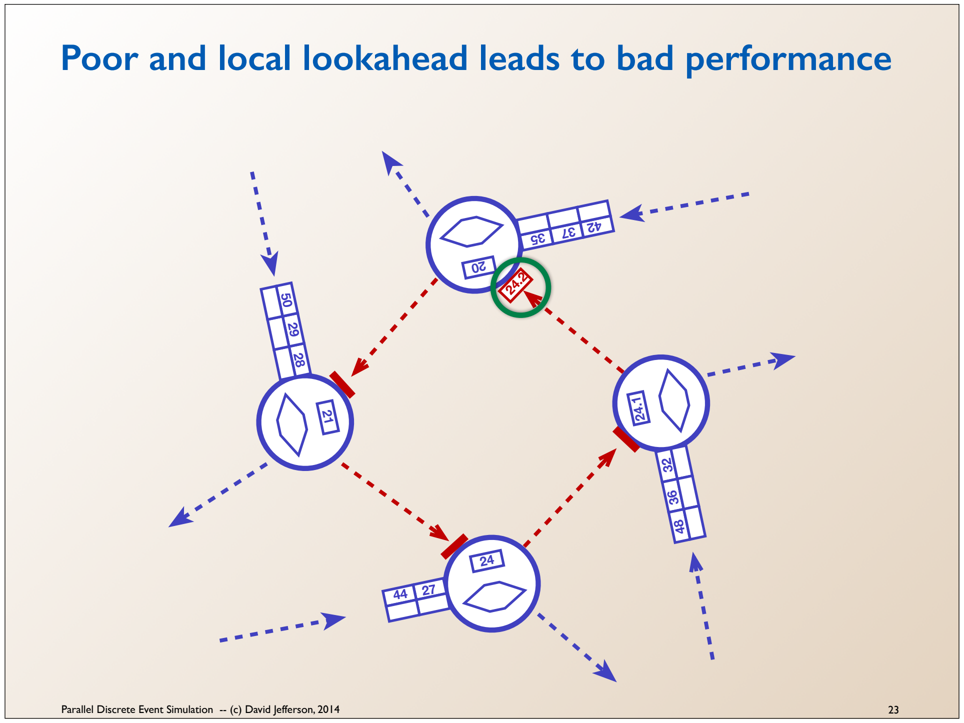


Poor and local lookahead leads to bad performance

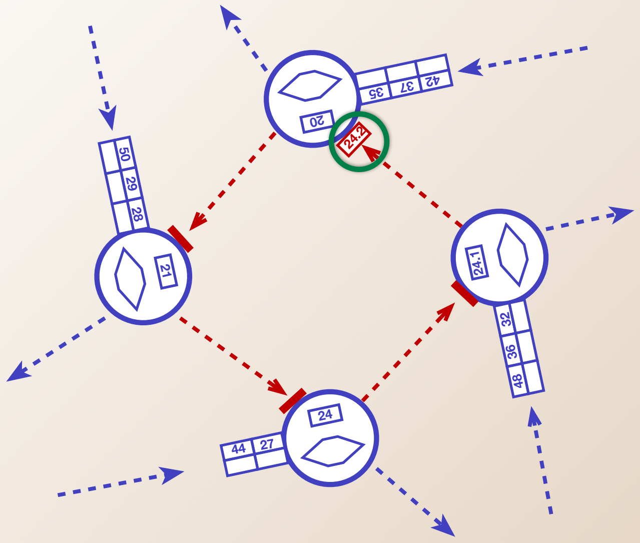


Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

22

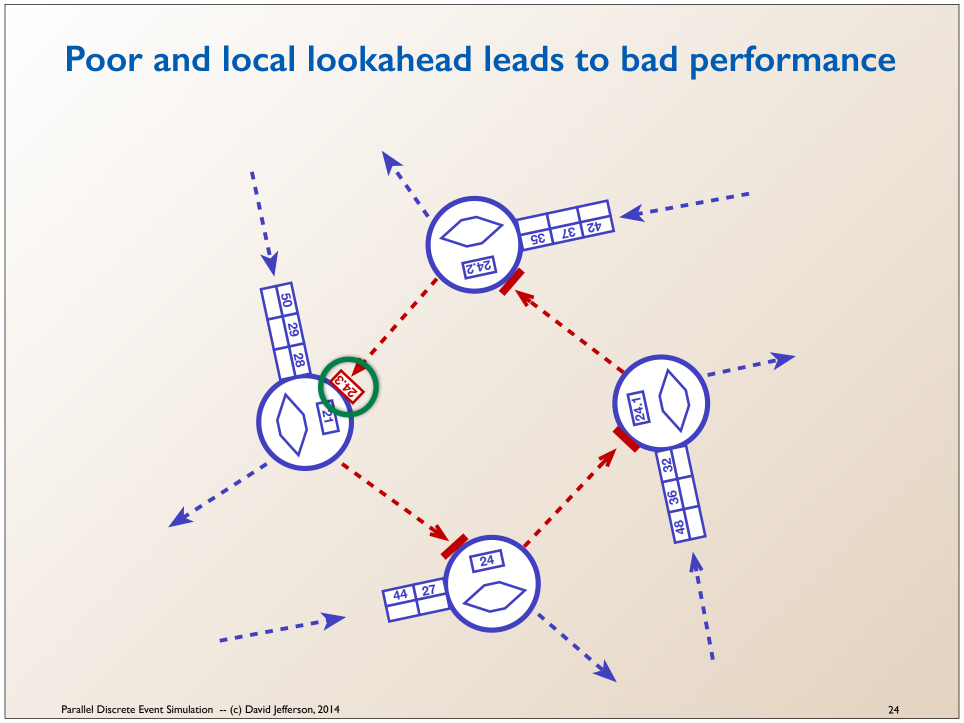


Poor and local lookahead leads to bad performance

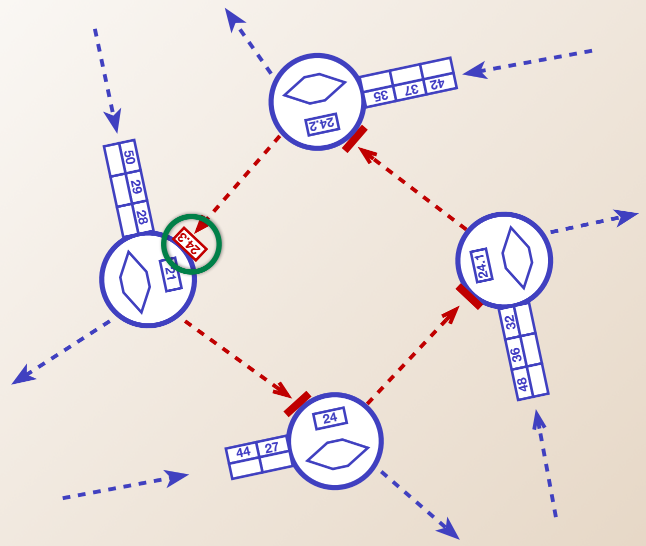


Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

23

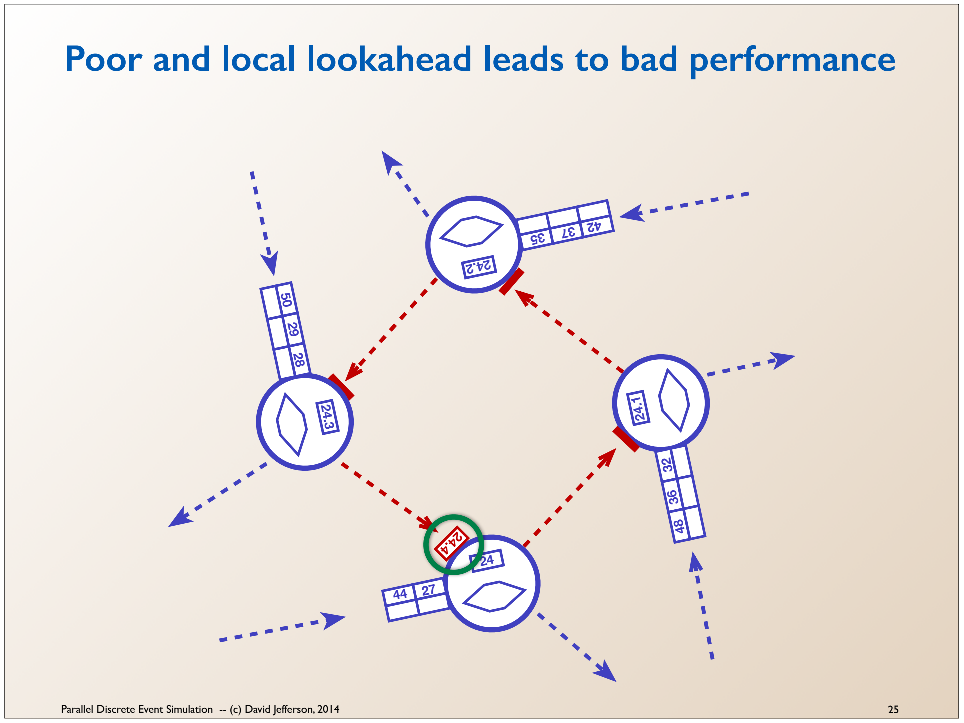


Poor and local lookahead leads to bad performance

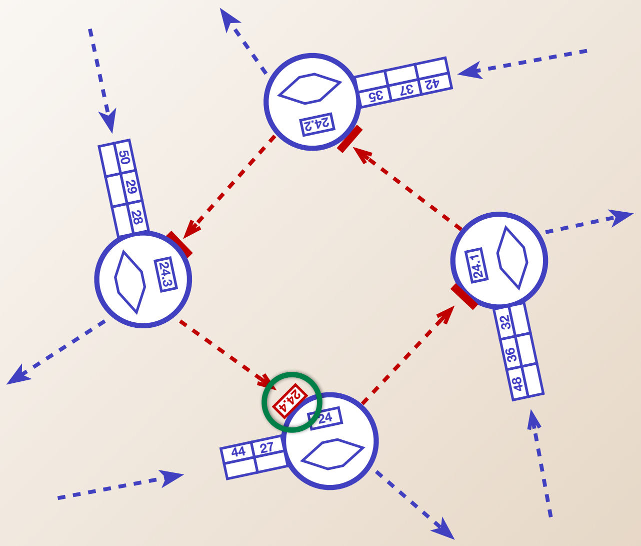


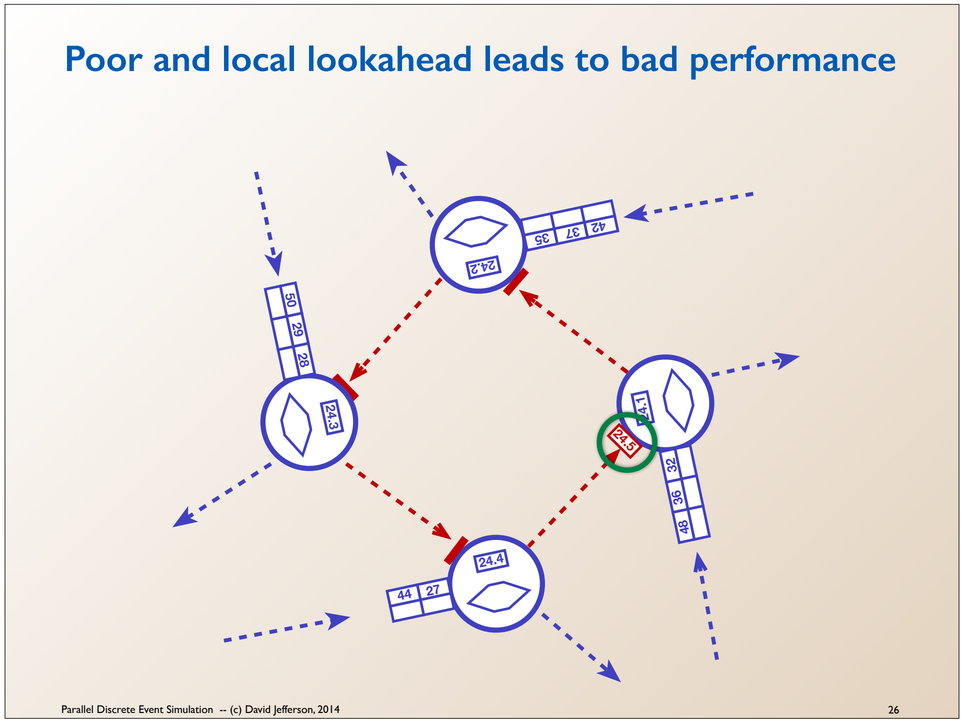
Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

24

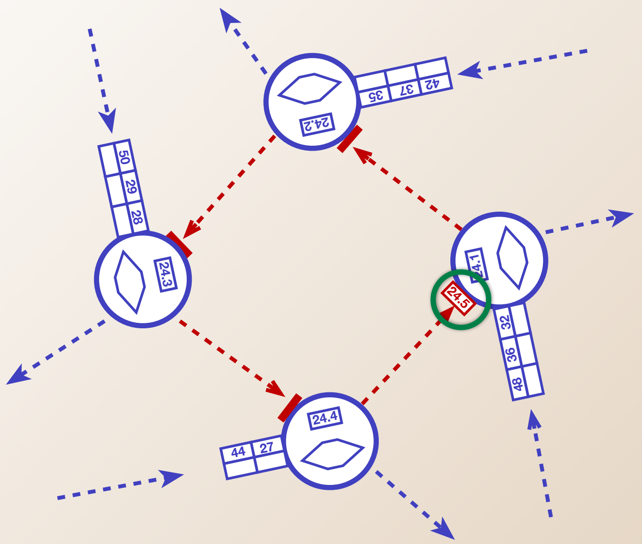


Poor and local lookahead leads to bad performance





Poor and local lookahead leads to bad performance



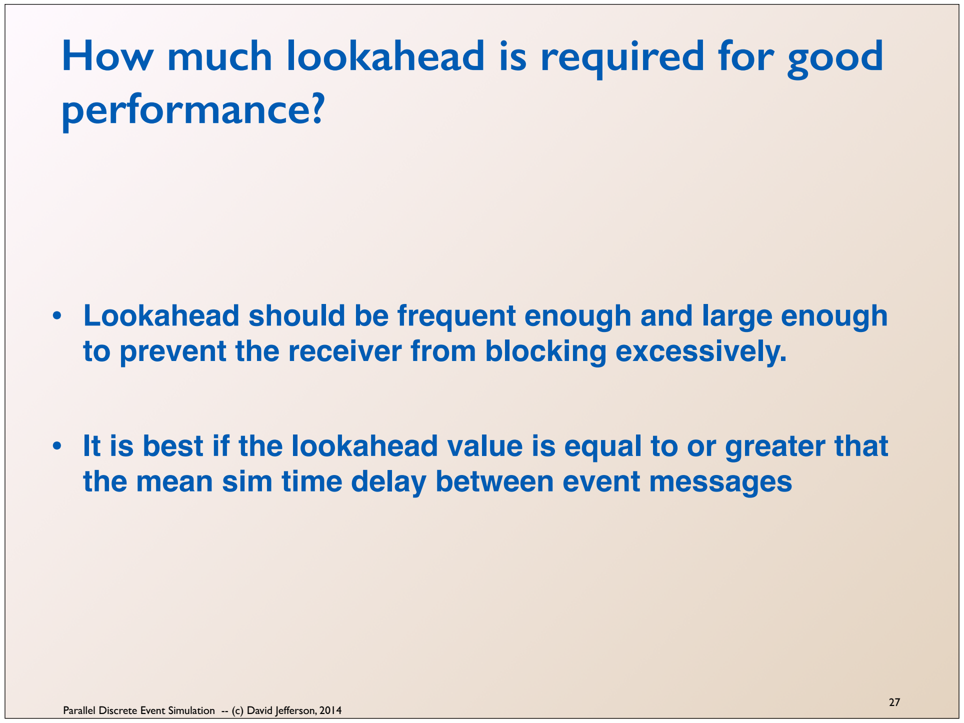
Parallel Discrete Event Simulation -- (c) David Jefferson, 2014

26

The last 5 slides have shown how null messages are moving around the cycle, incremented by 0.1 units of sim time with each hop. They will continue to go around the cycle like that until a null message has arrived at the South object carrying a time stamp > 27.0. That will be 8 time around the cycle, and a total of 32 sequential null messages. Only at that point it will it be safe for South to execute the event it has at the head of its other queue with a time stamp of 27.0. But then another cycle of null messages will start, this time having to count up to at least 28.0, which is the time of the next lowest event message, which happens to be in West's queue.

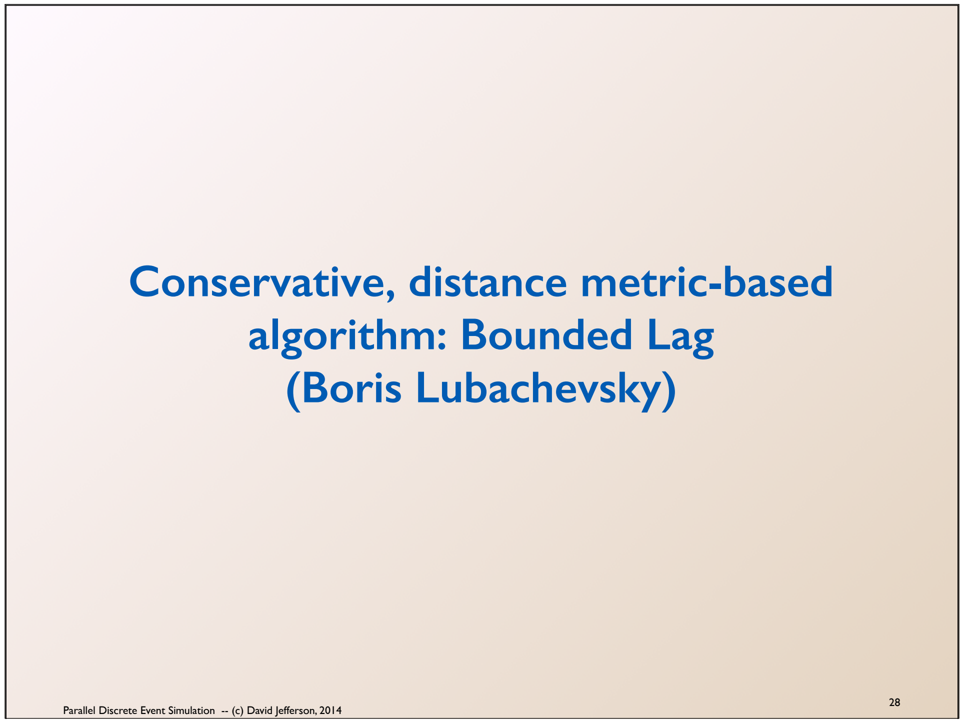
If we had chosen in this example a lookahead value of 0.01 instead of 0.1, then 10 times as many event messages would have to be transmitted, all sequentially, before the first real event message could be executed.

This is not deadlock, but it can be insanely slow and waste lots of bandwidth transmitting null messages.



How much lookahead is required for good performance?

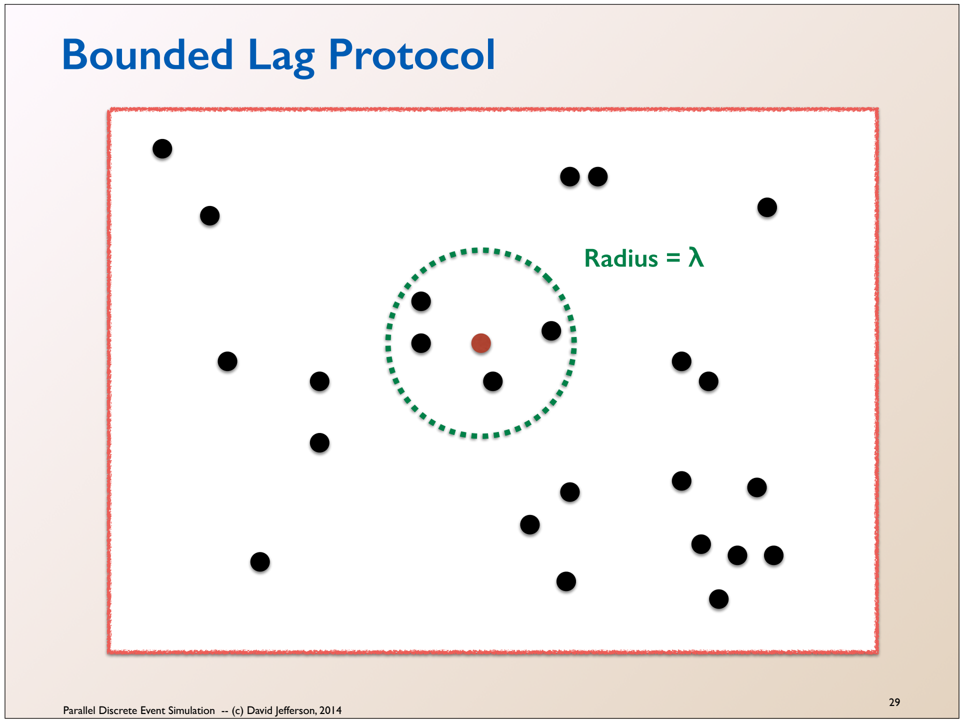
- Lookahead should be frequent enough and large enough to prevent the receiver from blocking excessively.
- It is best if the lookahead value is equal to or greater that the mean sim time delay between event messages



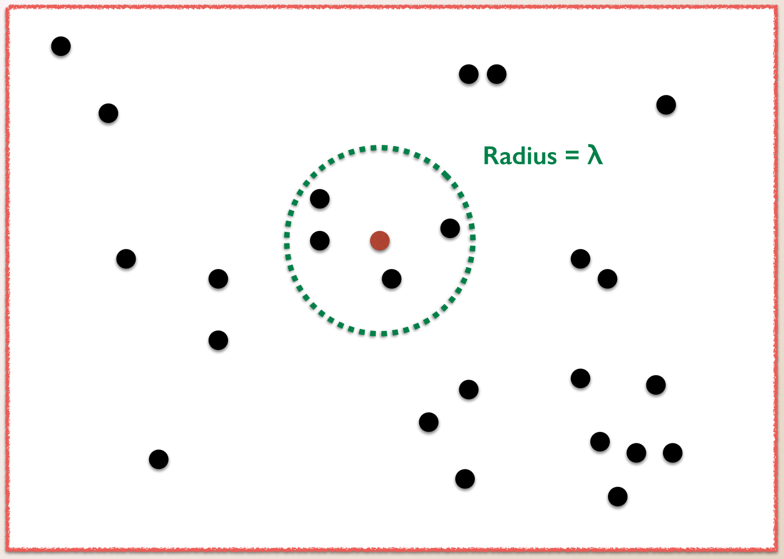
Conservative, distance metric-based algorithm: Bounded Lag (Boris Lubachevsky)

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

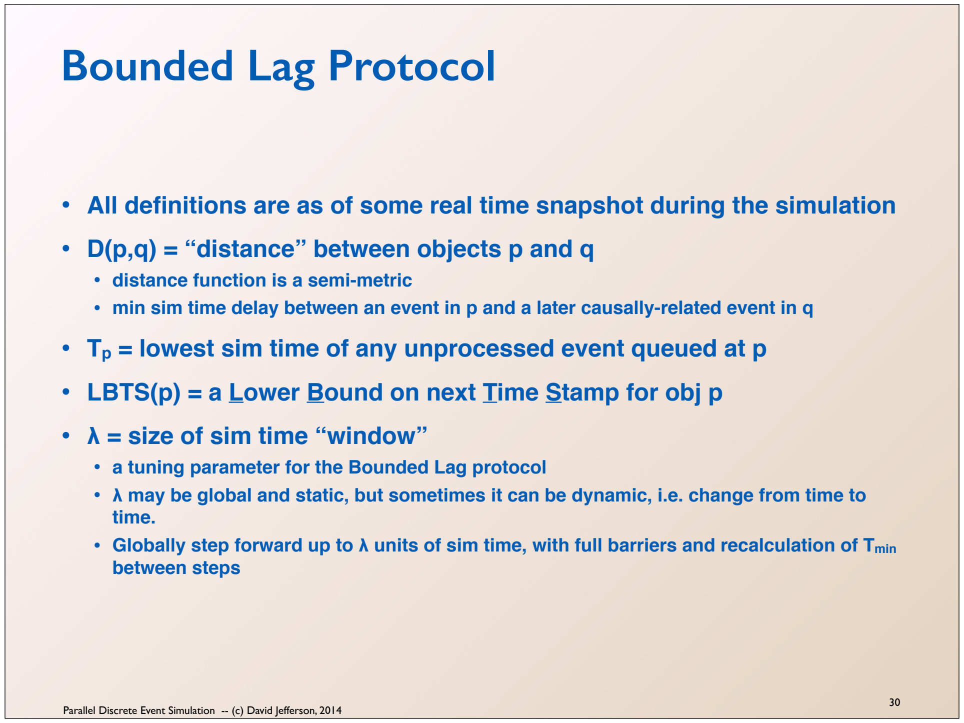
28



Bounded Lag Protocol



In the Bounded Lag protocol objects are considered to be located in a "space" and there is a "distance" between them that reflects the delay in simulation time required for each to affect the others. In this diagram the red object is in the center of a "sphere" of radius λ, meaning that the other 4 objects within the sphere are the only ones that can possibly interact with it in less that λ delay. In other words if an object within the sphere (other than the one at the center) is at simTime **t**, then it cannot schedule and event for the red object at a time lower than **t+λ.**



Bounded Lag Protocol

- All definitions are as of some real time snapshot during the simulation
- D(p,q) = “distance” between objects p and q
  - distance function is a semi-metric
  - min sim time delay between an event in p and a later causally-related event in q
- T_p = lowest sim time of any unprocessed event queued at p
- LBTS(p) = a Lower Bound on next Time Stamp for obj p
- λ = size of sim time “window”
  - a tuning parameter for the Bounded Lag protocol
  - λ may be global and static, but sometimes it can be dynamic, i.e. change from time to time.
  - Globally step forward up to λ units of sim time, with full barriers and recalculation of T_{min} between steps

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

30

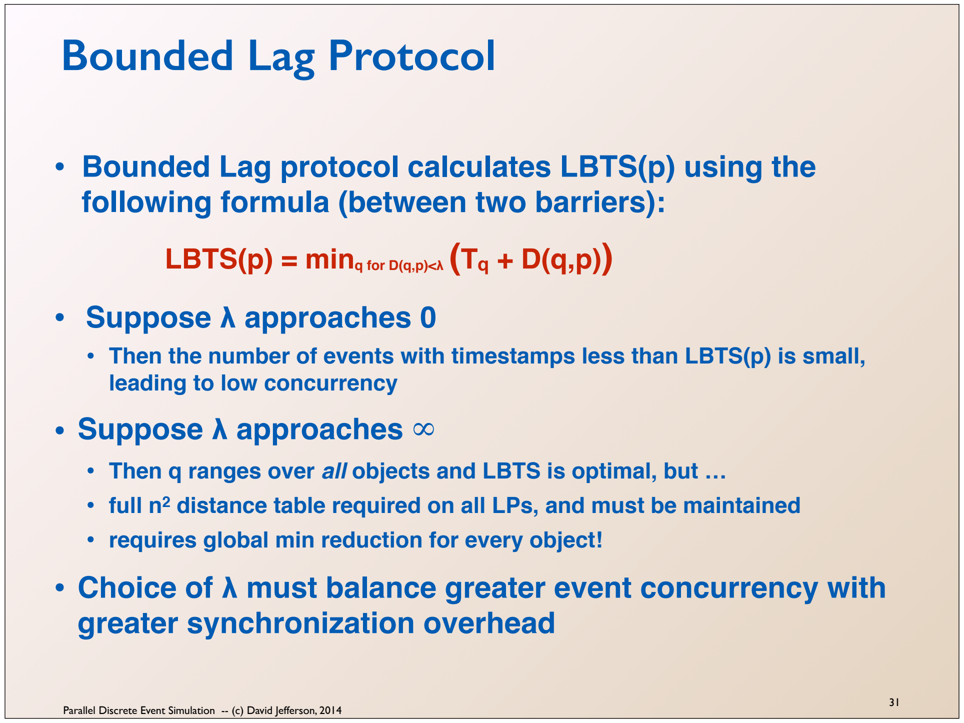
The distance function described need not be a true metric function. The only real requirement is

D(p,q)>= 0	Positive definiteness applies

But the other properties of a true metric are not required

D(p,p)	need not be == 0	
D(p,q)	need not be == D(q,p)	
D(p,q) + D(q,r)	need not be >= D(p,r)	Triangle Law need not apply

The term “bounded lag” refers to the fact that at any given moment all objects must have their sim clocks between T_{min} and $T_{min}+\lambda$, i.e. the difference in time between the lowest and fastest simulation clocks at any instantaneous global snapshot of the simulation (the “lag”) is bounded by λ.



Bounded Lag Protocol

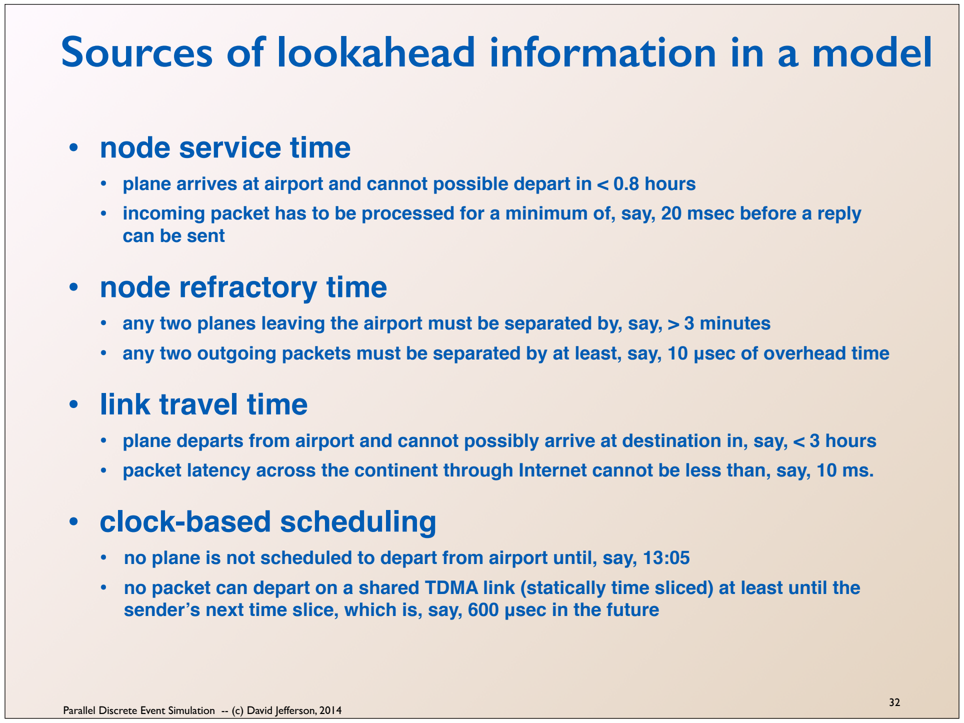
- Bounded Lag protocol calculates LBTS(p) using the following formula (between two barriers):

$$\mathrm{LBTS}(p) = \min_{q \text{ for } D(q,p)<\lambda} \left(T_q + D(q,p)\right)$$

- Suppose λ approaches 0
  - Then the number of events with timestamps less than LBTS(p) is small, leading to low concurrency
- Suppose λ approaches ∞
  - Then q ranges over *all* objects and LBTS is optimal, but …
  - full n^2 distance table required on all LPs, and must be maintained
  - requires global min reduction for every object!
- Choice of λ must balance greater event concurrency with greater synchronization overhead

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

31



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

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

32

Almost any property or logic in the model that constrains when an event message may be sent, or the delay into the future when it is to be processed can be a source of lookahead information. It is up to the modeler to analyze the model and offer the best lookahead information that is feasible.