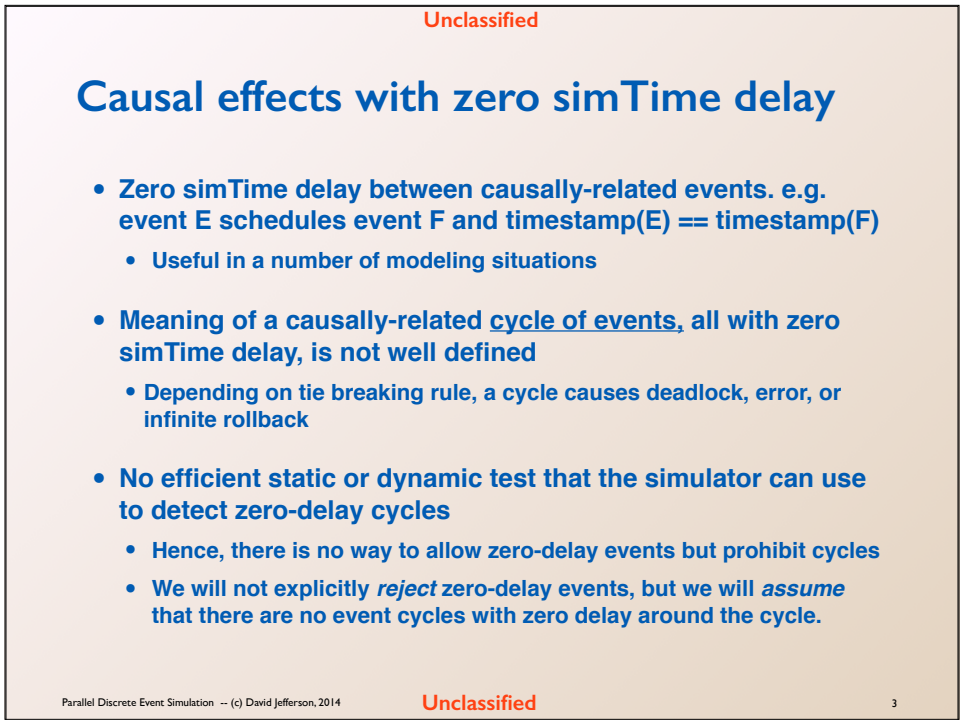
Parallel Discrete Event Simulation Course
#4

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-649697

Reprise



Zero simTime delay is useful in some circumstances when there is no danger of a cycle. One example:

Suppose an object A in a simulation is big and fat and has a lot of potential internal parallelism. So the model builder would like to divide it into several objects A, A2, A3, each sharing part of the state that the original A had. But s/he does not want to change the interface that A (or the A's) have to all of the other objects in the simulation. So s/he selects one of the several objects to retain the old name A and receive all of the event messages from outside as before. Every event message that A receives it also forwards to A2 and A3 with zero time delay. Now they all (A, A2, and A3) have copies of all incoming event messages that the original A would have received, and with the same time stamps, and no risk of cycles, and no need to change the code anywhere else in the model.

Multiple events scheduled for *same simTime* and *the same object*

- Tie in space-time
- Events are generally not commutative
- Thus we need ***tie breaking*** *rule to indicate how the simulator should handle it.*
- A tie-breaking rule should
  - generalize to n-way tie
  - be symmetric
  - be deterministic
  - be portable, i.e. platform-, implementation-, object name- and configuration-independent

Tie-breaking rule is important semantic issue, with real modeling consequences but little performance impact.

After this discussion we will largely ignore it in this class.

Unclassified

Potential tie-breaking rules for the simulator

Require event methods to be commutative?	Not desirable or enforceable
Treat it as an error?	Happens legitimately, so not an error
Execute atomically in arbitrary or random order?	Not portable or not deterministic
Execute events concurrently in separate threads?	Not deterministic or repeatable; requires inter-thread synchronization
Earliest send time is first executed?	Not symmetric, still allows ties
FIFO - earliest scheduled in real time is first executed	Not deterministic or repeatable
Order by name of scheduling object?	Not invariant under name changes; still allows ties
Order by name of method (i.e. give priority to certain event methods)	Still allows ties; not invariant under method nam changes
Modeler specified insertion to be first or last among equals?	Does not solve problem. What if two tying events are designated as last?
Extend simTime with extra low order bits to break ties (i.e. priority in low order its)?	How are those bits chosen? Still allows ties
Require receiving LP code to break ties among a ***set of event messages***?	My favorite, but harder to implement, more event overhead, and complicates model code

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 Unclassified 5

Note: We are rejecting many potential tie breaking rules here. But I should be clear that I reject them for the simulator. The model code is free to make several of these choices. But the handling of ties must be a choice that the *model writer* makes, not the *simulator*.

Two events that say "Turn left" and Take on step forward" are not commutative. They leave you in different states depending on what order they are executed in. And if the two events arrive with the sam simulation time stamp, how should they be handled? The simulator won't know, but the modeler may have a convention in mind for how to handle it.

Require commutativity: -- no way to enforce, and you don't want to anyway since in most cases events are not commutative.
Error --amounts to forcing programmer to break ties, defensible as a last resort if some other method o breaking ties is in place

Arbitrary order -- no good; leads to nondeterminism

Threads -- no good; nondeterministic, even if events are synchronized mutual exclusion

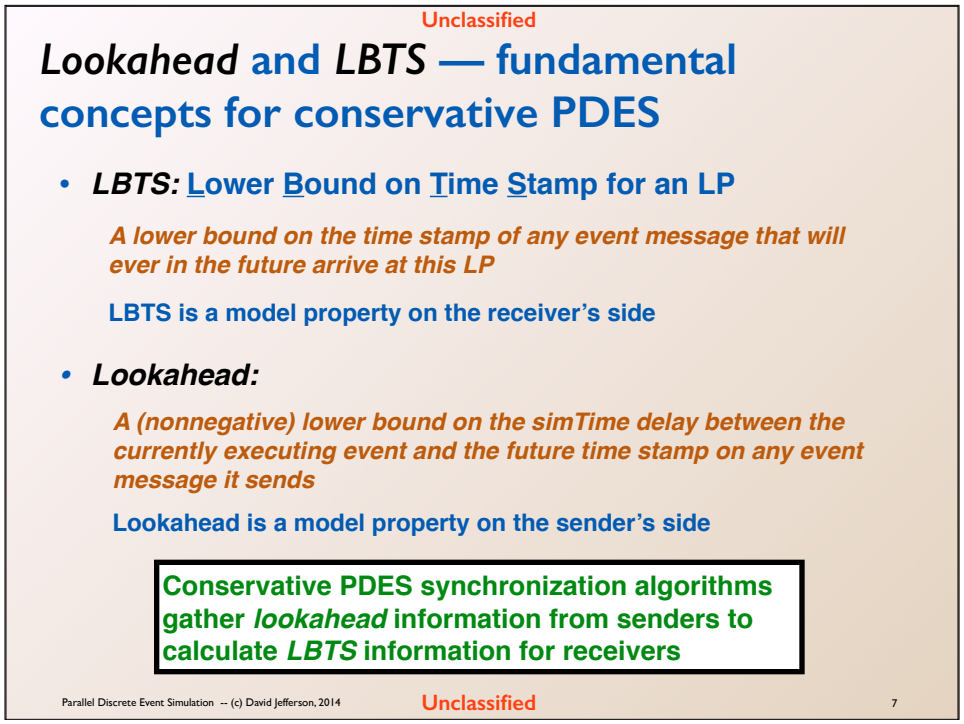
First scheduled -- common in sequential algorithm, but does leads to nondeterminism in parallel

Programmer chooses -- breaks the separation between simulator and simulation

Virtual time -- very good, but ties are still possible unless nondeterministic mechanism is added

Multi-event -- this is the one I favor, but it breaks the one-event, one method-call convention. It defines an "event" as the handling of the entire unordered set of event messages that arrive with the same simulation timestamp. Most of the time there are no ties, and the set is a singleton. But on the rare occasion of ties the set contains more than one element, and then the simulator calls a multi-event handler form the model, and that handler decides how to handle the multiple events. The point is that it is the model's job to decide why ties in time, mean, not the simulator's job.

The fundamental notions of conservative synchronization: LBTS and Lookahead



LBTS is usually cited as a simulation *time value*, whereas Lookahead is usually defined as a *delay* in simulation time from the current time until the lowest time stamp on any event message it will ever send. But the two notions are really symmetric. We could instead define *sender LBTS* and *receiver LBTS*.

Sender LBTS is a lower bound on any event message time stamp that the LP will ever send from now on.

Receiver LBTS is a lower bound on any event message time stamp that the LP will ever receive from now on.

The sender LBTS is just the current simulation time of the LP + its lookahead. This should make it clear that the two notions are complementary.

It should also be clear that altho LBTS and Lookahead values are lower bounds, they are not necessarily the tightest possible lower bounds. It is common that an object knows that its lookahead value at the moment is some value Δ, but in fact it does not send its event next message until out farther in the future than that. The LBTS that the modeler can calculate at a given moment in simTime for an object may be lower than the delay turns out to be.

A boundary case is that 0 is always a lower bound on the simTime delay until sending the next event message. It is not a very helpful lower bound, but sometimes (such as in the case of Poisson processes) it is the best you can do.

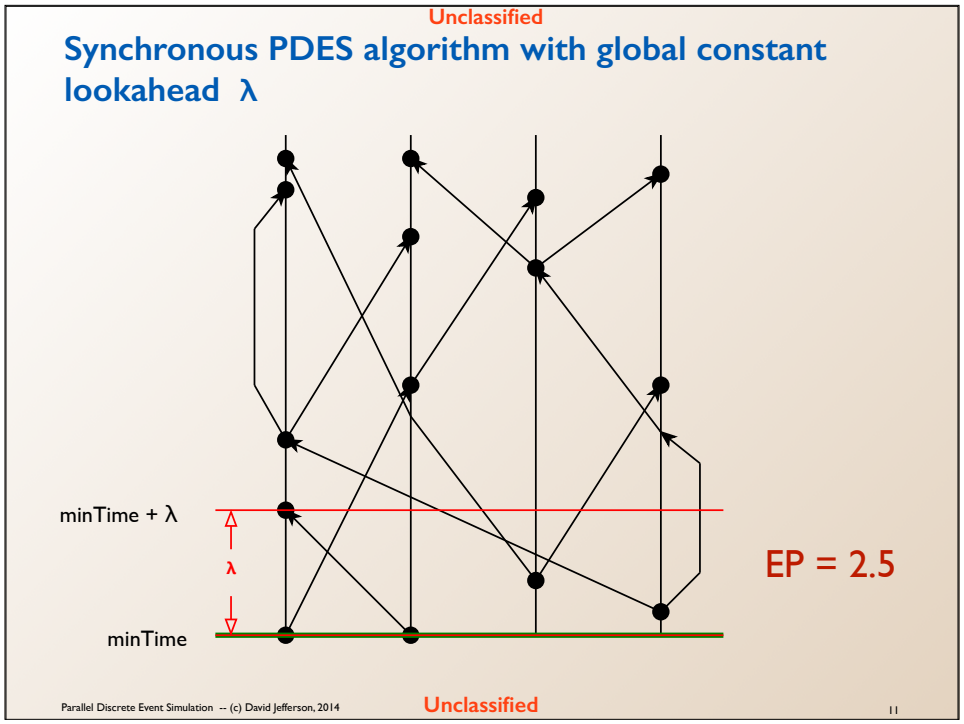
First Conservative PDES algorithm: Synchronous Algorithm with Lookahead Window

Synchronous algorithm with global, constant lookahead λ

- ***Global*** **lookahead: all objects have the same lookahead value of at least λ**
- ***Constant*** **lookahead: the lookahead value λ is unchanged throughout the simulation time**
- **Thus, any object can schedule events for any other, but only with time stamps >= (simTime + λ)**
  - **Fatal error to schedule an event for a time sooner than that**
- **Assume λ > 0**

Synchronous PDES algorithm with global constant lookahead λ

```
while (true) do {
  minTime == min(all objects obj)(obj.simTime);
  barrier();
  if ( minTime >= stopTime ) break;
  ! LBTS == minTime + λ
  execute all enqueued events scheduled for
                    times <= (minTime + λ);
  barrier();
}
```



Execute all events such that

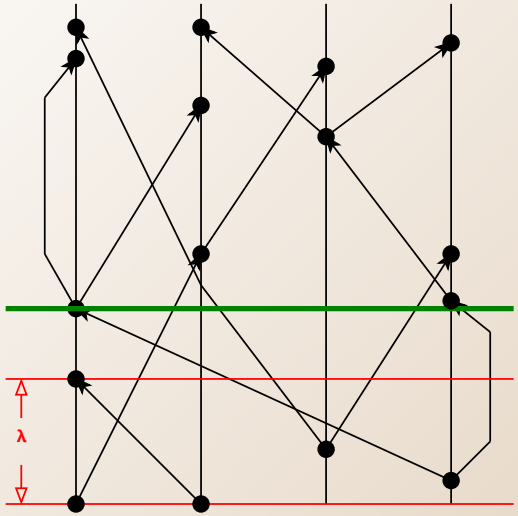
minTime <= t <= minTime + λ

EP = effective parallelism, assuming every event takes 1 unit of wall clock time and assuming synchronization overhead (between λ-windows) is 0.

Here the effective parallelism, S / (M + v), in the first λ-window is 2.5 because S = 5 events were executed in the time it take to run M = 2, with overhead v = 0.

Unclassified

Synchronous PDES algorithm with global constant lookahead λ



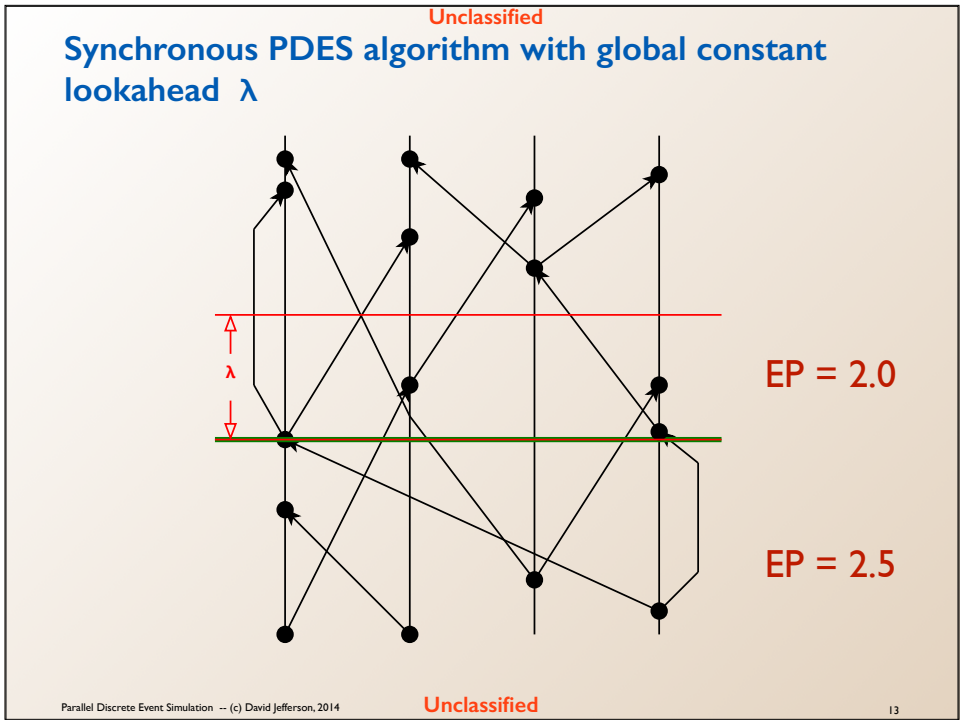
EP = 2.5

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

Unclassified

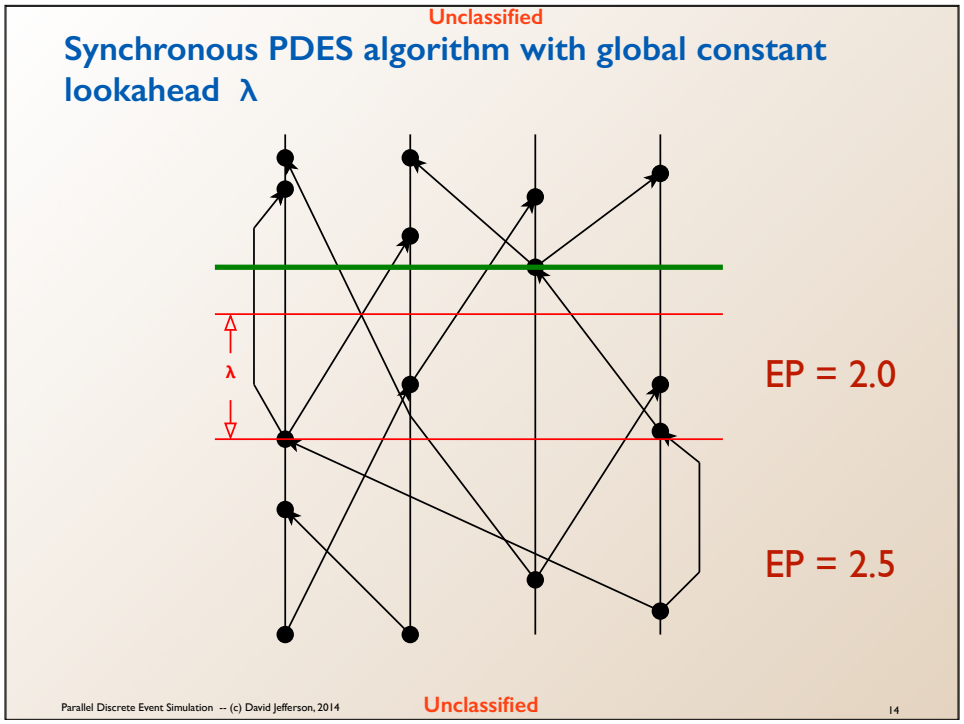
12

Calculate new minTime (green line)

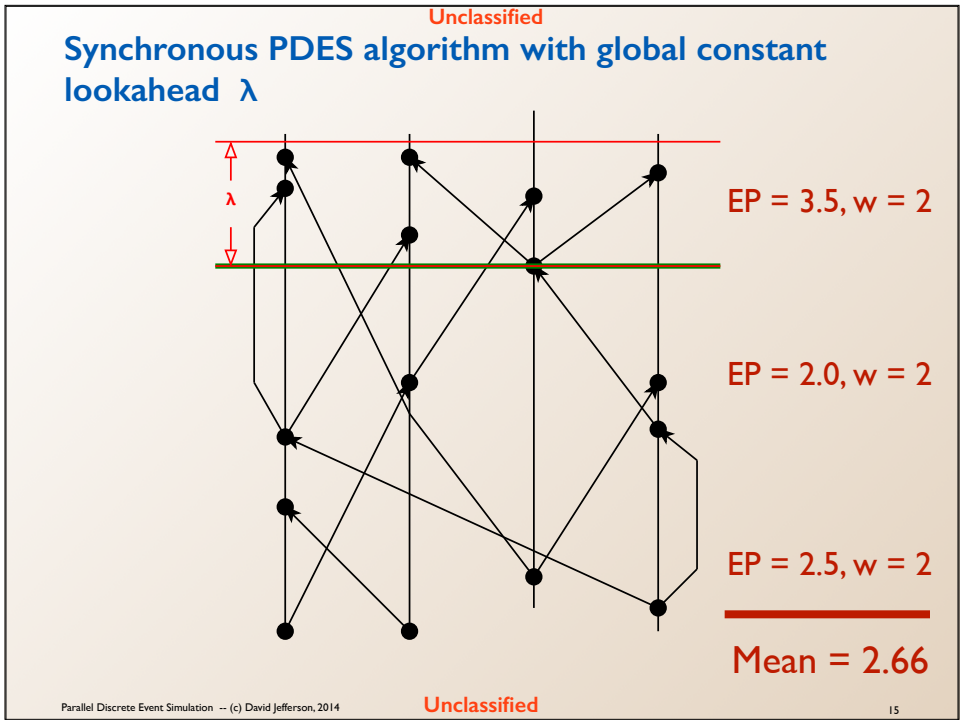


Again execute all events such that

minTime <= t <= minTime + λ



Again calculate new minTime (green line)



Again execute all events such that

minTime <= t <= minTime + λ

… and so forth

The mean effective parallelism for this whole simulation is 2.66

The mean actually has to be calculated as a weighted mean, with the effective parallelism of each λ-window weighted by the length of wall clock time (M) that that window to too execute. As it happens, in this example, all three windows were executed in 2 time units, so they are equally weighted.

Unclassified

Performance of the synchronous conservative PDES algorithm with global lookahead λ

- **Performance in each λ-window depends on:**
  - n = number of LPs (objects)
  - M = maximum over all LPs of the total length in wall clock time to execute the events in each LP's λ-window
  - S = sum over all LPs of the total length in wall clock time of the events in each LP's λ-window (i.e. time to execute sequentially on one processor)
  - v = window overhead (time for calculation and distribution of minTime + time for barrier synch)
- **Then, assuming 1 LP per node**
  - S/n <= M <= S
  - Balance [1/n .. 1] = S / (n * M)
  - Time to execute λ-window = M + v
  - Effective parallelism in the λ-window = S / (M + v)
  - Parallel efficiency = S / ((M + v) * n)
  - As λ declines to 0, S declines to toward 0
  - When λ = 0, the algorithm is equivalent to naïve algorithm.

v generally grows as log n, but for present purposes we will consider it a constant at any given scale n. It includes barrier synch, which thus includes all message delivery times because barriers wait for message quiescence.

Effective parallelism is obviously maximized when M + v is small compared to S.

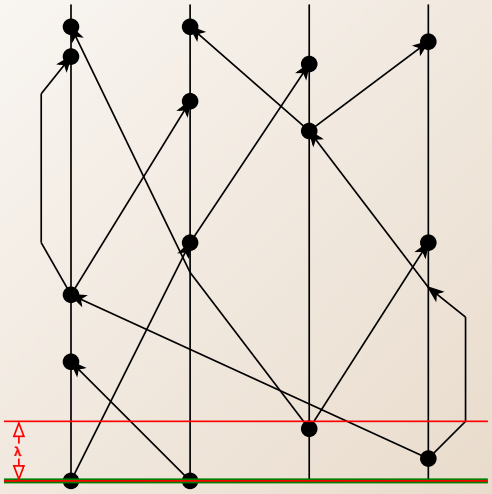
We thus want M to be small compared to S, which is the same as saying we want the simulation to be balanced.

We also want v to be small compared to S because of course v is overhead not much under control of the model programmer.

Note that as λ gets smaller there are fewer events in each window, so S and M both get smaller, and the balance gets worse as well. However, v generally does not get smaller, so as a result both effective parallelism and parallel efficiency get word. This is why a large λ — a long lookahead — is crucially important.

Unclassified

Synchronous PDES algorithm with global constant lookahead λ

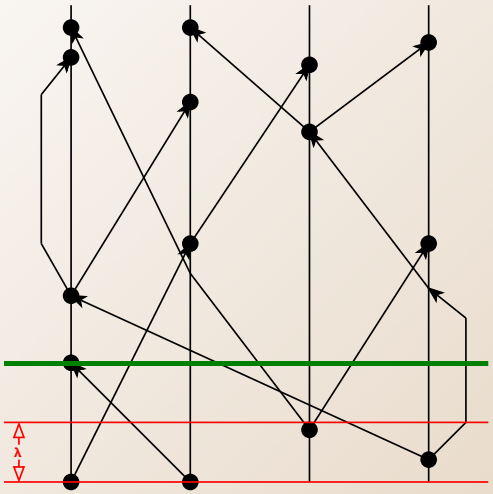


EP = 4

Unclassified

Unclassified

Synchronous PDES algorithm with global constant lookahead λ

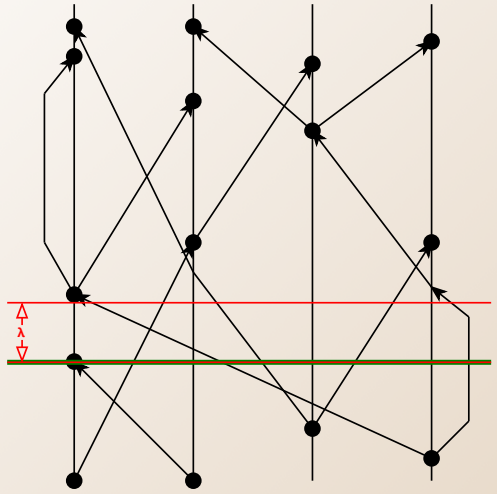


EP = 4

Unclassified

Unclassified

Synchronous PDES algorithm with global constant lookahead λ



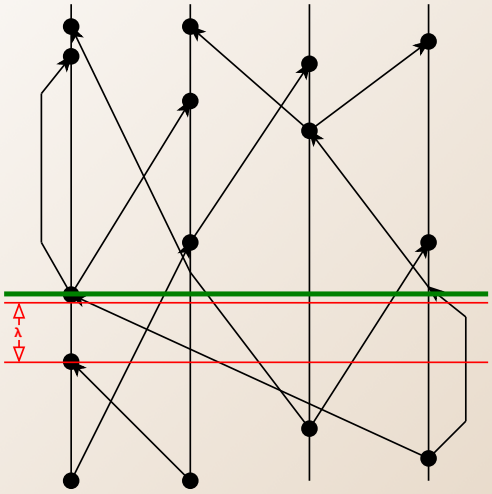
EP = 1.0

EP = 4.0

Unclassified

Unclassified

Synchronous PDES algorithm with global constant lookahead λ



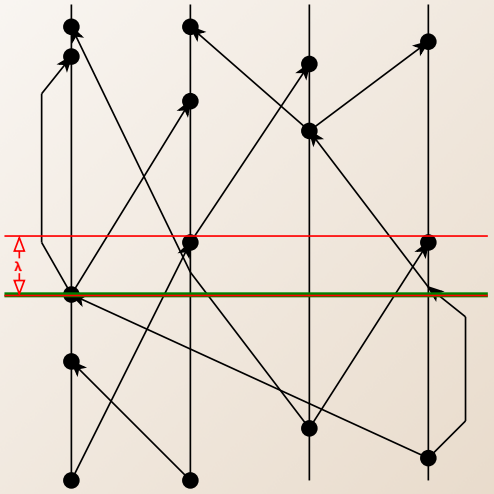
EP = 1.0

EP = 4.0

Unclassified

Unclassified

Synchronous PDES algorithm with global constant lookahead λ



EP = 3.0

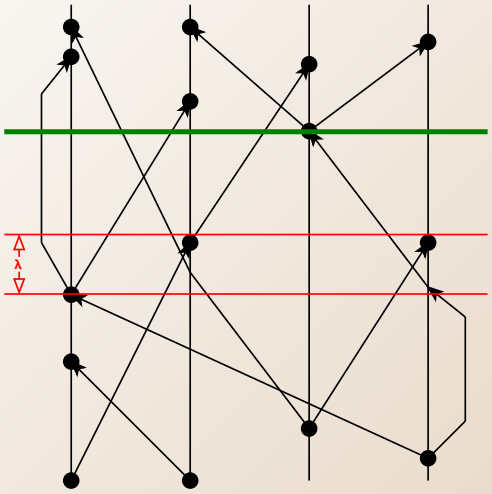
EP = 1.0

EP = 4.0

Unclassified

Unclassified

Synchronous PDES algorithm with global constant lookahead λ



EP = 3.0

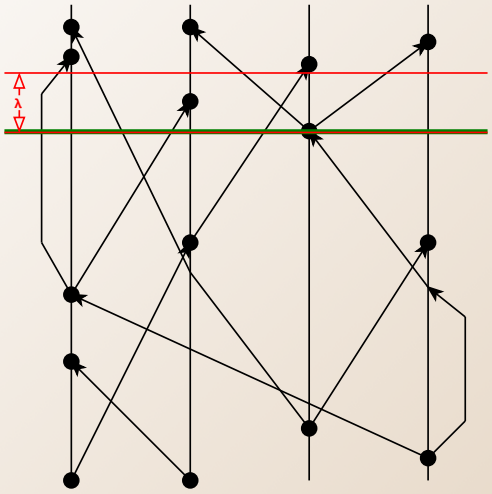
EP = 1.0

EP = 4.0

Unclassified

Unclassified

Synchronous PDES algorithm with global constant lookahead λ



EP = 2.0

EP = 3.0

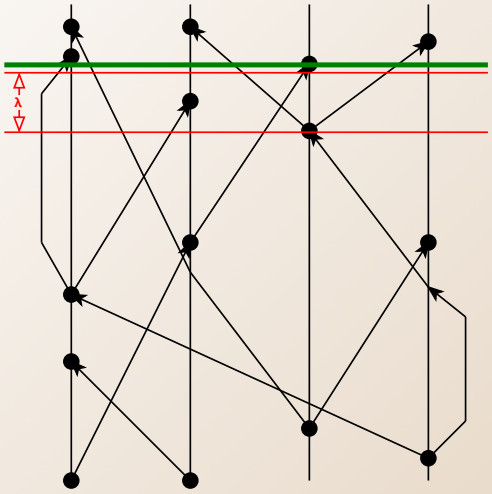
EP = 1.0

EP = 4.0

Unclassified

Unclassified

Synchronous PDES algorithm with global constant lookahead λ



EP = 2.0

EP = 3.0

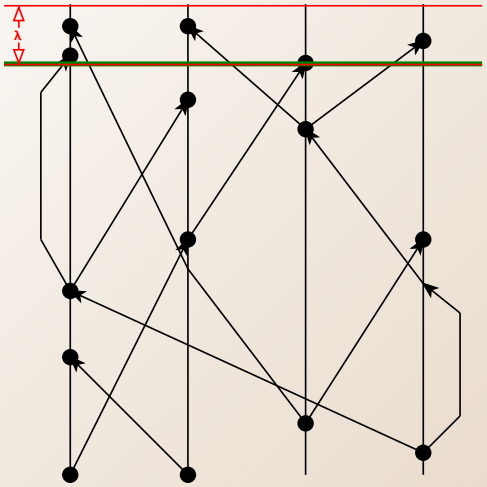
EP = 1.0

EP = 4.0

Unclassified

Unclassified

Synchronous PDES algorithm with global constant lookahead λ



EP = 2.5, w = 2

EP = 2.0, w = 1

EP = 3.0, w = 1

EP = 1.0, w = 1

EP = 4.0, w = 1

Mean = 2.5

Unclassified

Again the mean effective parallelism is calculated as a weighted mean, and this time not all of the λ-windows are equally weighted. The last one takes twice as long as the others.

Summary properties of the Synchronous PDES Algorithm with Constant Global Lookahead λ

- Requires a single, global, unchanging lookahead value, λ, for all LPs and all time
- Applies to *any* discrete event model — no structural restrictions other than known λ-value, which can be 0.
- λ should be as large as possible consistent with the dynamics of the model
- In the worst case, λ == 0, the algorithm reduces to the naïve synchronous parallel algorithm discussed earlier, and its performance is usually much worse than the sequential algorithm.
- Does not take advantage of the fact the lookahead values may differ from LP to LP, and from time to time.
- We can do better!

End Reprise

Conservative Parallel Discrete Event Simulation of Static Graph-type Models

Weaknesses of synchronous, global static λ-window algorithm

- **It assumes the same λ-value for all objects.**
  - **Some objects may be able to offer larger λ-guarantees than the global minimum**
  - **When two simulations are federated with this synchronization algorithm, the smaller λ-value must be used**
- **It assumes the same λ-value for all time**
  - **different temporal phases of the computation may permit larger λ-values**
- ***Global reduction* required to calculate the start of each λ-window.**

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 order-preserving**

K. Mani Chandy and Jayadev Misra

Misra **Chandy**

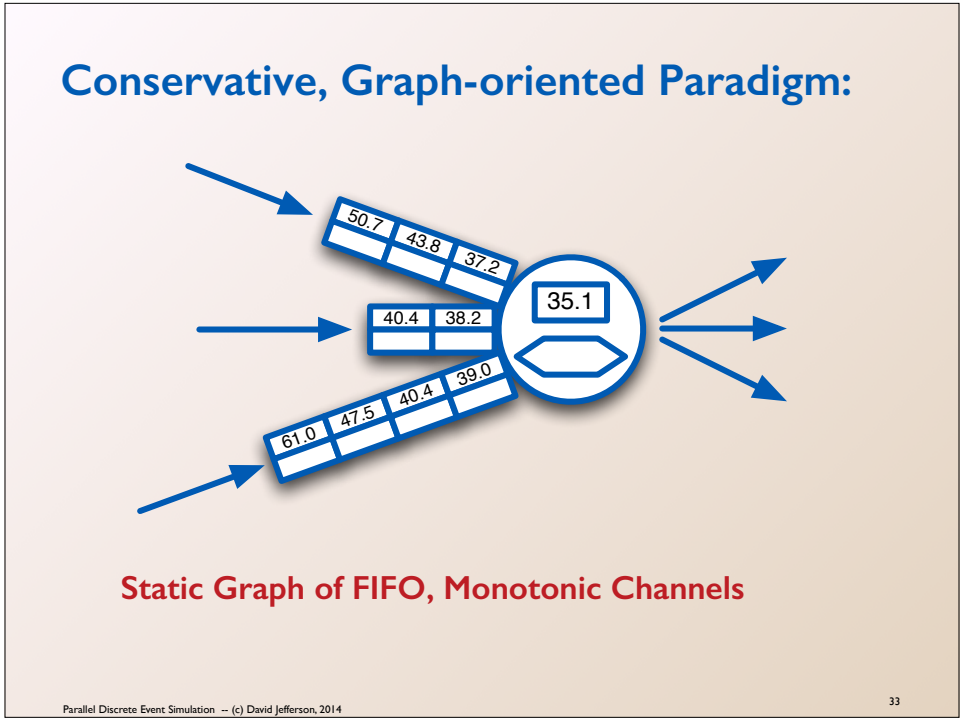
Initial work independently duplicated by Randy Bryant

Profs. Chandy and Misra of U. of Texas at Austin (and independently, Randy Bryant of CMU) were among the first researchers to consider the question of how to parallelize discrete event simulations. They started with formalizing the synchronization problem with a restricted class of discrete event models, the static graph models, and they introduced the null message mechanism for that class of models.

Footnote: Why are the asymmetric, mismatched terms "conservative" and "optimistic" used in this field? Couldn't a better pair of terms have been found? Here is the true story (as I remember it after 30 years). In the 1980s there was something of a rivalry between the two camps of researchers: those who advocated what came to be known as conservative methods, more or less led by Chandy and Misra, and those who championed optimistic methods (more or less led by me). I wanted to call the two categories "pessimistic" and "optimistic". Chandy and Misra did not like the word "pessimistic" applied to their methods. They suggested "conservative" and "liberal". I did not like the term "liberal" applied to my methods (too political). So we compromised, each agreeing that we name our respective methods. They chose "conservative" and I chose "optimistic".

Which models are appropriate for conservative graph-oriented PDES?

- **Models based on static graphs:**
  - networks at the packet / link level
  - queueing systems
  - transportation systems
  - infrastructure
- **Not models in which many objects potentially interact directly with many other objects**
  - networks at the connection level (where any node can connect to any other)
  - models involving objects moving freely in space and interacting, e.g. particles, or mobile networks
- **Other models for which conservative PDES is poor:**
  - models with poor "lookahead" properties
  - models requiring dynamic creation
  - models with arbitrary time "inversions"



Core conservative graph-oriented algorithm:

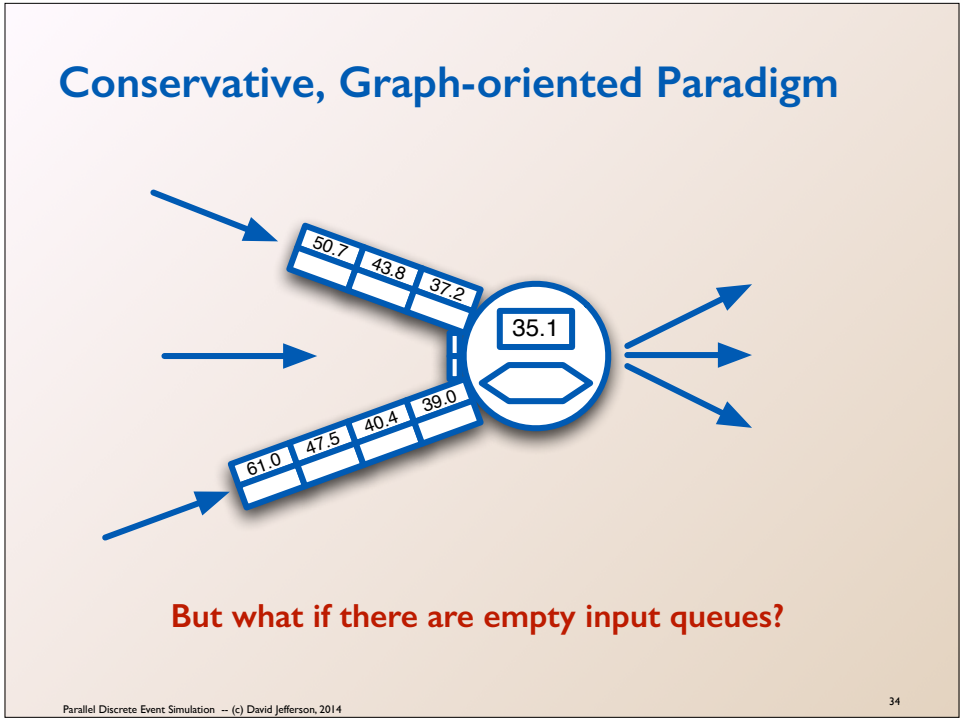
1) Each object has a static number of input queues, one for each incoming arc in the static communication graph, i.e. one for each other object that ever sends event messages to it.

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, of course, be static.
c) Neither new channels now new objects can be created. (This condition can be partially relaxed later, but .)
d) Channels can be effectively deleted by the sender using a simple trick; 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:



Core conservative graph-oriented algorithm:

But 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 the empty queue(s) nonempty.

Note: The only way more than one queue can be empty is either (a) at initialization time when queues are initialized to be empty, or (b) in the case of ties, since in the absence of ties only one message is removed at a time from the set of input queues.

Unclassified

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

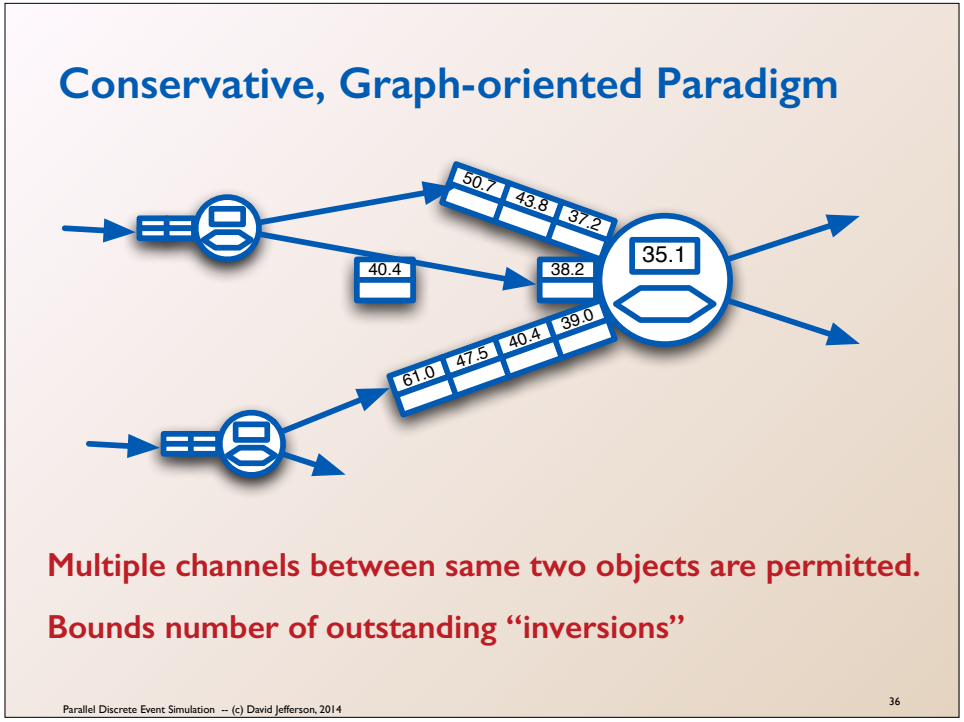
Unclassified

35

Notice that this algorithm does execute its events in exactly once in strictly nondecreasing order. (But recall discussion of ties, which this algorithm ignores.)

Notice also the big improvement that there is no global reduction and no global barrier. We calculate the LBTS without any need for global calculation, because in this paradigm it is not true that *any* object can send and event message to *any* other. Since only a small staid number of other objects can send to this one, we only need to do the min-reduction over that small number of possible senders.

But note: This algorithm is called naïve because it does not use any *lookahead* information. That will cause trouble, as discussed in the next slides!

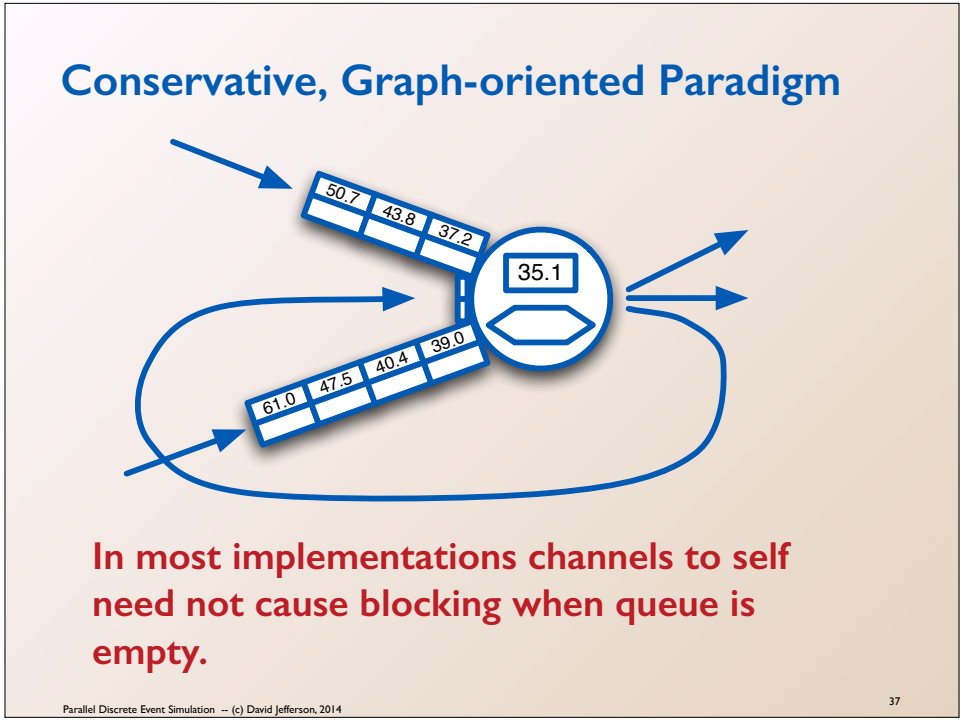


The upper left object has two channels leading to the same object on the right. Since receiving objects do not use the information of which channel a message arrives on (much as procedures do not use the information as to which other procedure called them), then there are really only two reasons to do this:

1) A two objects were combined into one at some stage of software development, but objects that send event messages were not modified accordingly
2) Time inversions: The sender needs to send an event message with a timestamp lower than one it has previously sent. It cannot do that on the same channel, but it *can* do it on a different channel

The message with time stamp 40.4 cannot be sent along the upper channel because messages with time stamps higher than that are already queued there. But it can be sent along the lower channel. If it is necessary to have two inversions, e.g sending messages with time stamps 50, 40, 30 in that order, which is two inversion, then three channels are required. More generally, for n inversions, n+1 channels between the same two objects are required. Since the number of channels between two objects is static, so is the max number of inversions.

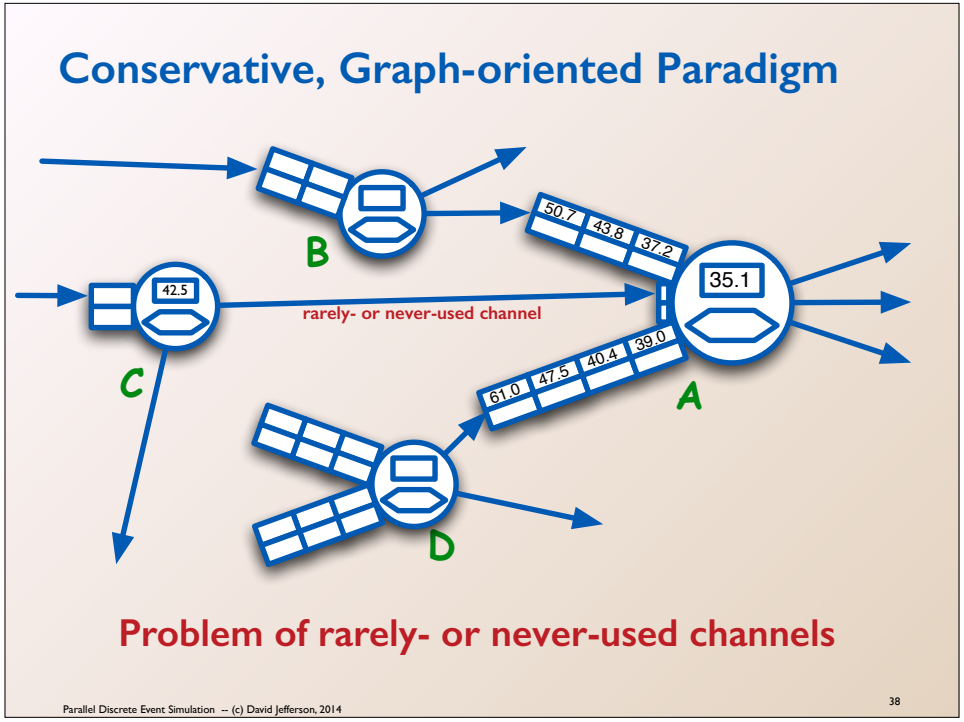
(**Warning:** Extra channels that are rarely or never used are *very* costly, for reasons to be described later. So don't use this trick lightly.)



Here is one significant exception to the rule that when an object has an empty queue it must block. An empty queue associated with a self-channel does not need to cause blocking when its queue is empty. If an event sends an event message-to-self, and the implementation enqueues it *synchronously*, i.e. during the sending event, before it completes, then if that self-channel queue happens to be empty when it comes time to select the next event to execute, there is no possibility that the next message arriving on the self channel will have a time stamp lower than those at the head of the other incoming channels. Furthermore, if the algorithm *did* block on an empty self-channel in that case, it would block forever, because no message will ever arrive to make it nonempty!

If, however, event messages-to-self are sent *asynchronously*,then the channel has to be treated like all of the others, and the circularity that it causes in the graph is like any other circularity and must be handled properly.

This might seem a minor detail, but event messages to self are quite common and also constitute circularity in the graph. This optimization can save a lot of overhead and trouble.

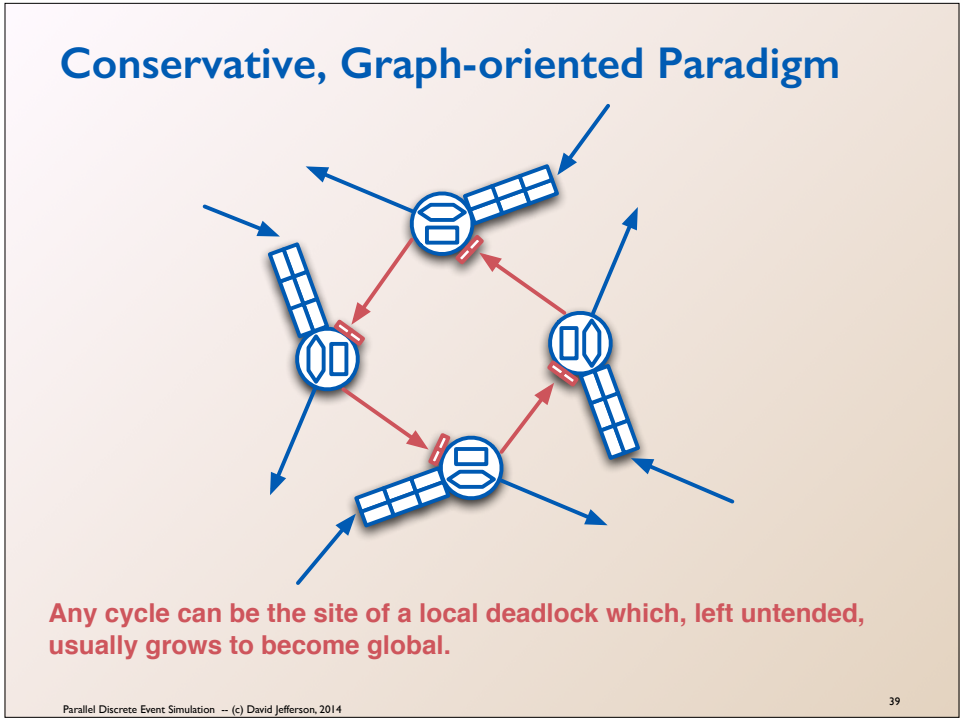


Here is one serious problem with the *naïve* conservative algorithm. What happens if object C has a channel to A but *rarely or never sends event messages* along that channel?Then the input queue associated with that channel at A is almost always, or always, empty! Hence, A is blocked most of the time, 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!

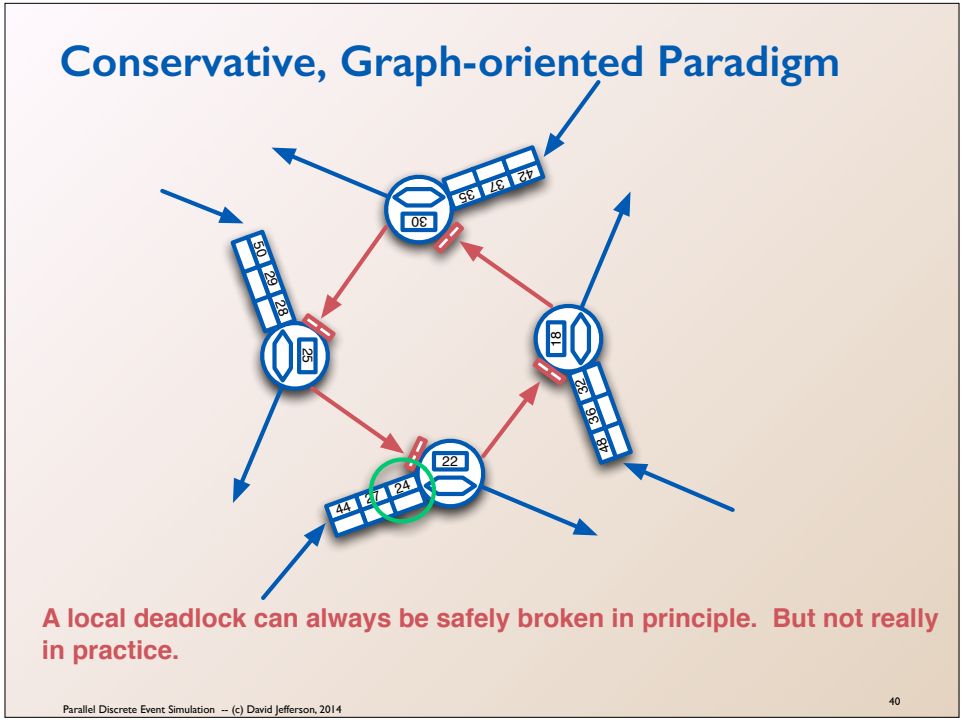


The most worrisome hazard for a conservative PDES is that any directed cycle of queues in the graph can become a local deadlock if all become simultaneously empty. And the deadlock grows inexorably as the queues to which the objects in the deadlock should be sending messages eventually become empty and 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*.

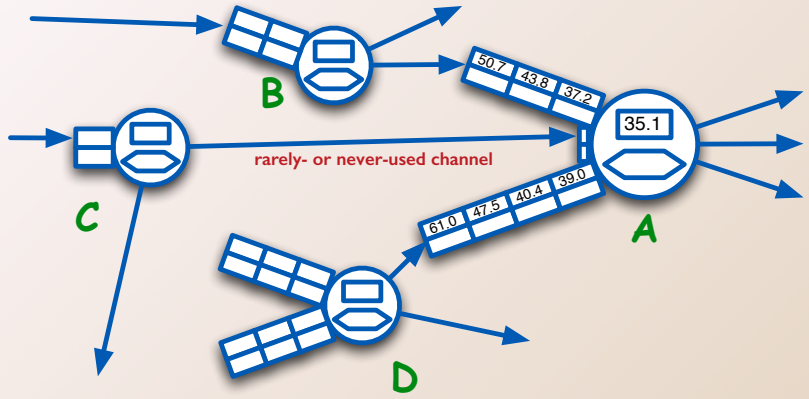


A cycle of empty event queues is only a deadlock according to the naïve paradigm in which an object always blocks until it all of its queues are nonempty. In general a not-so-naïve simulator can in principle break such a local deadlock safely by just selecting the lowest time stamped event message among 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.

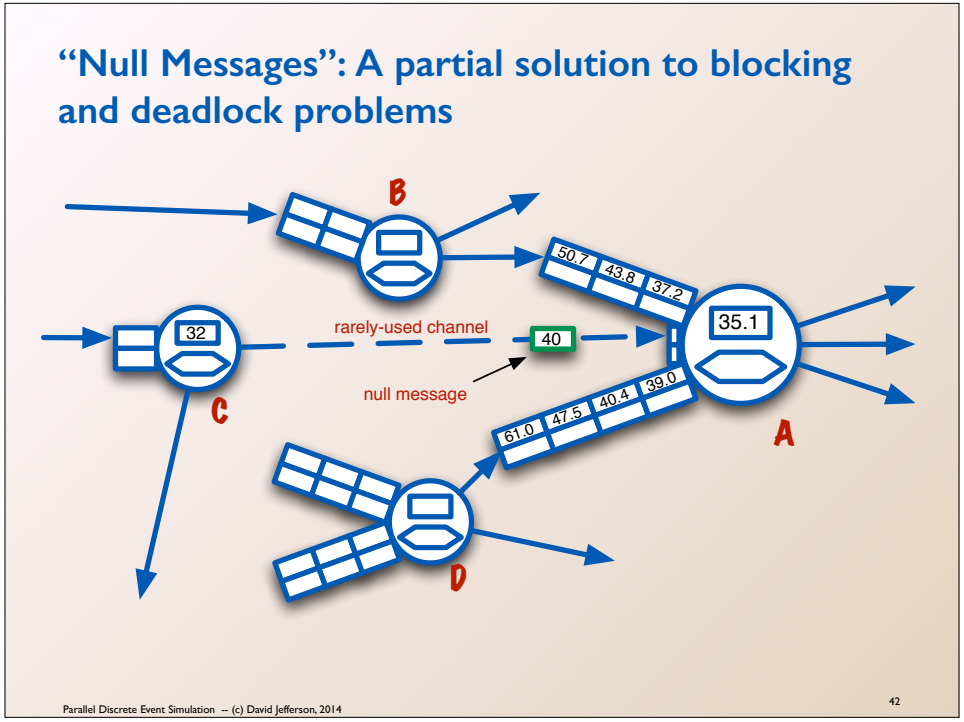
The problem is that the simulator cannot generally recognize a local deadlock when it happens. There is no fast, scalable algorithm for this. (For any given model, however, one might create such an algorithm as a special case.)

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.

"Lookahead": A partial solution to conservative blocking and deadlock problems



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.



The fundamental concept needed to deal with the problems of rarely- or never-used channels, and including cyclic local deadlocks, is "lookahead". Lookahead in this case takes the form of a guarantee from a Sender to a Receiver on a channel 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 this: "I don't know when or if I will send another message down this channel. But I guarantee now that if I ever do, it will not have a time stamp lower than t." A message that carries only lookahead information and does not carry any real event 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) 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 a practical requirement for conservative execution of any kind, including this graph-oriented algorithm. 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 either sequential or optimistic 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").