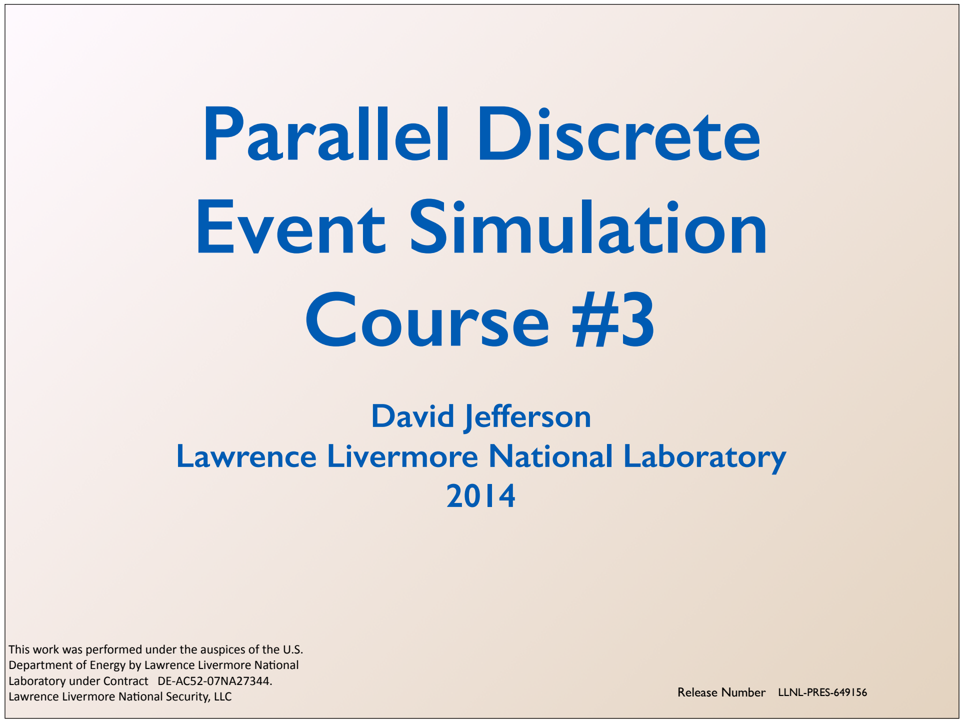
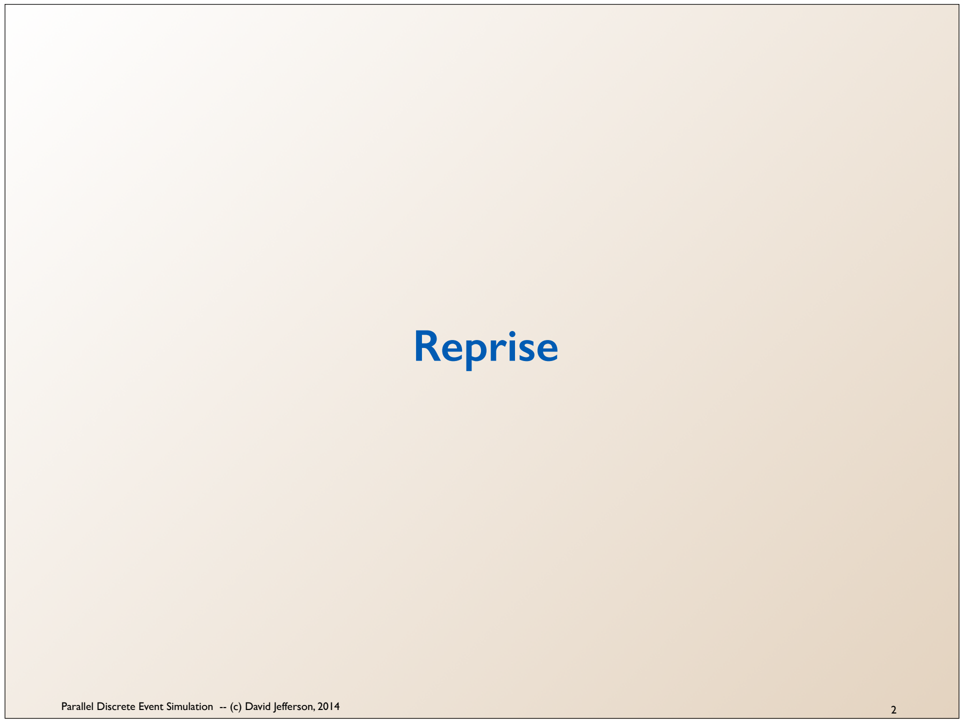


Parallel Discrete Event Simulation Course #3

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-649156

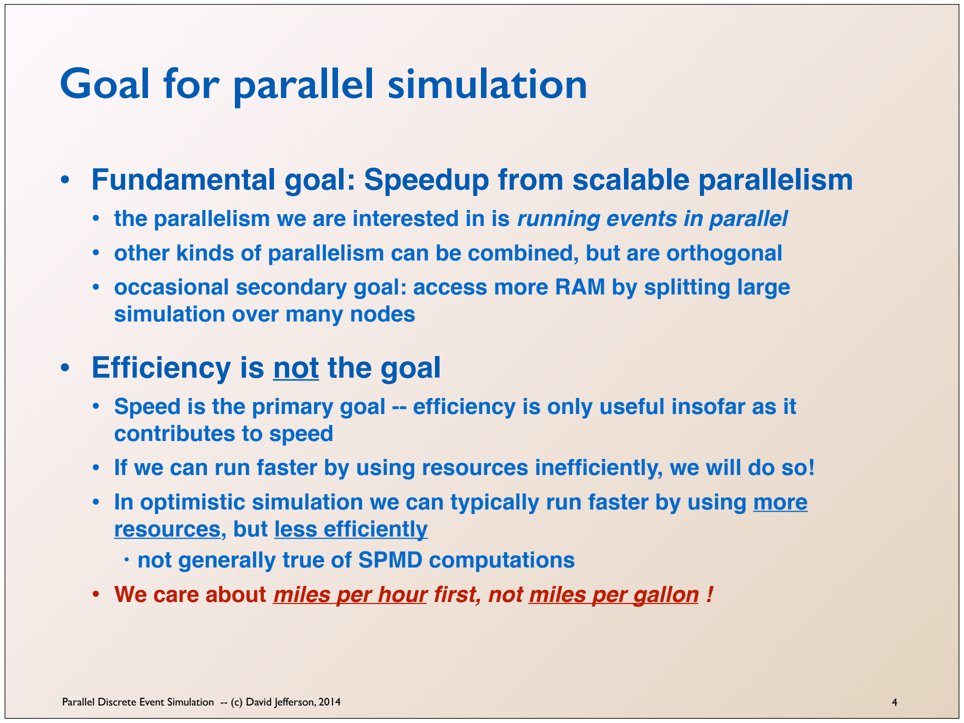


Reprise

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

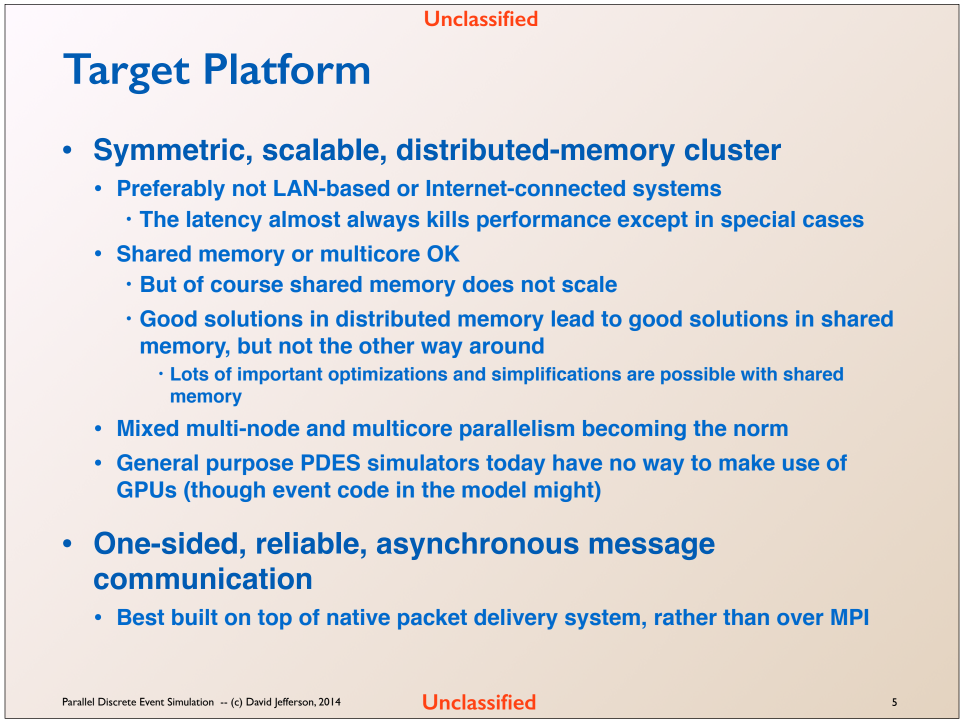
2

The General Paradigm of Parallel Discrete Event Simulation



Goal for parallel simulation

- **Fundamental goal: Speedup from scalable parallelism**
  - the parallelism we are interested in is *running events in parallel*
  - other kinds of parallelism can be combined, but are orthogonal
  - occasional secondary goal: access more RAM by splitting large simulation over many nodes
- **Efficiency is <u>not</u> the goal**
  - Speed is the primary goal -- efficiency is only useful insofar as it contributes to speed
  - If we can run faster by using resources inefficiently, we will do so!
  - In optimistic simulation we can typically run faster by using <u>more resources</u>, but <u>less efficiently</u>
    - not generally true of SPMD computations
  - We care about *<u>miles per hour</u> first, not <u>miles per gallon</u> !*


Unclassified

Target Platform

- **Symmetric, scalable, distributed-memory cluster**
  - **Preferably not LAN-based or Internet-connected systems**
    - **The latency almost always kills performance except in special cases**
  - **Shared memory or multicore OK**
    - **But of course shared memory does not scale**
    - **Good solutions in distributed memory lead to good solutions in shared memory, but not the other way around**
      - **Lots of important optimizations and simplifications are possible with shared memory**
  - **Mixed multi-node and multicore parallelism becoming the norm**
  - **General purpose PDES simulators today have no way to make use of GPUs (though event code in the model might)**
- **One-sided, reliable, asynchronous message communication**
  - **Best built on top of native packet delivery system, rather than over MPI**

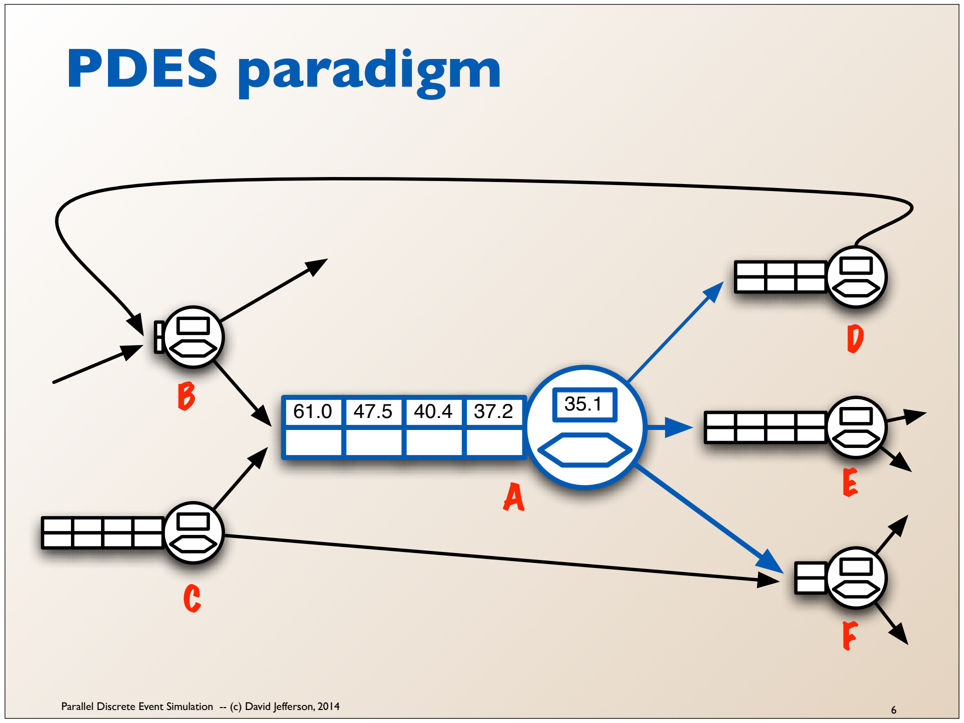
Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 Unclassified 5

We target our algorithms for the distributed memory architecture because
(a) the problem is more difficult than in shared memory, as several important simplifications logical and performance apply in shared memory, and
(b) we are interested in *scalable* simulation mechanisms

Now that multicore machines are nearly universal, there is a need to allow many objects (LPs) to *reside* in the memory of the same node, and we will discuss that later in the course.

But even then, we will not allow shared variables between two objects. It is not that it is impossible to define it, but it so greatly complicates synchronization algorithms that it is not worth it, and it risks nondeterminism.

PDES paradigm



Each circle with its input queue is a simulation object or process or logical process in the simulation. There may be millions of such objects in a big simulation

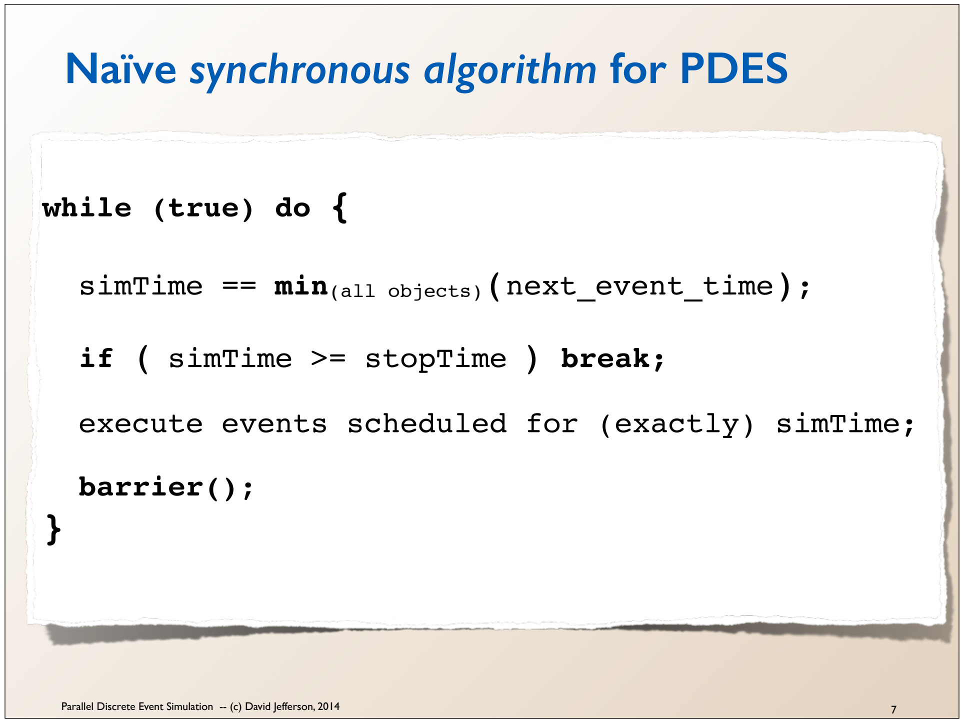
Each double rectangle is an event message, indicating a method to be called and parameters to be passed to the method, and the simulation time at which it should be done. Hexagons represent the state of an object

Each object has its own event message queue, like the event list of the sequential DES algorithm. It is sorted by the timestamp of the message. (Of course it is not really a linear list)

Each object has its own simulation clock that tells its simTime (or virtual time). The process in the center of this diagram is processing an event message whose time stamp is 35.1; that process is now simulating at time 35.1. Other clocks need not necessarily agree -- some will be ahead and others will be behind.

Objects all execute mostly asynchronously, and in parallel, but are synchronized with one another to maintain causal consistency. (The reason will be clear later: if you attempt to keep all objects time synchronized, you get no parallelism!)

The global event list of the sequential algorithm have been divided into many event queues, on for each object, so it is no longer a sequential bottleneck. Likewise, the single, global simulation clock has been quasi-replicated into many individual simulation clocks, one for each object, so it is not a sequential bottleneck either.



Naïve *synchronous algorithm* for PDES

```
while (true) do {

  simTime == min(all objects)(next_event_time);

  if ( simTime >= stopTime ) break;

  execute events scheduled for (exactly) simTime;

  barrier();
}
```

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 7

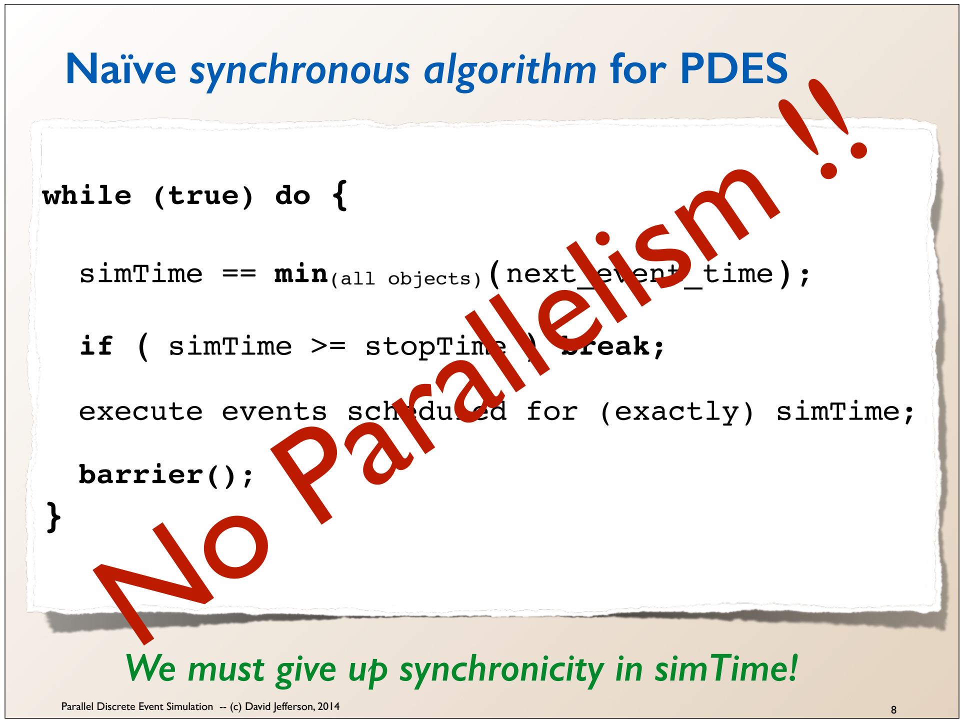
Loop body has 3 global collective operations:

1st line: Calculation of minTime is a global reduction to find the minimum simulation time of any as-yet-unexecuted event message globally
2nd line:Should we terminate?
3rd line: Execute all events scheduled for (exactly) the simTime just calculated.
Note that more than one object may have event messages tied at the minTime and each of them may have more than one event message tied at that time. (Obviously we must use appropriate tie-breaking rule — discussed later.)
4th line: Barrier — waits for all events to complete and for all sent event messages to be delivered. (Unnecessary if the min-reduction is done in a way that includes a barrier, as with MPI).

This is the kind of algorithm that might occur to a naïve programmer who is familiar with SPMD-style scientific programming. It has the property that no event at simTime == t starts executing before all events with simTime < t are finished.

This algorithm is correct, i.e. equivalent to the sequential algorithm (up to tie-breaking rule).

Unfortunately, because ties are so rare, it is in most cases essentially a *sequential* algorithm, regardless of how many cores or processors are applied. Unless there are many ties at each distinct simTime, and unless those ties are spread more or less evenly across all cores/processors, then there is essentially no parallelism achieved.



Naïve *synchronous algorithm* for PDES

```
while (true) do {
  simTime == min(all objects)(next_event_time);
  if ( simTime >= stopTime ) break;
  execute events scheduled for (exactly) simTime;
  barrier();
}
```

No Parallelism !!

We must give up synchronicity in simTime!

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

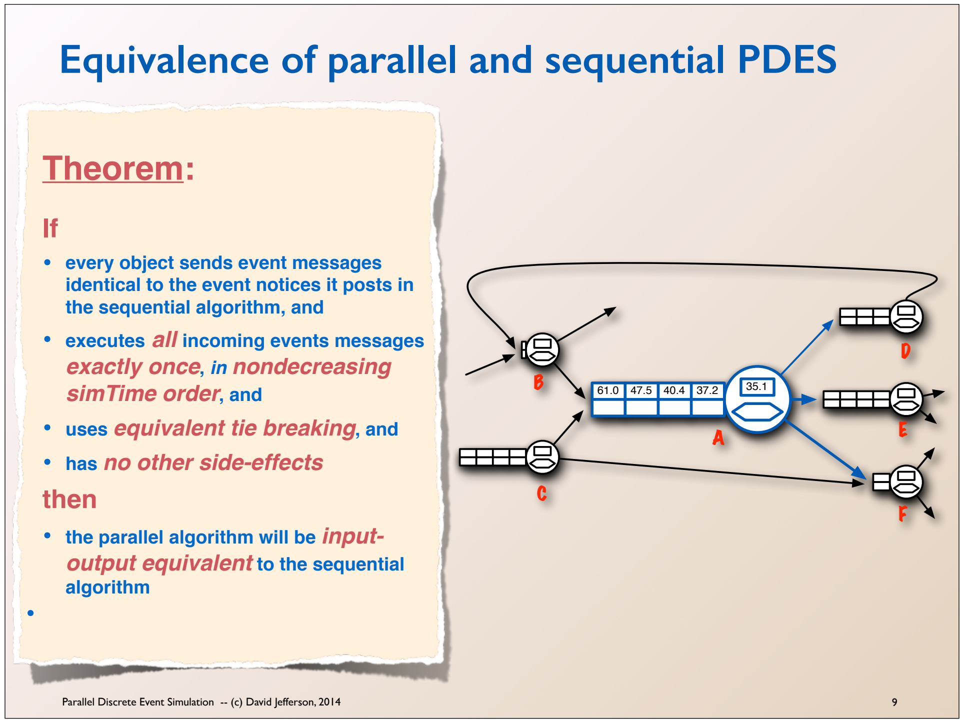
8

The problem is the hard synchronization property that no event at simTime == t starts executing before all events with simTime < t are finished

We must give up that property.

We must allow some events to execute before others with lower timestamps, as long as the causality constraints are not violated, or else we generally get no parallelism!

This is why the objects' simulation clocks and are not kept synchronized with one another. It would force sequentialization.



Equivalence of parallel and sequential PDES

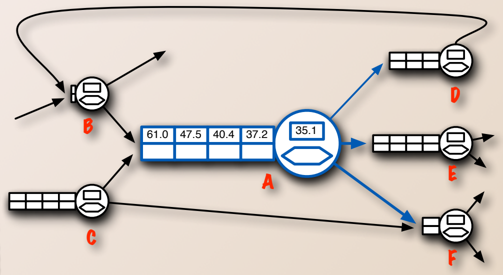
Theorem:

If

- every object sends event messages identical to the event notices it posts in the sequential algorithm, and
- executes *all* incoming events messages *exactly once*, *in nondecreasing simTime order*, and
- uses *equivalent tie breaking*, and
- has *no other side-effects*

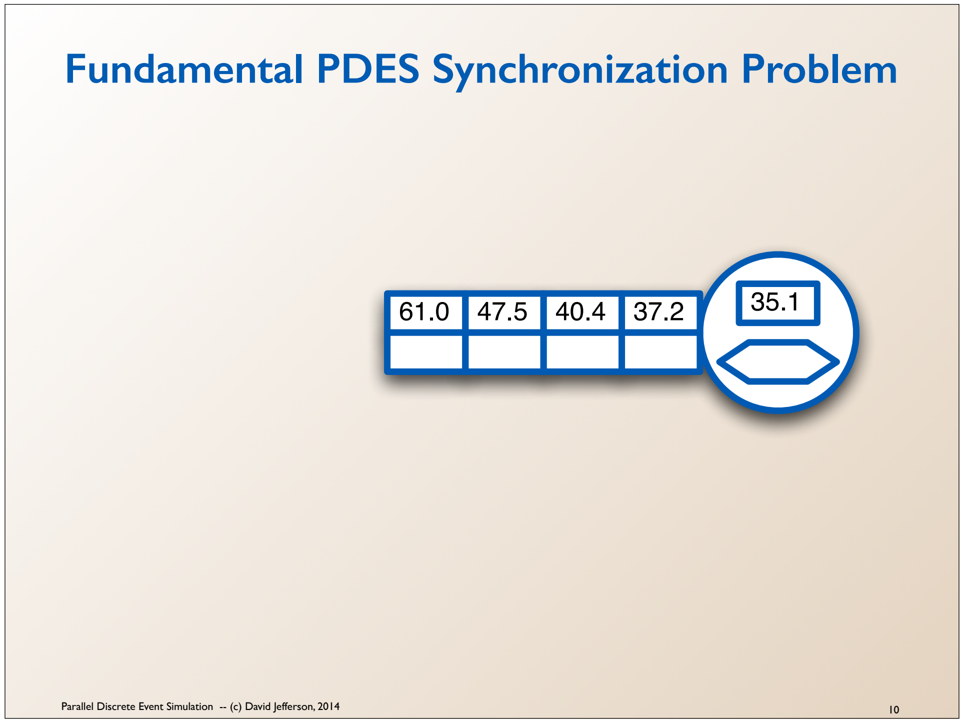
then

- the parallel algorithm will be *input-output equivalent* to the sequential algorithm
-

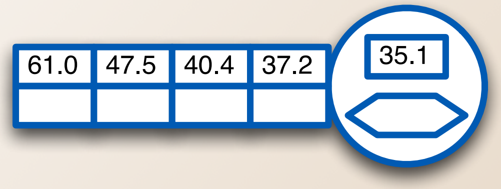


Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 9

This theorem is no-doubt in Euclid somewhere! :-)



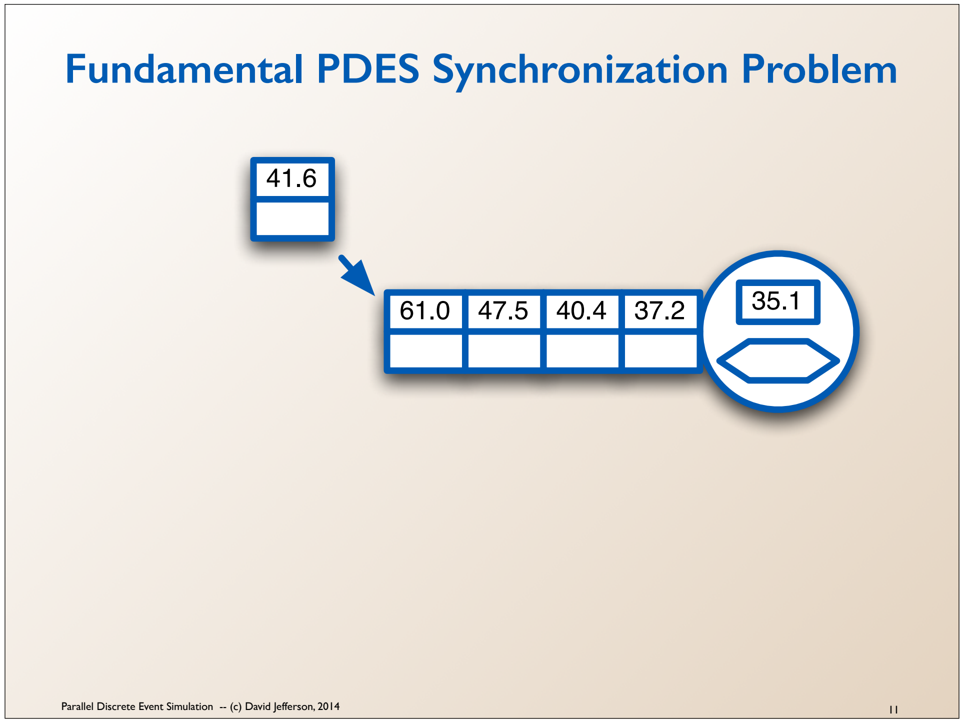
Fundamental PDES Synchronization Problem



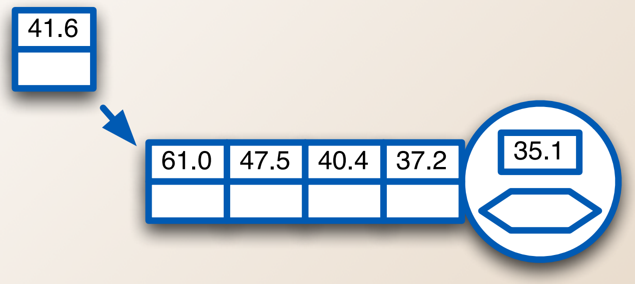
Looking at one single object on its processor node

Event messages arrive asynchronously from other objects on other nodes

They do not necessarily arrive in increasing timestamp order, but they are inserted in the queue sorted that way



Fundamental PDES Synchronization Problem

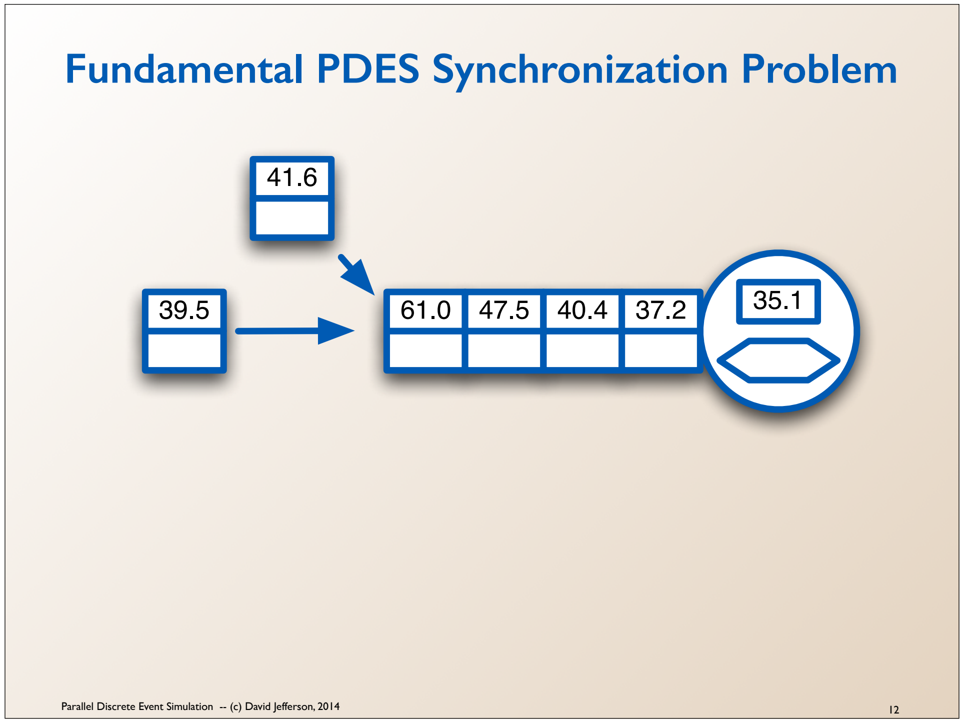


Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 11

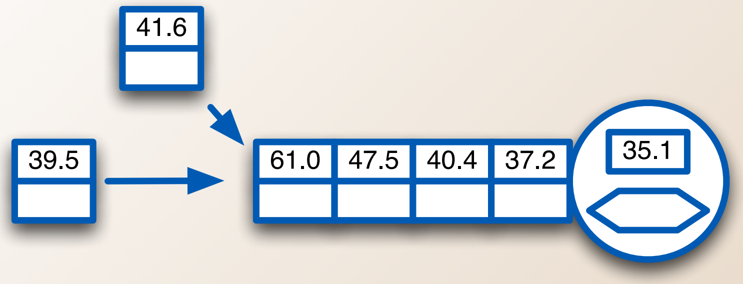
This object MUST execute its event messages in strictly increasing time order

So after finishing event 35.2, what is the next event message its should execute?

Considering that 41.6 is in the way, it might appear that it is safe to execute 37.2 and 40.4

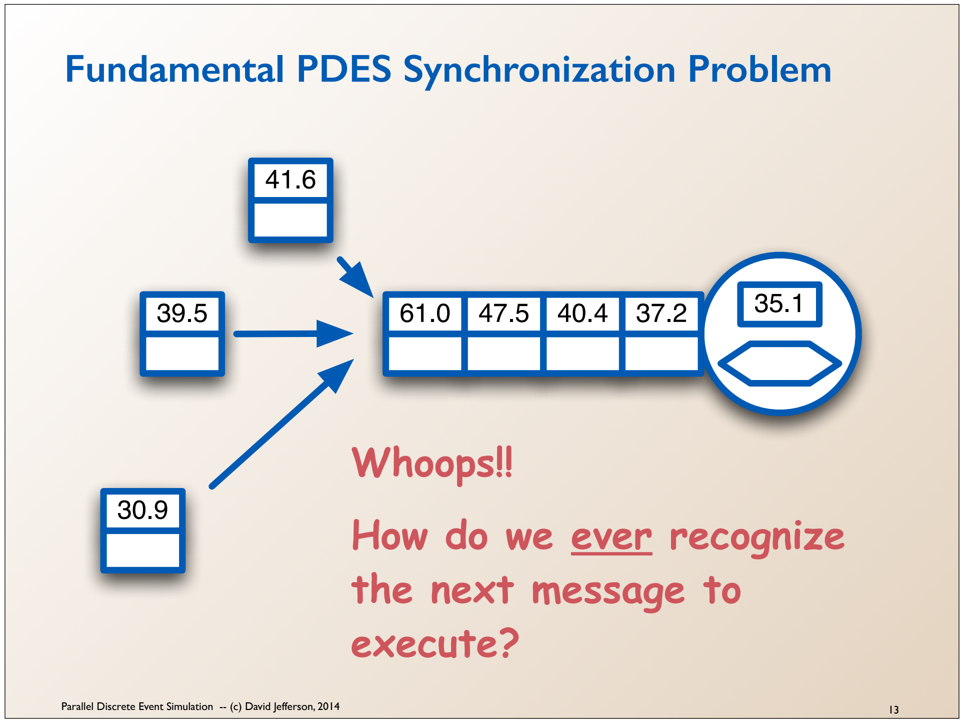


Fundamental PDES Synchronization Problem

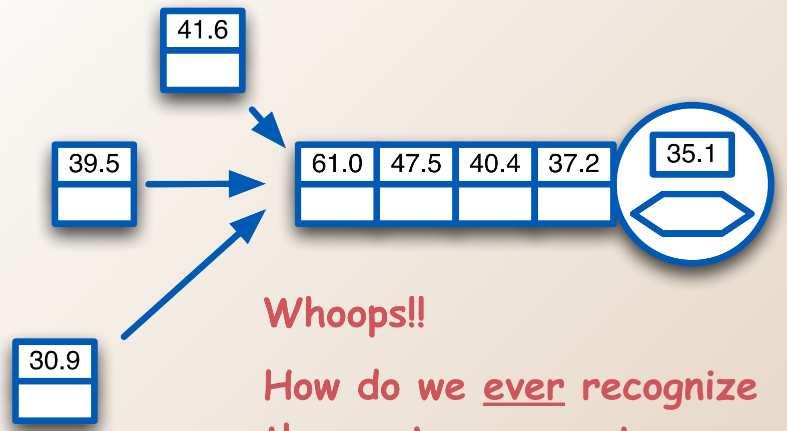


But no, it cannot execute 40.4 because unbeknownst to this process, a message with time stamp 39.5 is on the way!

But at least we can execute 37.2 next!



Fundamental PDES Synchronization Problem



Whoops!!

How do we <u>ever</u> recognize the next message to execute?

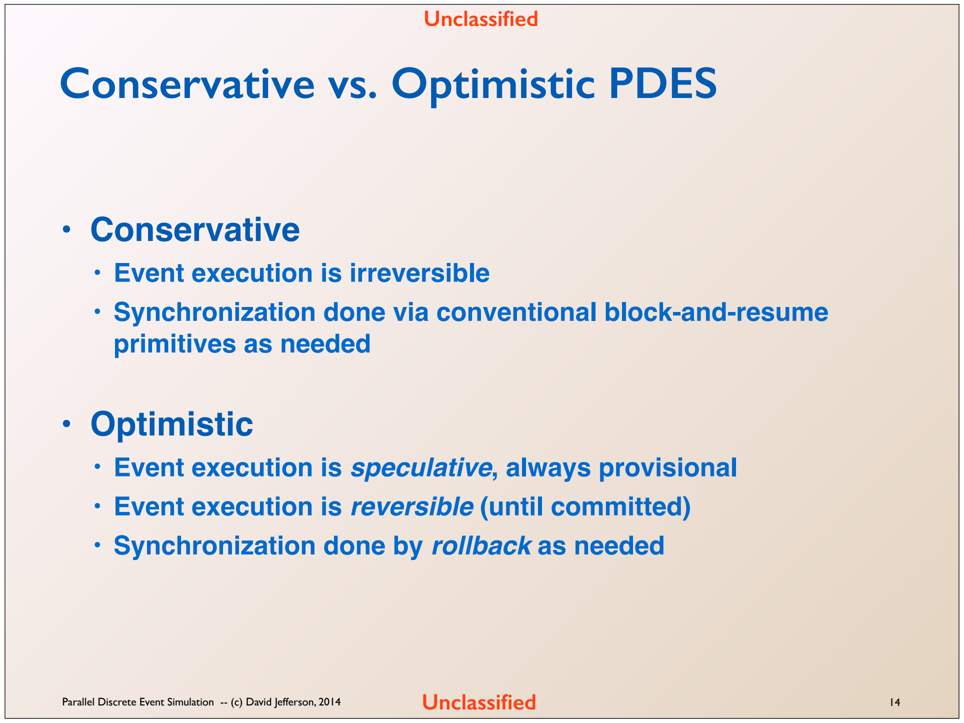
Oh, but wait! We cannot execute 37.2, and in fact, we should not have executed 35.1, because a message with timestamp 30.9 is going to arrive!

How can you even recognize the next event to be executed????

When can you execute any events at all?

Two broad kinds of solutions:

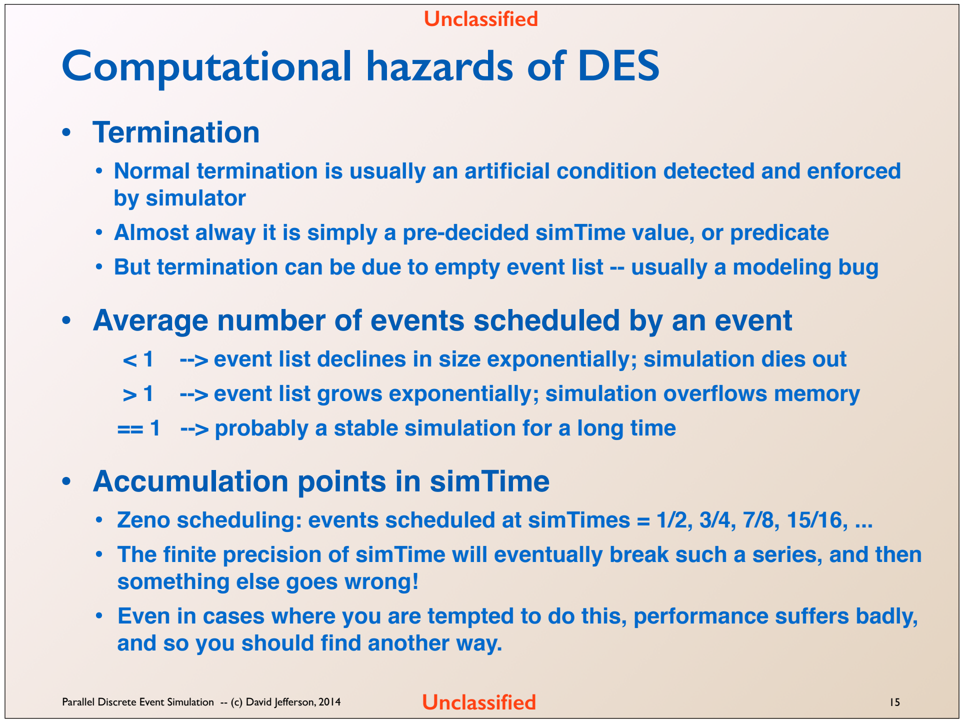
- Irreversible events: conservative
- Reversible events: optimistic



Conservative vs. Optimistic PDES

- Conservative
  - Event execution is irreversible
  - Synchronization done via conventional block-and-resume primitives as needed
- Optimistic
  - Event execution is *speculative*, always provisional
  - Event execution is *reversible* (until committed)
  - Synchronization done by *rollback* as needed

Conservative synchronization might be compared to a country in which cars have forward gears and brakes, but no reverse gears. Think of the trouble that causes: head-in parking, home driveways and garages, parallel parking. Think of deadlocks on narrow roads, bridges and tunnels, or Manhattan traffic.


Unclassified

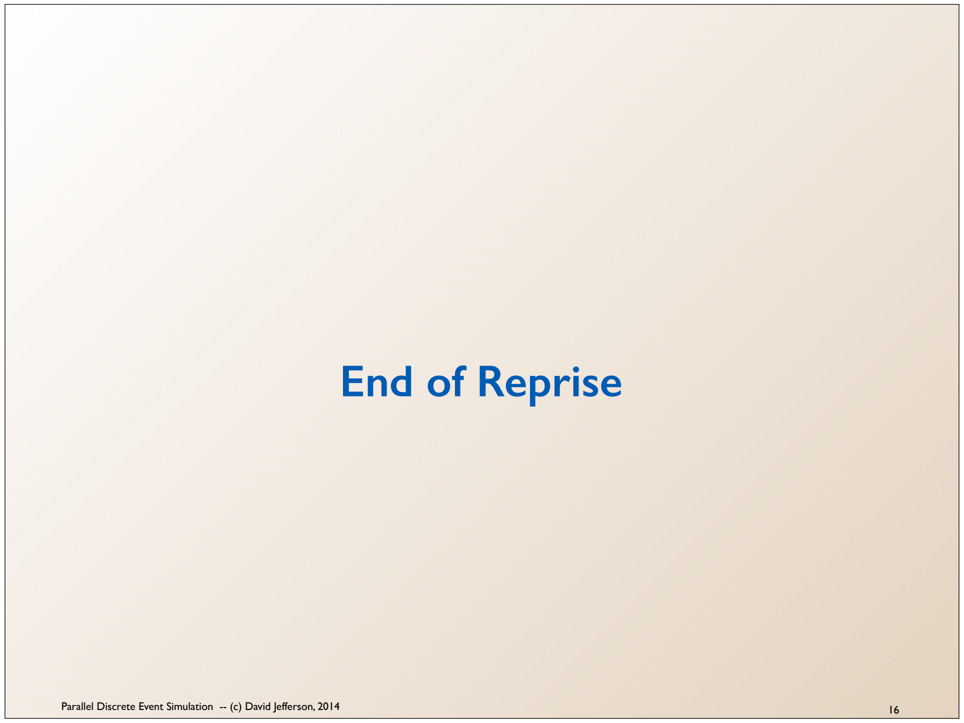
Computational hazards of DES

- **Termination**
  - Normal termination is usually an artificial condition detected and enforced by simulator
  - Almost alway it is simply a pre-decided simTime value, or predicate
  - But termination can be due to empty event list -- usually a modeling bug
- **Average number of events scheduled by an event**
  - < 1 --> event list declines in size exponentially; simulation dies out
  - > 1 --> event list grows exponentially; simulation overflows memory
  - == 1 --> probably a stable simulation for a long time
- **Accumulation points in simTime**
  - Zeno scheduling: events scheduled at simTimes = 1/2, 3/4, 7/8, 15/16, ...
  - The finite precision of simTime will eventually break such a series, and then something else goes wrong!
  - Even in cases where you are tempted to do this, performance suffers badly, and so you should find another way.

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 Unclassified 15

These are modeling problems that the designer of a model should be conscious of We will not say more about these issues. They apply equally well to sequential and all kinds of parallel DES algorithms. We only want to guarantee that whatever behavior the sequential algorithm has in these cases the parallel algorithm has as well.

"Zeno scheduling" (my term — not a standard term) arises in a situation where, for example, two physical objects in space are on a collision course, and it is important in the simulation to calculate *precisely* when collision occurs, and exactly what parts of the object touch with what energy and momentum, etc. In order to narrow down the exact time of collision, events with shorter and shorter delays may be scheduled, yielding a Zeno patten of activity.

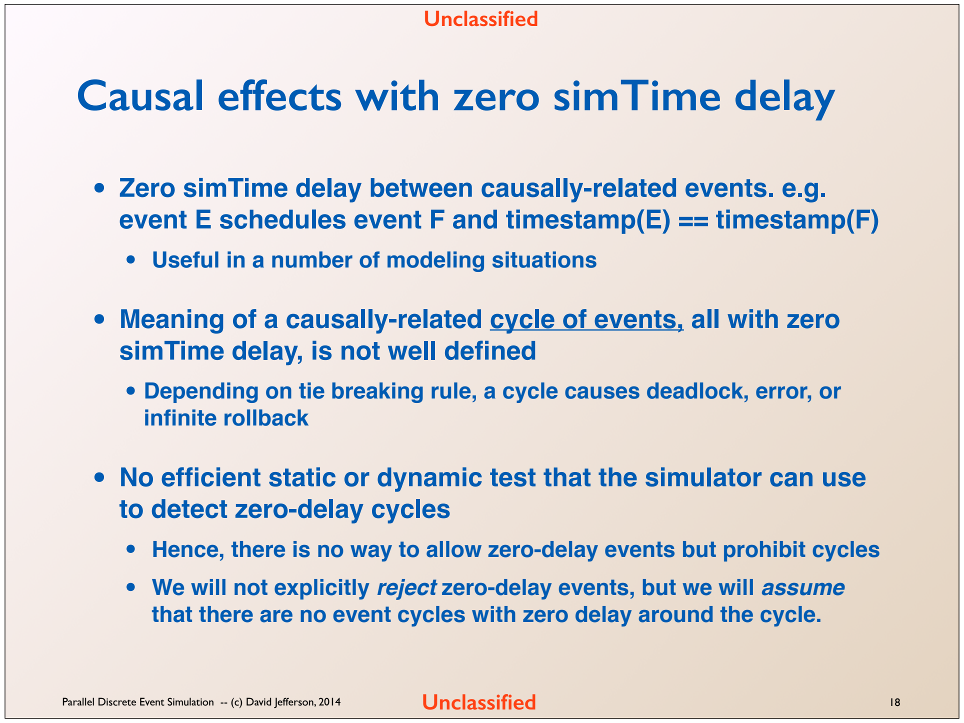


End of Reprise

Ties in simulation time

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

17

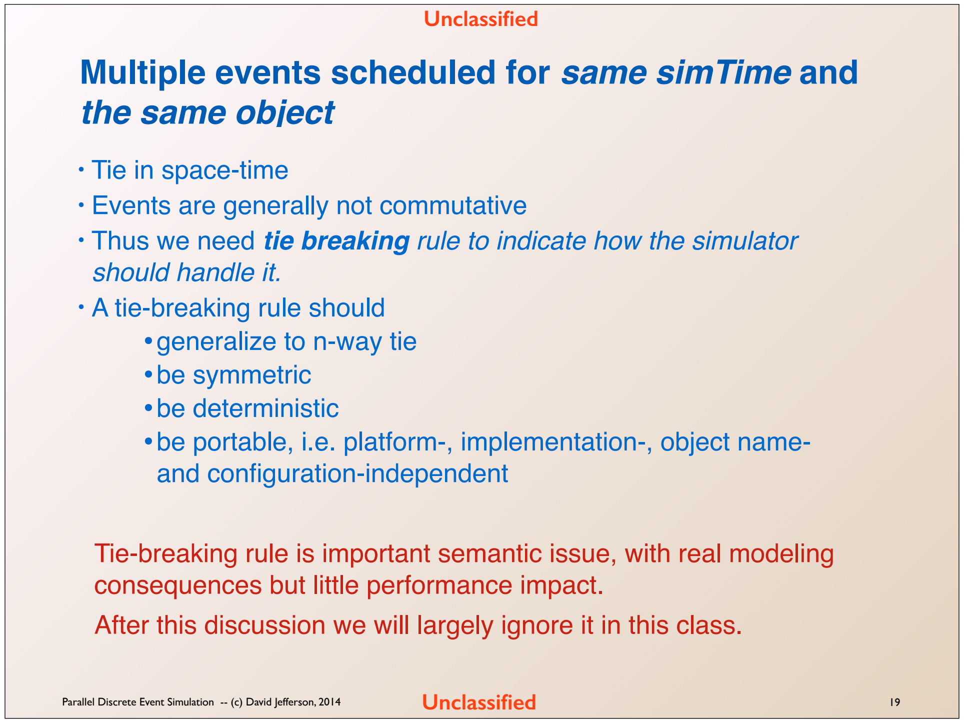


Causal effects with zero simTime delay

- **Zero simTime delay between causally-related events. e.g. event E schedules event F and timestamp(E) == timestamp(F)**
  - **Useful in a number of modeling situations**
- **Meaning of a causally-related cycle of events, all with zero simTime delay, is not well defined**
  - **Depending on tie breaking rule, a cycle causes deadlock, error, or infinite rollback**
- **No efficient static or dynamic test that the simulator can use to detect zero-delay cycles**
  - **Hence, there is no way to allow zero-delay events but prohibit cycles**
  - **We will not explicitly *reject* zero-delay events, but we will *assume* that there are no event cycles with zero delay around the cycle.**

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.

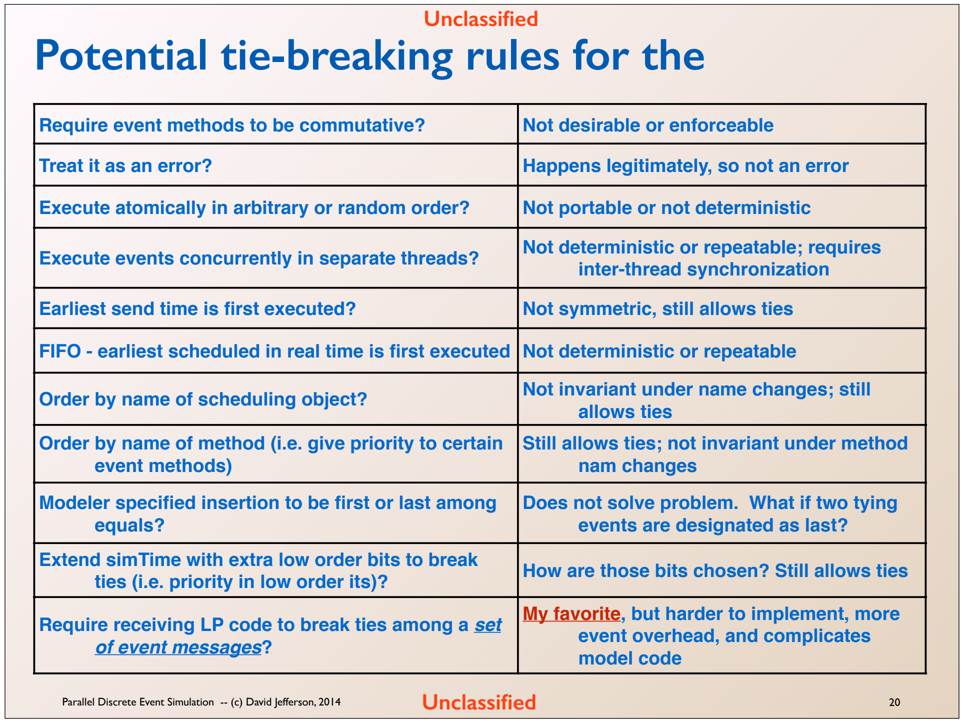


Multiple events scheduled for *same simTime* and *the same object*

- 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

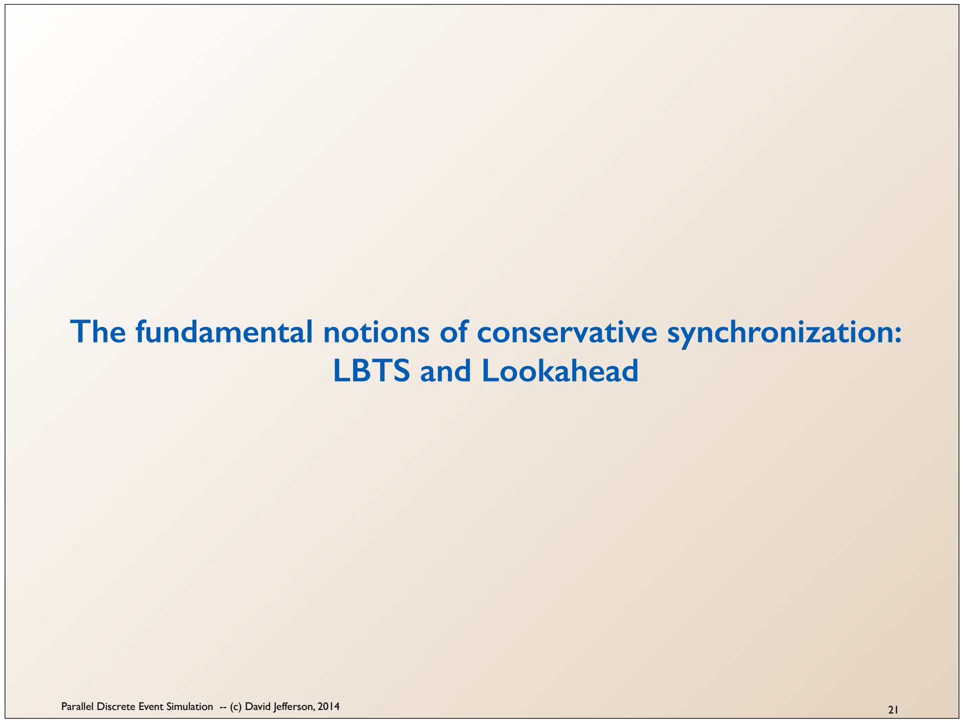
Potential tie-breaking rules for the

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

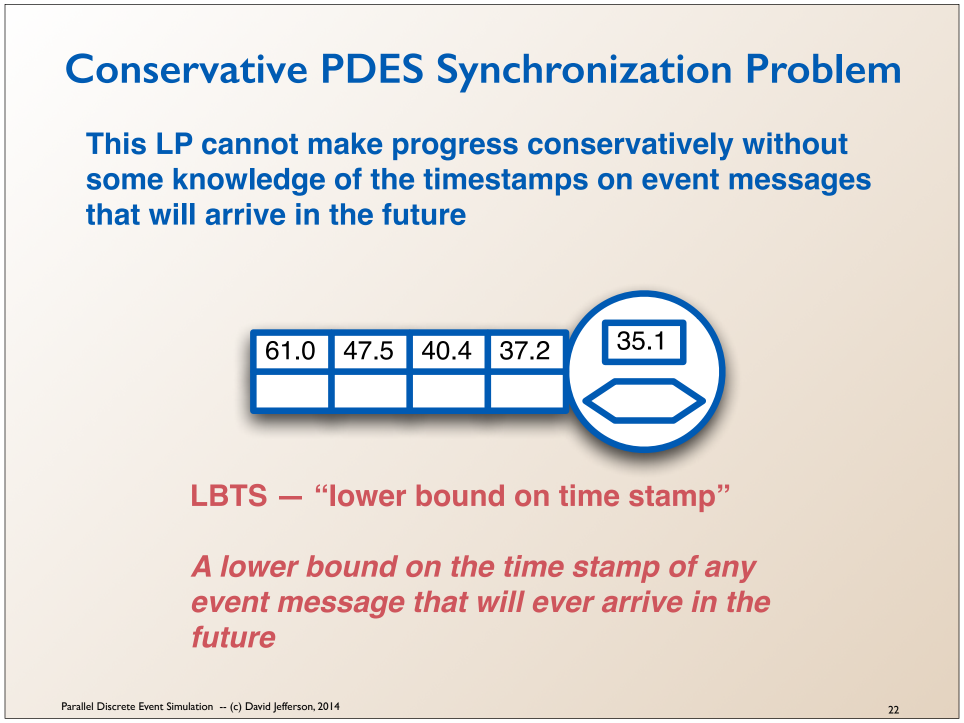
Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 Unclassified 20

Two events that say "Turn left" and Take one 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 same 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.

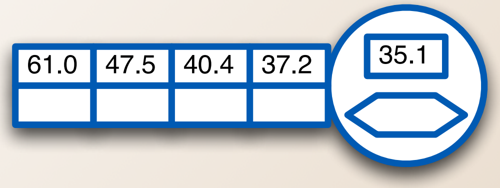


The fundamental notions of conservative synchronization: LBTS and Lookahead



Conservative PDES Synchronization Problem

This LP cannot make progress conservatively without some knowledge of the timestamps on event messages that will arrive in the future



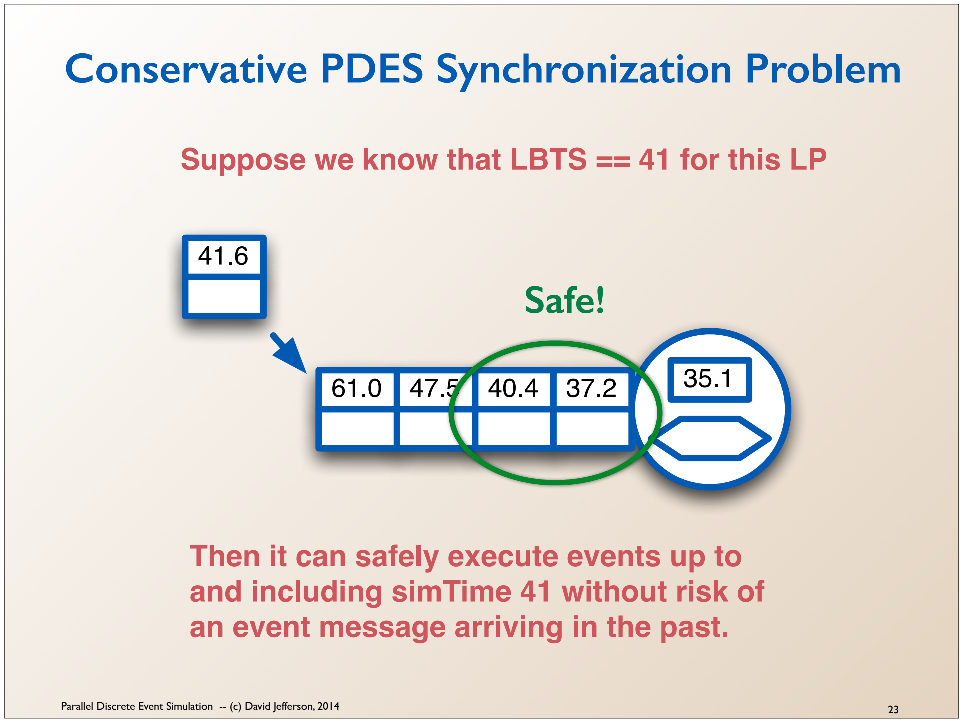
LBTS — "lower bound on time stamp"

A lower bound on the time stamp of any event message that will ever arrive in the future

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

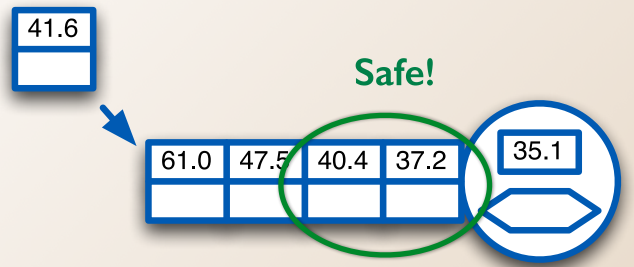
22

LBTS is only a lower bound on the time stamps of future arriving event messages. But it does not have to be the tightest lower bound. Any lower bound greater than the time stamp of the next message in the queue allows progress.



Conservative PDES Synchronization Problem

Suppose we know that LBTS == 41 for this LP



Then it can safely execute events up to and including simTime 41 without risk of an event message arriving in the past.

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

23

Look at one single object on its processor node. Event messages arrive asynchronously from other objects on other nodes. They do not necessarily arrive in increasing timestamp order, but they are inserted in the queue sorted that way.

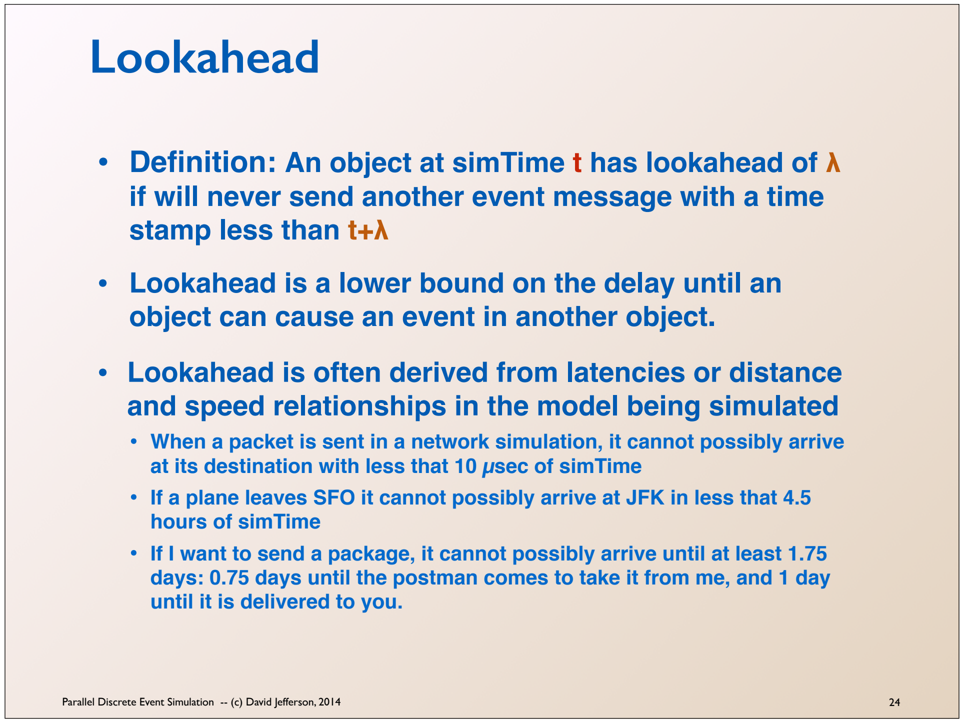
As we have previously discussed, if the object cannot rollback, then it cannot execute any events or make any progress until and unless the simulator gets additional information. The object will just stay blocked forever, and the entire simulation will be in deadlock.

The additional information it needs is a lower bound on the timestamps on all event messages that may arrive in the future (real time future, not simulation time future).

We call that value LBTS — "lower bound on time stamp". In this example, if the LBTS happens to be 41, then the object can safely execute events up to (and if there are no ties, then including) simTime 41, but then it must block until it gets an update to LBTS.

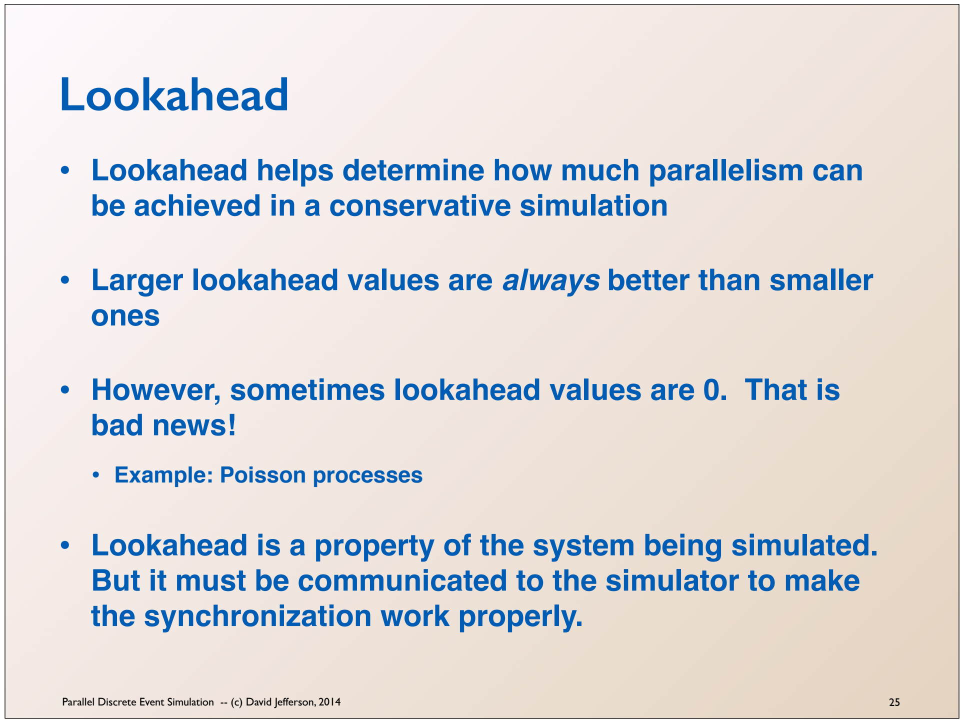
The LBTS for each object must be calculated by the simulator from time to time, in order for the simulation to continue to make progress. By the way LBTS is defined, it simply cannot decrease for any object. However, it may be updated several times without increasing, and it must eventually increase or the simulation will be in deadlock. For well-defined models, however, this is not a problem — it will increase. The only question usually is how fast!

Any time an LBTS is calculated, it must be enforced. If a message ever arrives with a time stamp less than whatever was calculated as LBTS then the simulation has a fatal error.



Lookahead

- Definition: An object at simTime t has lookahead of λ if will never send another event message with a time stamp less than t+λ
- Lookahead is a lower bound on the delay until an object can cause an event in another object.
- Lookahead is often derived from latencies or distance and speed relationships in the model being simulated
  - When a packet is sent in a network simulation, it cannot possibly arrive at its destination with less that 10 *μ*sec of simTime
  - If a plane leaves SFO it cannot possibly arrive at JFK in less that 4.5 hours of simTime
  - If I want to send a package, it cannot possibly arrive until at least 1.75 days: 0.75 days until the postman comes to take it from me, and 1 day until it is delivered to you.

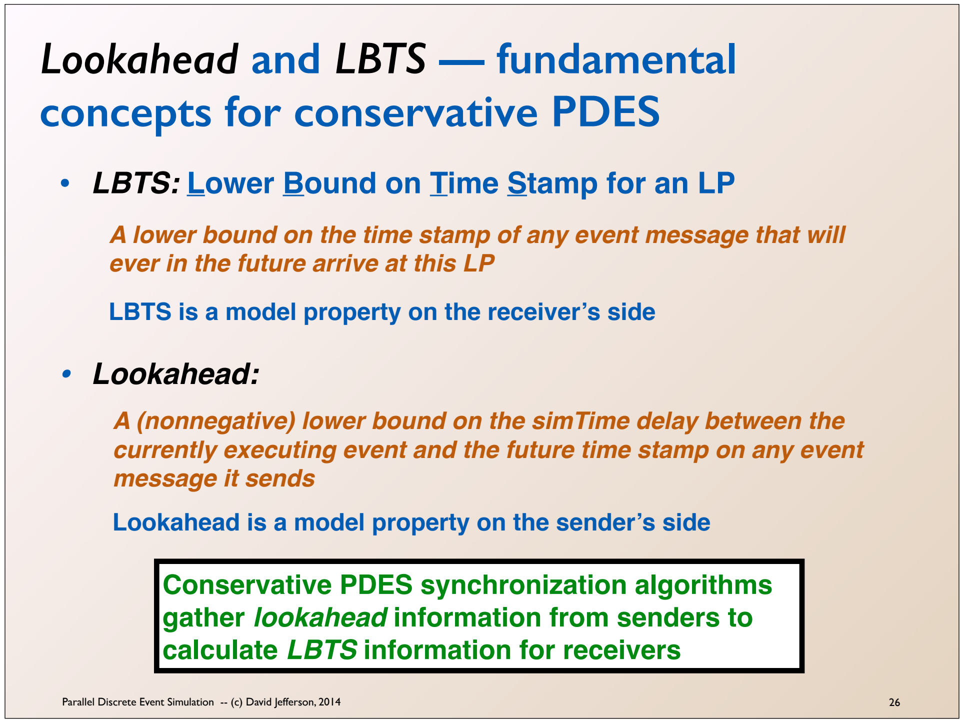


Lookahead

- Lookahead helps determine how much parallelism can be achieved in a conservative simulation
- Larger lookahead values are *always* better than smaller ones
- However, sometimes lookahead values are 0. That is bad news!
  - Example: Poisson processes
- Lookahead is a property of the system being simulated. But it must be communicated to the simulator to make the synchronization work properly.

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

As mentioned earlier, sometimes we allow zero-delay messages. That will not necessarily spoil a good lookahead, however.



Lookahead and *LBTS* — fundamental concepts for conservative PDES

- ***LBTS:*** **Lower Bound on Time Stamp for an LP**

  A lower bound on the time stamp of any event message that will ever in the future arrive at this LP

  LBTS is a model property on the receiver's side

- ***Lookahead:***

  A (nonnegative) lower bound on the simTime delay between the currently executing event and the future time stamp on any event message it sends

  Lookahead is a model property on the sender's side

Conservative PDES synchronization algorithms gather *lookahead* information from senders to calculate *LBTS* information for receivers

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 26

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.*

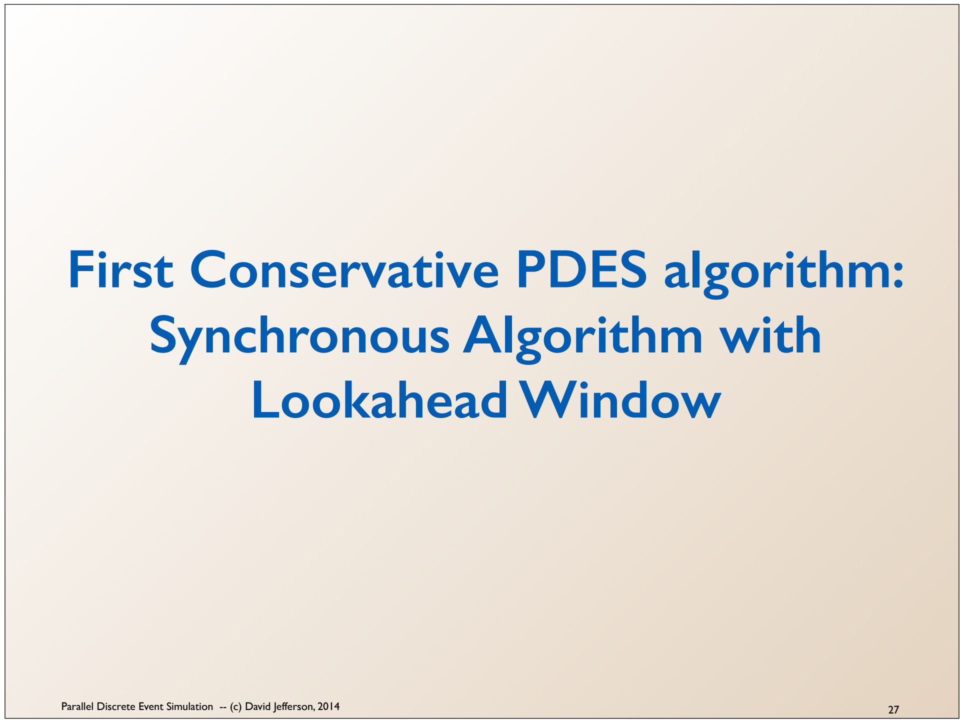
Sender LBTS is a lower bound on any event message time stamp that the LP will ever send from now on.

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

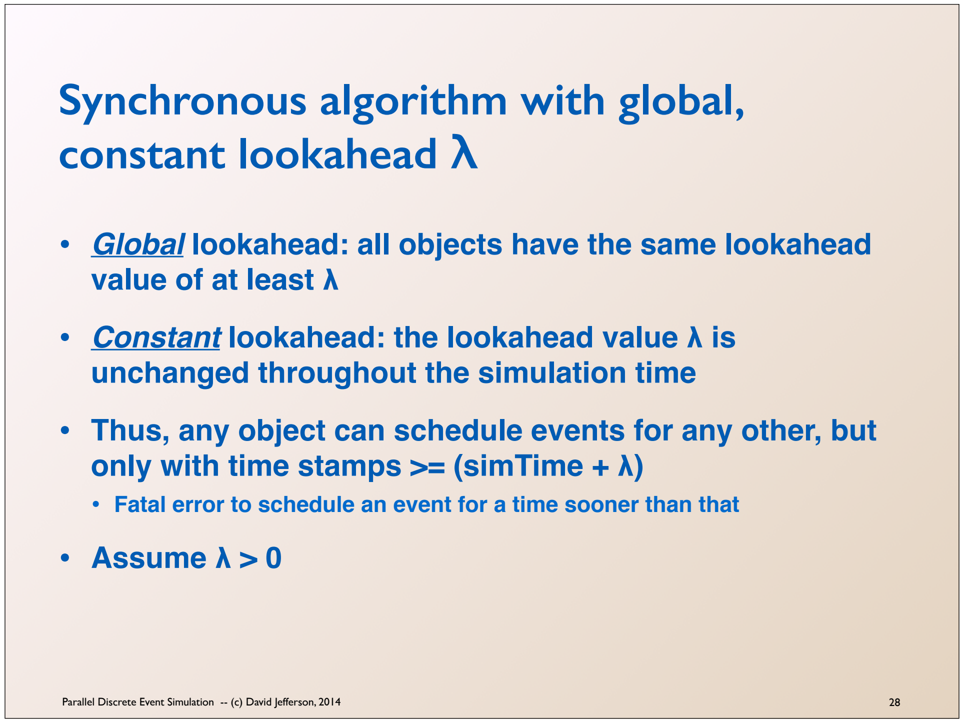
The sender LBTS is just the current simulation time of the LP + its lookahead. This should make it clear that the two notions are complementary.

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



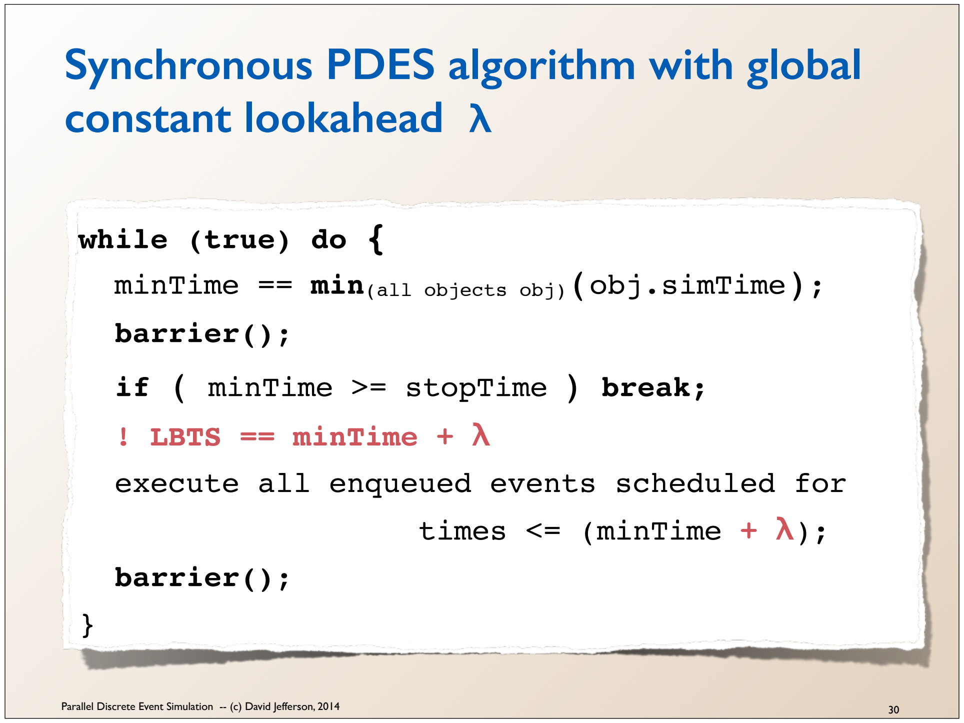
Synchronous algorithm with global, constant lookahead λ

- ***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

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 28

Naïve synchronous PDES algorithm

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

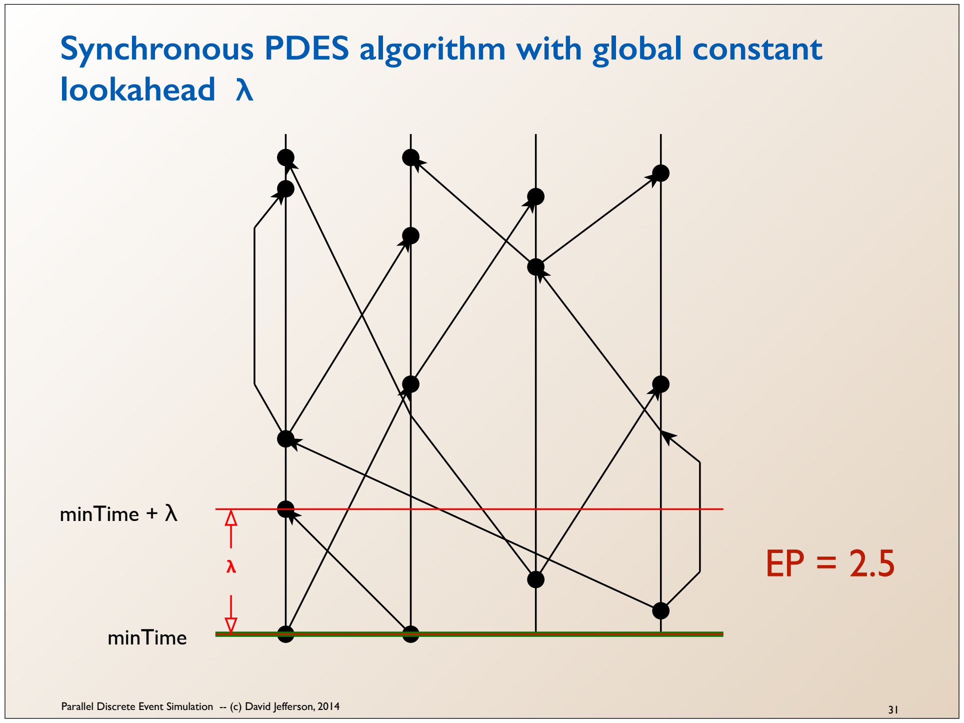


Synchronous PDES algorithm with global constant lookahead λ

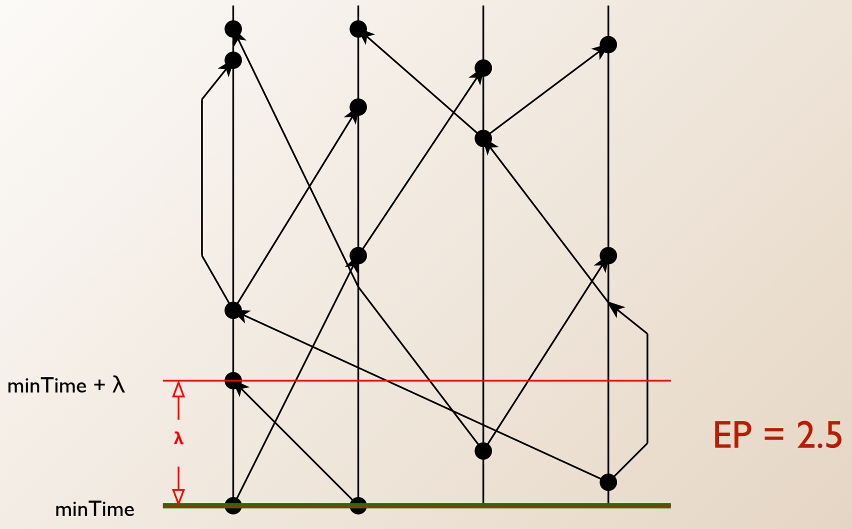
```
while (true) do {
  minTime == min(all objects obj)(obj.simTime);
  barrier();
  if ( minTime >= stopTime ) break;
  ! LBTS == minTime + λ
  execute all enqueued events scheduled for
                    times <= (minTime + λ);
  barrier();
}
```

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

30



Synchronous PDES algorithm with global constant lookahead λ



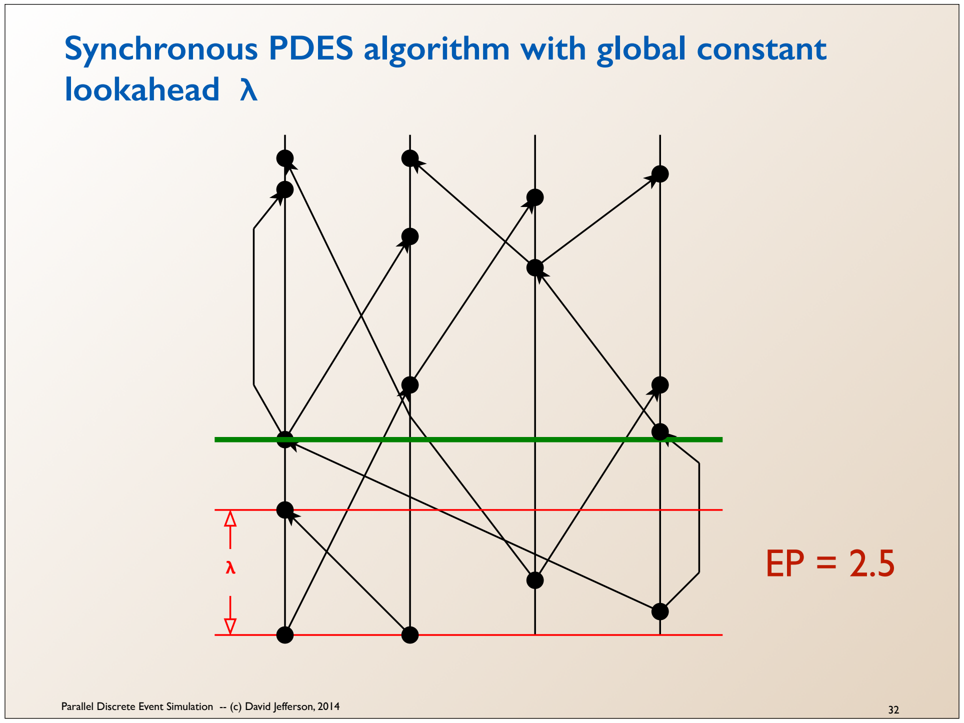
Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 31

Execute all events such that

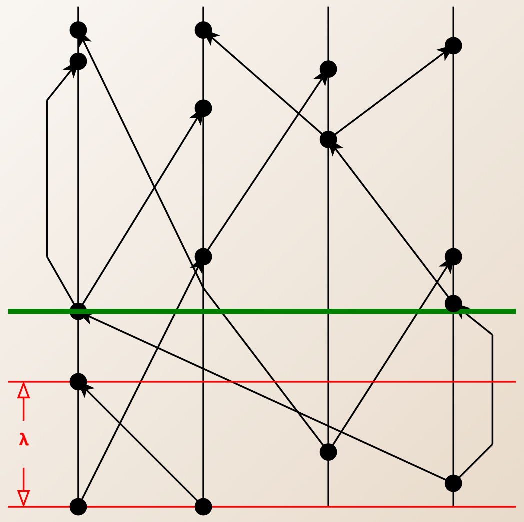
$$\text{minTime} <= t <= \text{minTime} + \lambda$$

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.

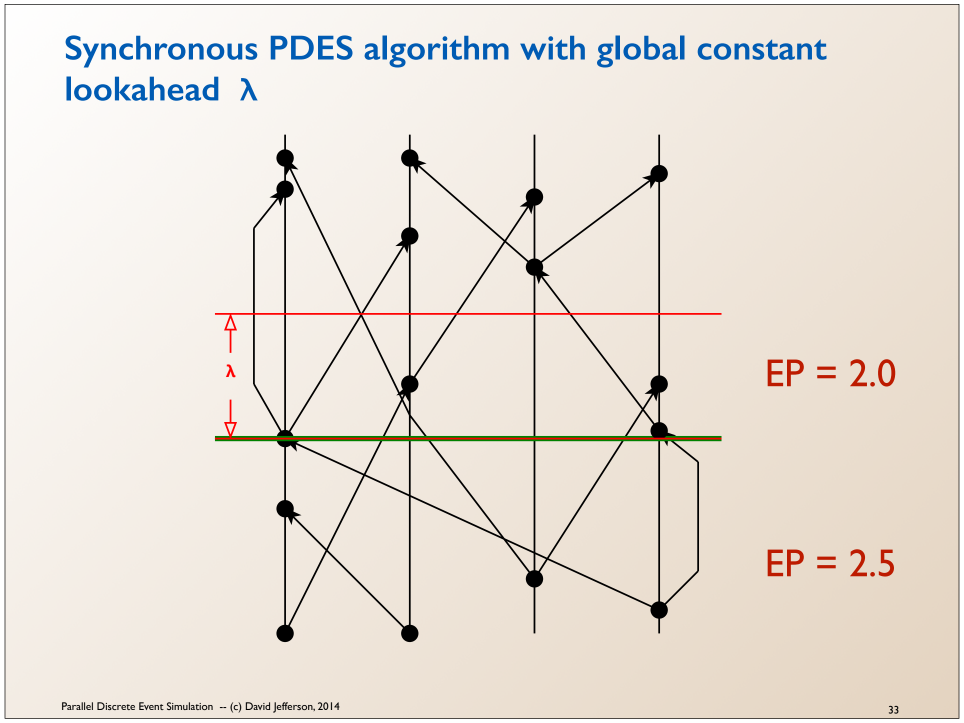


Synchronous PDES algorithm with global constant lookahead λ



EP = 2.5

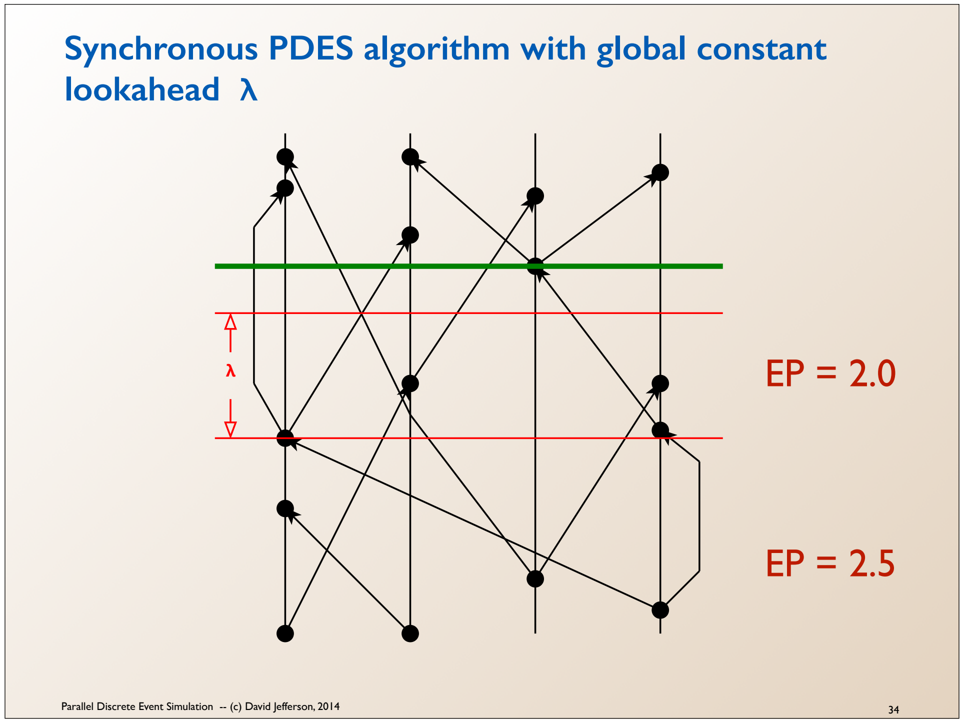
Calculate new minTime (green line)



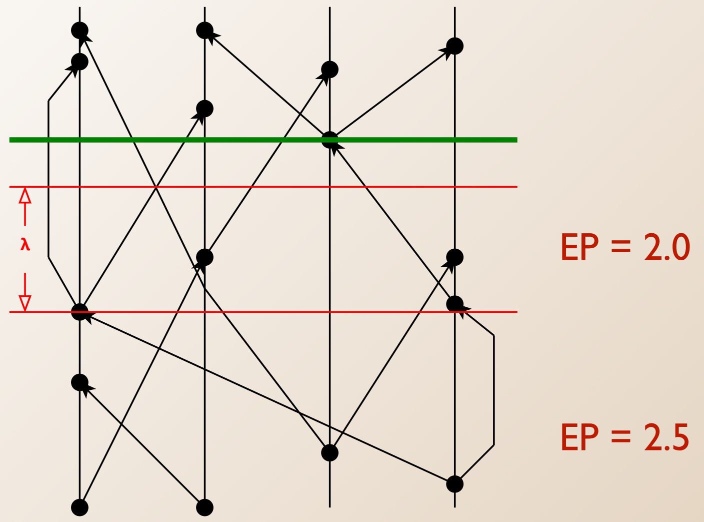
Synchronous PDES algorithm with global constant lookahead λ

Again execute all events such that

$minTime <= t <= minTime + \lambda$



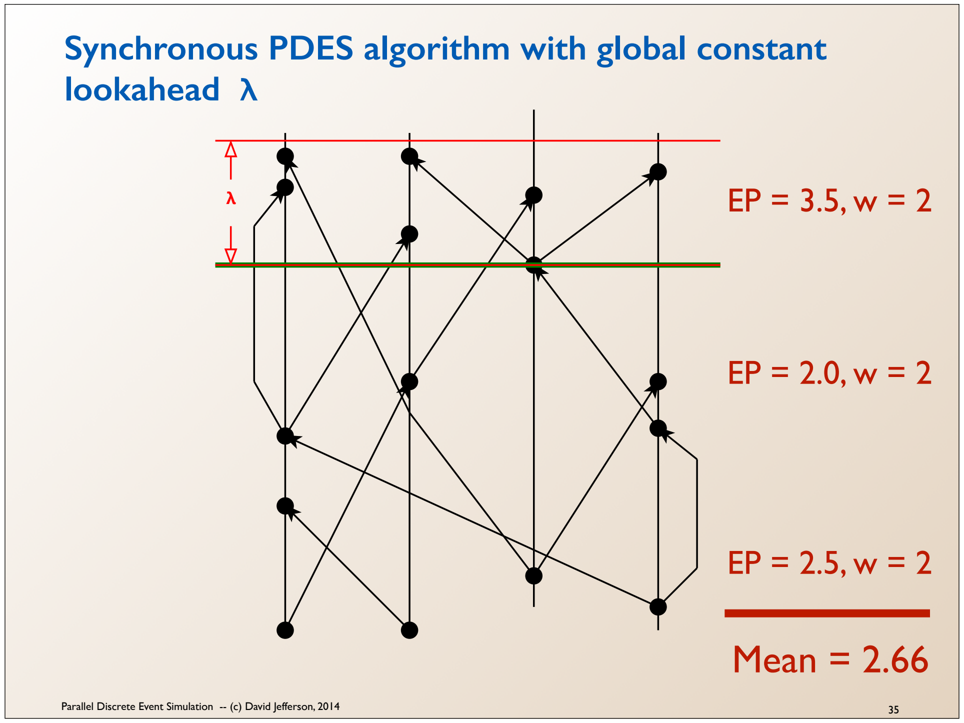
Synchronous PDES algorithm with global constant lookahead λ



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

34

Again calculate new minTime (green line)



Synchronous PDES algorithm with global constant lookahead λ

λ
EP = 3.5, w = 2
EP = 2.0, w = 2
EP = 2.5, w = 2
Mean = 2.66
Parallel Discrete Event Simulation -- (c) David Jefferson, 2014
35

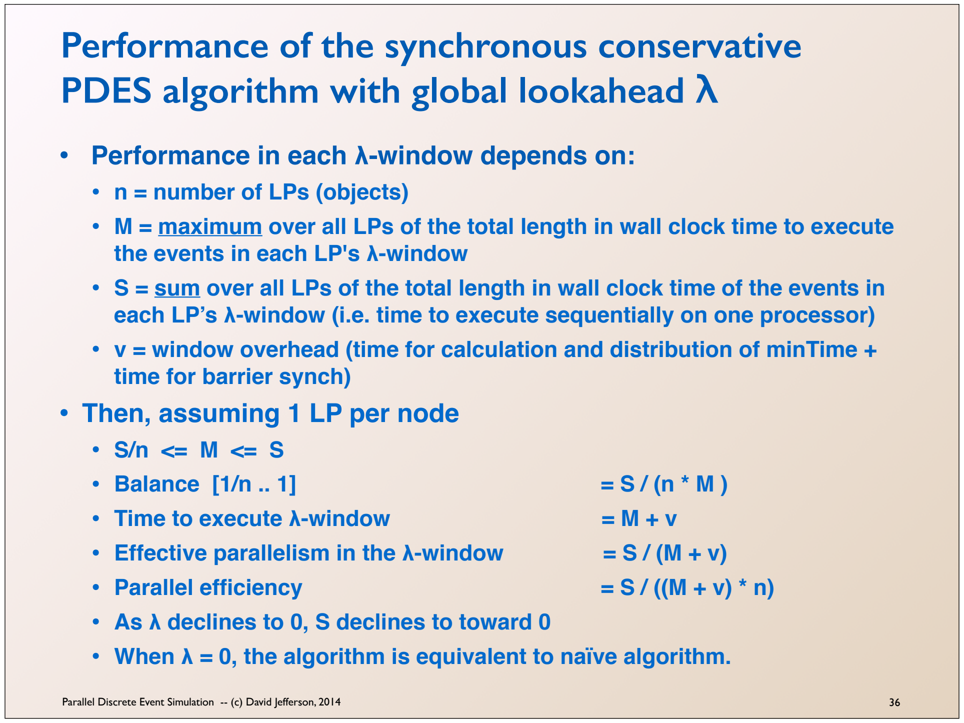
Again execute all events such that

$$\text{minTime} <= t <= \text{minTime} + \lambda$$

… 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.


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.

Parallel Discrete Event Simulation -- (c) David Jefferson, 2014 36

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.

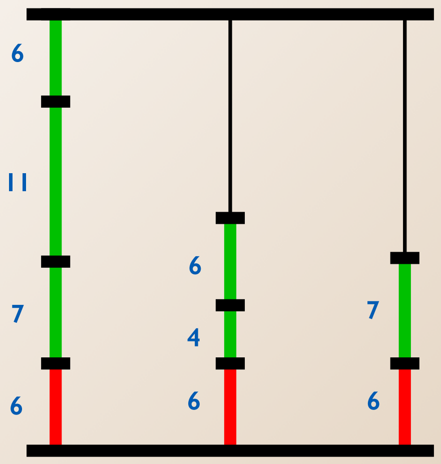
We thus want M to be small compared to S, which is the same as saying we want the simulation to be balanced.

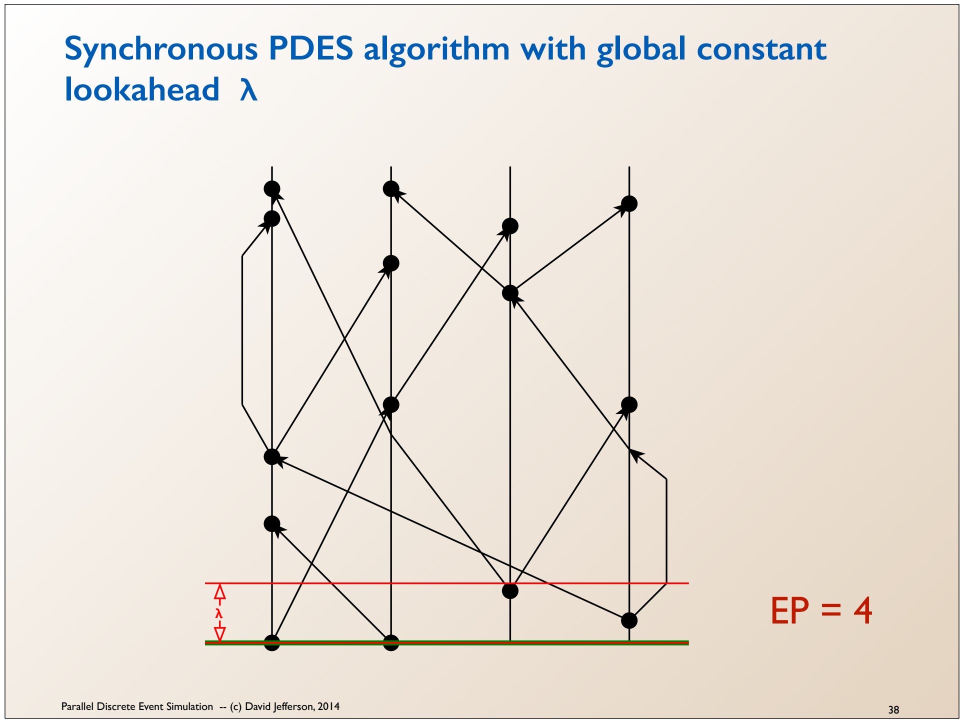
We also want v to be small compared to S because of course v is overhead not much under control of the model programmer.

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.

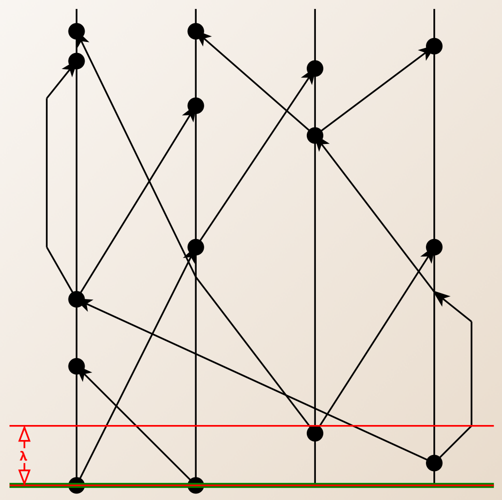
Example performance of a λ-window

```
n = 3
v = 6
M = 24
S = 41

balance
    = S / (n * M)
    = 41/ (3 * 24)
    = 0.569

time to execute
    = M + v
    = 30

effective parallelism
    = S / (M + v)
    = 41 / (24 + 6)
    = 1.37

parallel efficiency
    = S / ((M + v) * n)
    = 41 / ((24+6)*3)
    = 0.456
```





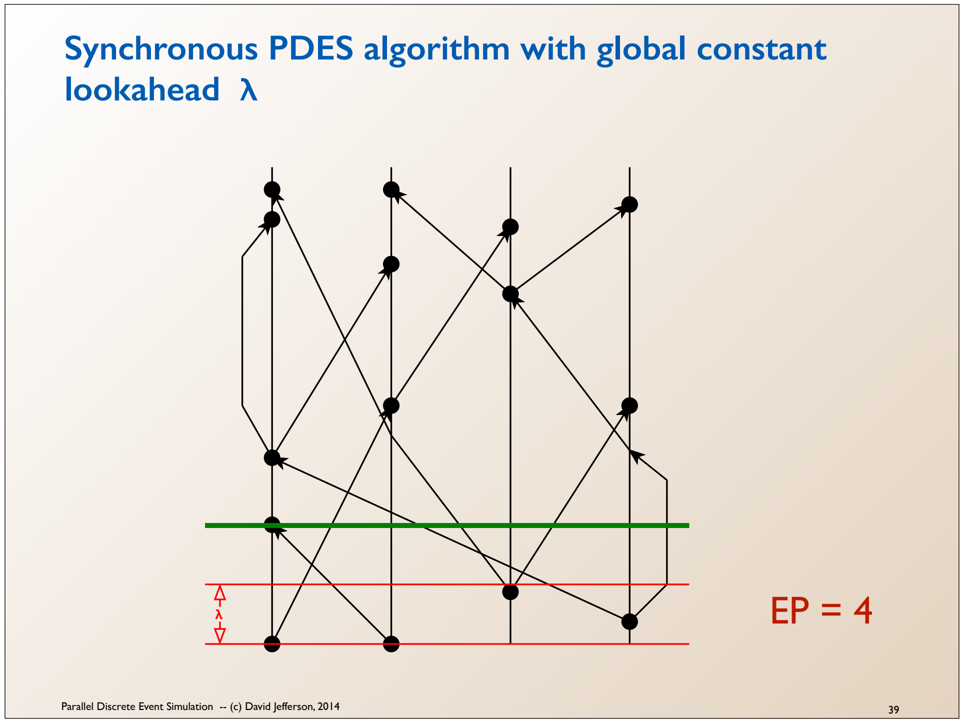
Synchronous PDES algorithm with global constant lookahead λ



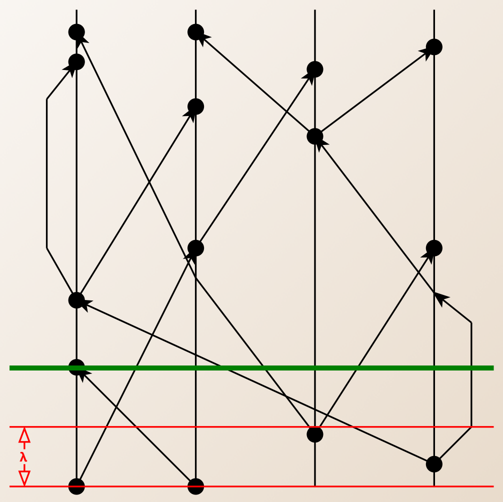
EP = 4

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

38



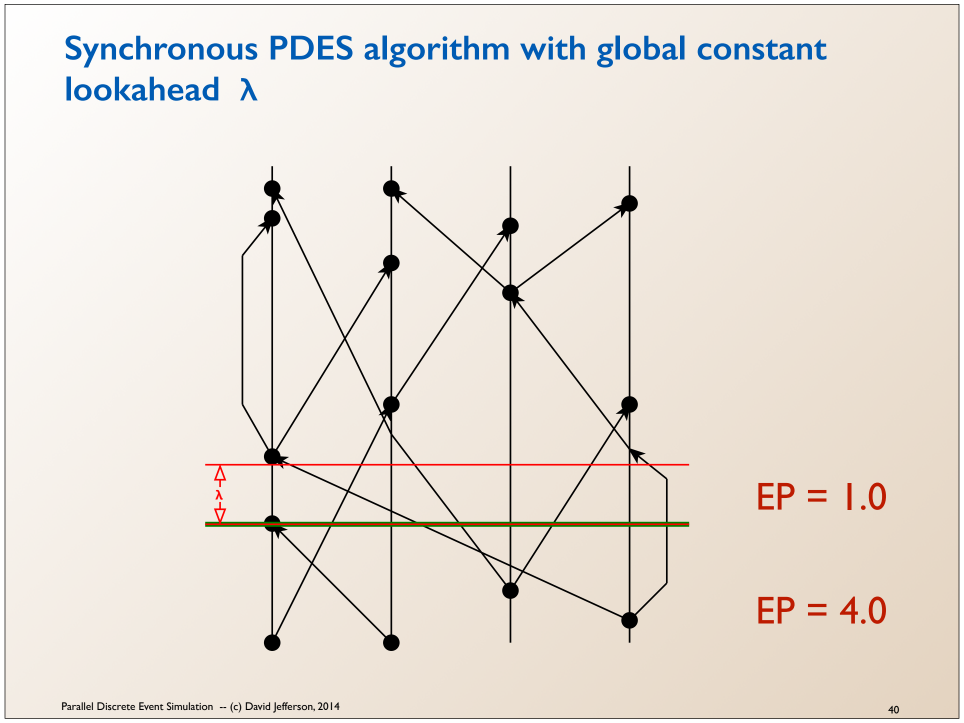
Synchronous PDES algorithm with global constant lookahead λ



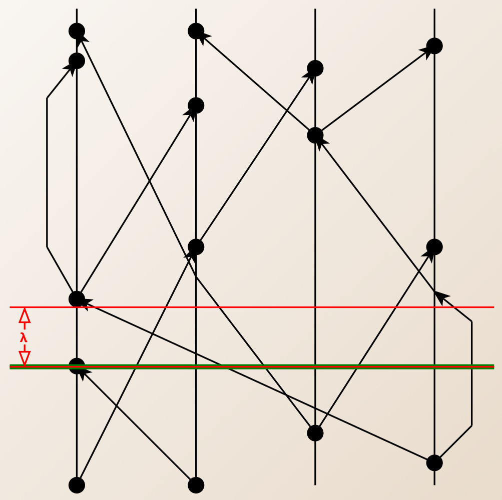
EP = 4

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

39



Synchronous PDES algorithm with global constant lookahead λ



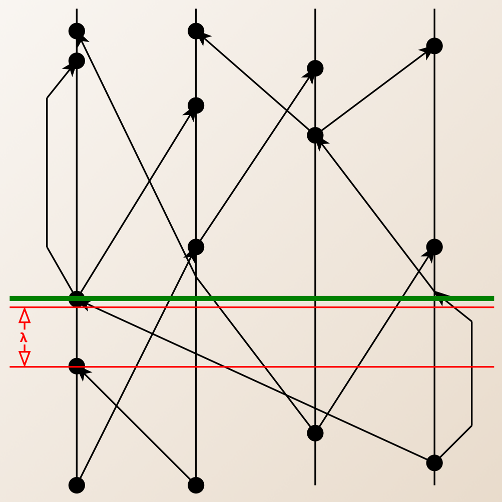
EP = 1.0

EP = 4.0

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

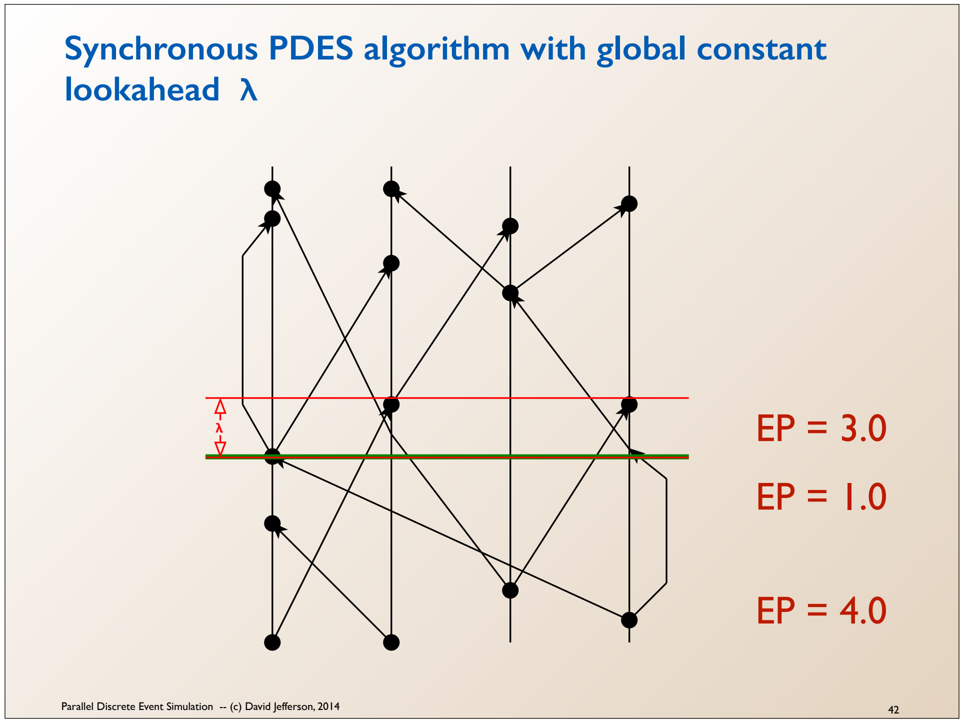
40

Synchronous PDES algorithm with global constant lookahead λ

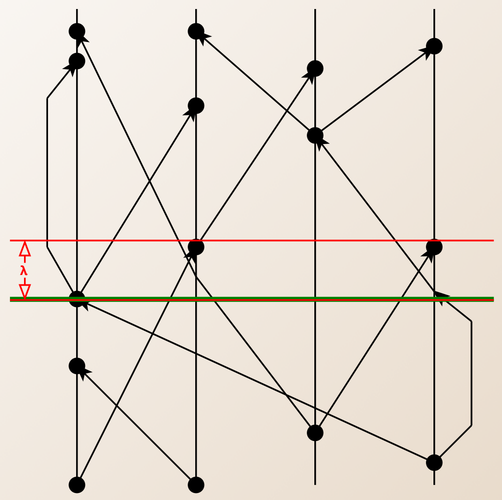


EP = 1.0

EP = 4.0



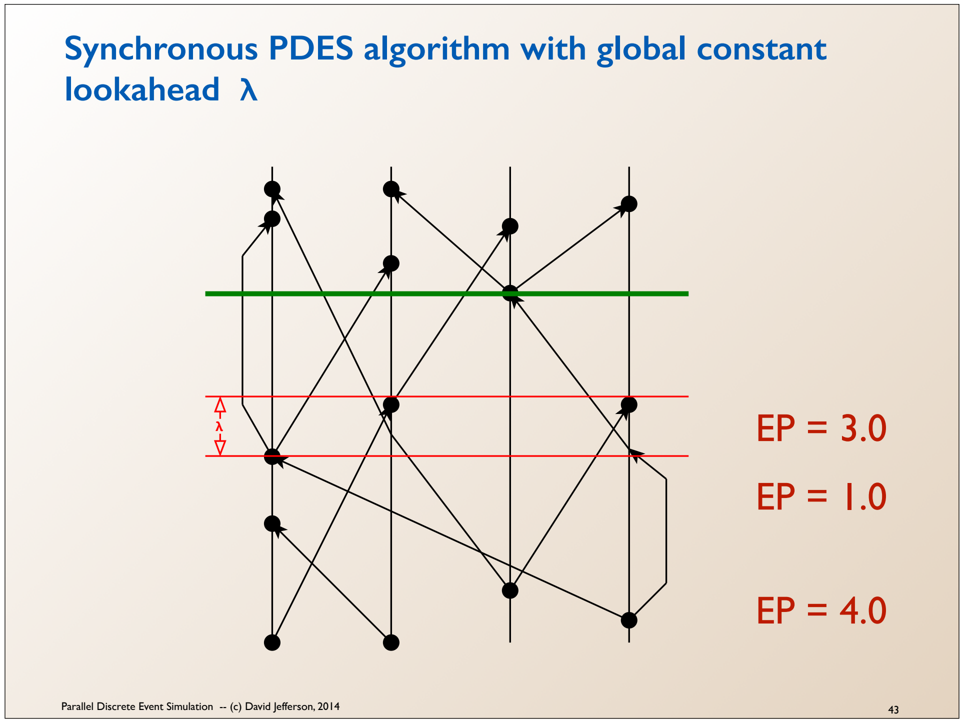
Synchronous PDES algorithm with global constant lookahead λ



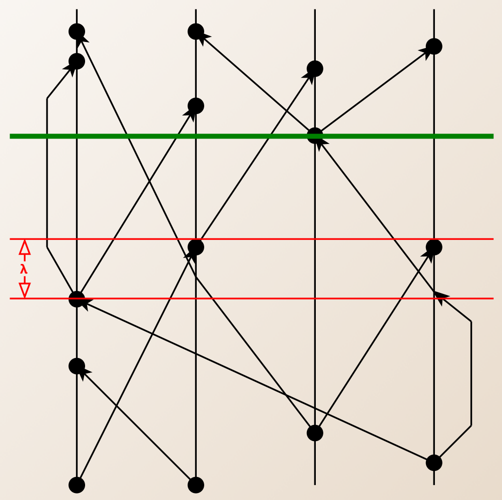
EP = 3.0

EP = 1.0

EP = 4.0



Synchronous PDES algorithm with global constant lookahead λ



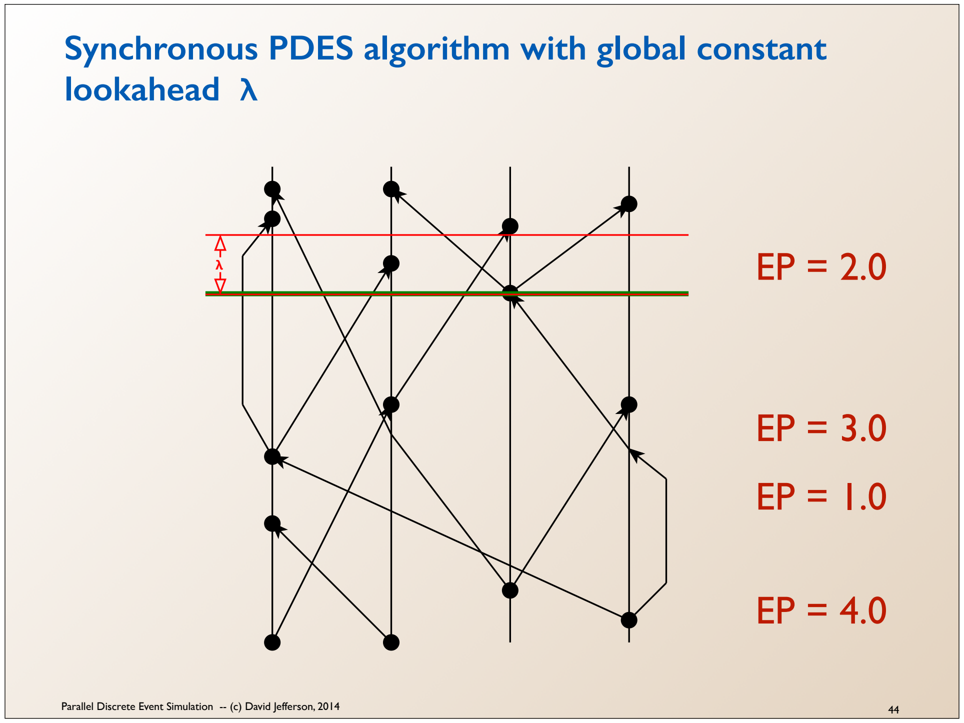
EP = 3.0

EP = 1.0

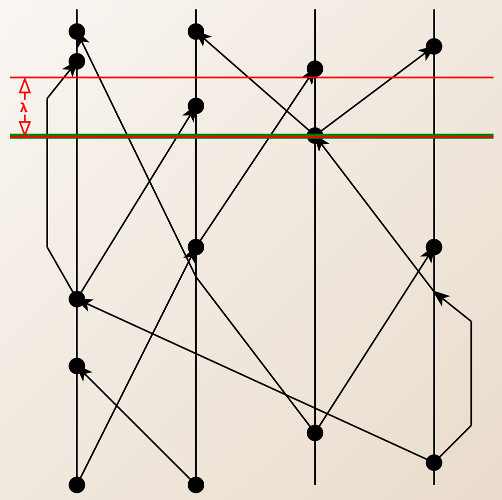
EP = 4.0

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

43



Synchronous PDES algorithm with global constant lookahead λ



EP = 2.0

EP = 3.0

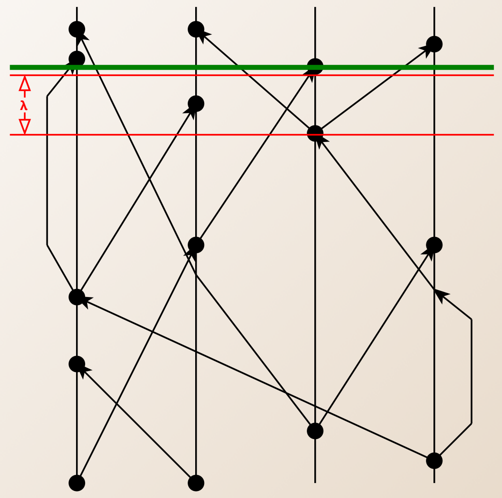
EP = 1.0

EP = 4.0

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

44

Synchronous PDES algorithm with global constant lookahead λ

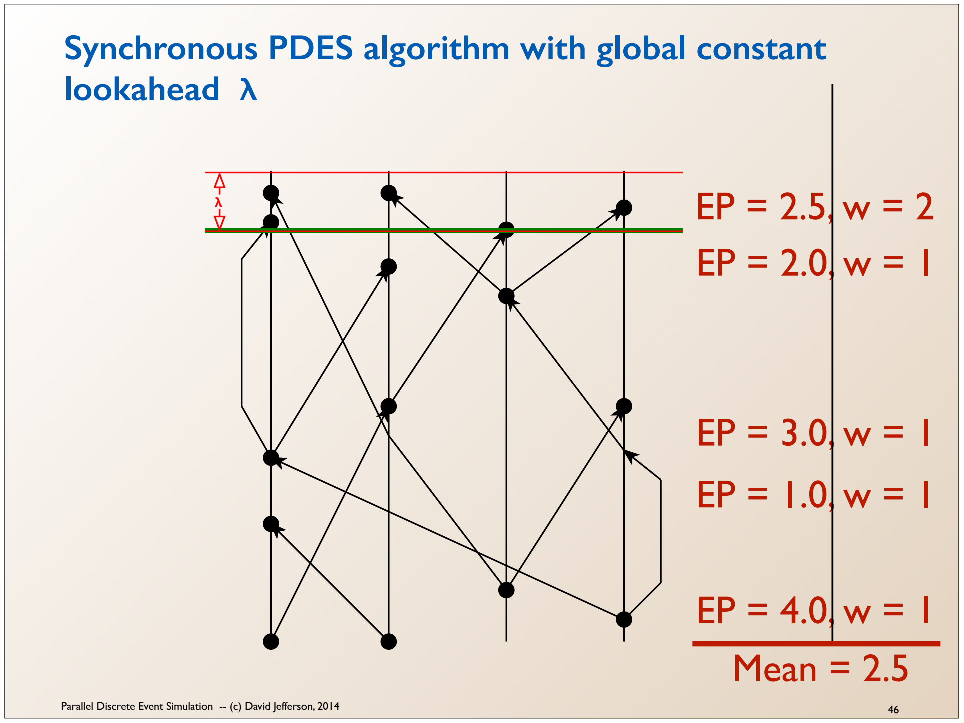


EP = 2.0

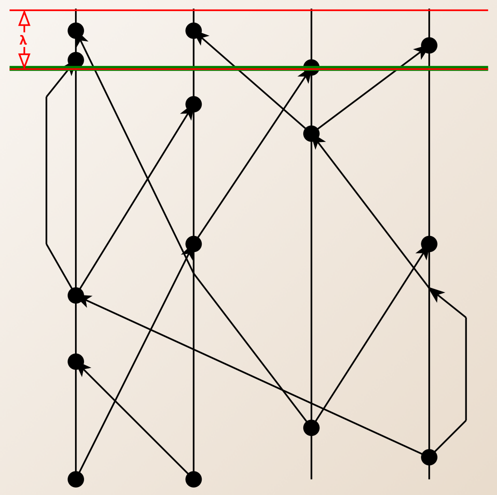
EP = 3.0

EP = 1.0

EP = 4.0



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

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

46

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!