Discrete Event Simulation

IN2045

Chapter 6 – Parallel Simulation

Dr. Alexander Klein
Stephan Günther
Prof. Dr.-Ing. Georg Carle

Chair for Network Architectures and Services
Department of Computer Science
Technische Universität München
http://www.net.in.tum.de

Some of today's slides/figures are borrowed from:
Richard Fujimoto

Parallel Simulation

- Motivation: Why to use parallel simulators and why it is difficult
- Conservative algorithms
  - Introduction: Only do what is allowed
  - Null messages: deadlock avoidance
  - Deadlock detection and recovery
  - Barrier synchronization and LBTS (lower bound on time stamp) calculation
- Optimistic algorithms
  - Time Warp algorithm: Do anything, roll back if necessary
  - Anti-Messages for rollback
  - Global Virtual Time (GVT) for fossil collection
  - State Saving Techniques
  - (Issues with Zero lookahead; Wide Virtual Time)
  - (Artificial Rollbacks to save memory)
- Summary
  - Pros and Cons of conservative and optimistic algorithms
  - Why not to use parallel simulators

Parallel Simulation: Summary/Outline

- Motivation: Why to use parallel simulators and why it is difficult
- Conservative algorithms
  - Introduction: Only do what is allowed
  - Null messages: deadlock avoidance
  - Deadlock detection and recovery
  - Barrier synchronization and LBTS (lower bound on time stamp) calculation
- Optimistic algorithms
  - Time Warp algorithm: Do anything, possibly roll back
  - Anti-Messages for rollback
  - Global Virtual Time (GVT) for fossil collection
  - State Saving Techniques
  - (Issues with Zero lookahead; Wide Virtual Time)
  - (Artificial Rollbacks to save memory)
- Summary
  - Pros and Cons of conservative and optimistic algorithms
  - Why not to use parallel simulators

Why should parallel simulation be used?

- Parallel computing is a general trend
  - Hardware power is increasingly improved by increasing the number of cores within one CPU
  - Hyper threading becomes more and more efficient
  - The clock cycle / computational power of single cores shows a slower increase

$$Speedup = \frac{Sequential\ \ execution\ \ time}{Parallel\ \ execution\ \ time}$$

Why do we still often use sequential simulation instead of parallel simulation?

Problem of Simulation Parallelisation

- Suppose that multiple processors pop and process events from The One Global Event List
- Problem:
  - CPU 1 picks and processes E1 at simulation time T1
  - CPU 2 picks and processes E2 at simulation time T2
  - E1 generates a new Event E3 at T3…
    - …but T3 < T2
    - and E3 changes the system state so that E2 now should show different effects
- Possible solutions:
  - Undo processing of E2: Wastes CPU time (and RAM)
  - Allow processing of E2 only when it is safe: Increases complexity and thus needs more CPU time (and RAM)

Parallel Simulation: Summary/Outline

- Motivation: Why to use parallel simulators and why it is difficult
- Conservative algorithms
  - Introduction: Only do what is allowed
  - Null messages: deadlock avoidance
  - Deadlock detection and recovery
  - Barrier synchronization and LBTS (lower bound on time stamp) calculation
- Optimistic algorithms
  - Time Warp algorithm: Do anything, possibly roll back
  - Anti-Messages for rollback
  - Global Virtual Time (GVT) for fossil collection
  - State Saving Techniques
  - (Issues with Zero lookahead; Wide Virtual Time)
  - (Artificial Rollbacks to save memory)
- Summary
  - Pros and Cons of conservative and optimistic algorithms
  - Why not to use parallel simulators

Event-Oriented Simulation: Non-parallel Example

Event handler procedures

state variables

Integer: InTheAir;
Integer: OnTheGround;
Boolean: RunwayFree;

```
Arrival Event
{
   …
}
```

```
Landed Event
{
   …
}
```

```
Departure Event
{
   …
}
```

Simulation application

Simulation executive

Event processing loop

Now = 8:45

Pending Event List (PEL)

9:00

9:16

10:10

```
While (simulation not finished)
   E = smallest time stamp event in PEL
   Remove E from PEL
   Now := time stamp of E
   call event handler procedure
END-LOOP
```

Parallel Discrete Event Simulation

- Extend example to model a network of airports
  - Encapsulate each airport in a logical process (LP)
  - Logical processes can schedule events (send messages) for other logical processes
- More generally...
  - Physical system
- Collection of interacting physical processes (airports)
- Simulation
  - Collection of logical processes (LPs)
  - Each LP models a physical process
- Interactions between physical processes modeled by scheduling events between LPs

LP Simulation Example

- Now: current simulation time
- InTheAir: number of aircraft landing or waiting to land
- OnTheGround: number of landed aircraft
- RunwayFree: Boolean, true if runway available

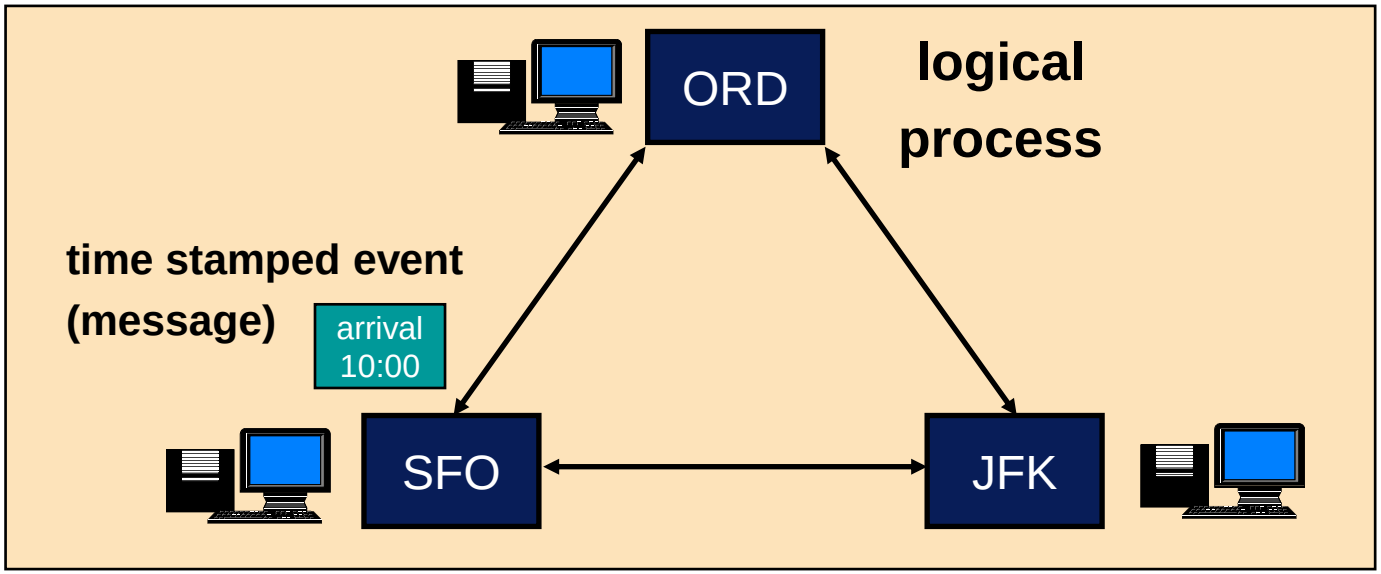
```
Arrival Event:
InTheAir := InTheAir+1;
If (RunwayFree)
   RunwayFree:=FALSE;
   Schedule Landed event (local) @ Now+R;
```

```
Landed Event:
InTheAir:=InTheAir-1;      OnTheGround:=OnTheGround+1;
Schedule Departure event (local) @ Now + G;
If (InTheAir>0) Schedule Landed event (local) @ Now+R;
Else RunwayFree := TRUE;
```

```
Departure Event (D = delay to reach another airport):
OnTheGround := OnTheGround - 1;
Schedule Arrival Event (remote) @ (Now+D) @ another airport
```

Approach to Parallel/Distributed Execution

- LP paradigm appears well suited to concurrent execution
- Map LPs to different processors
  - Might be different processors or even different computers
  - Multiple LPs per processor (optional)
- Communication via message passing
  - All interactions via messages
  - No shared state variables



Synchronization Problem and Local Causality

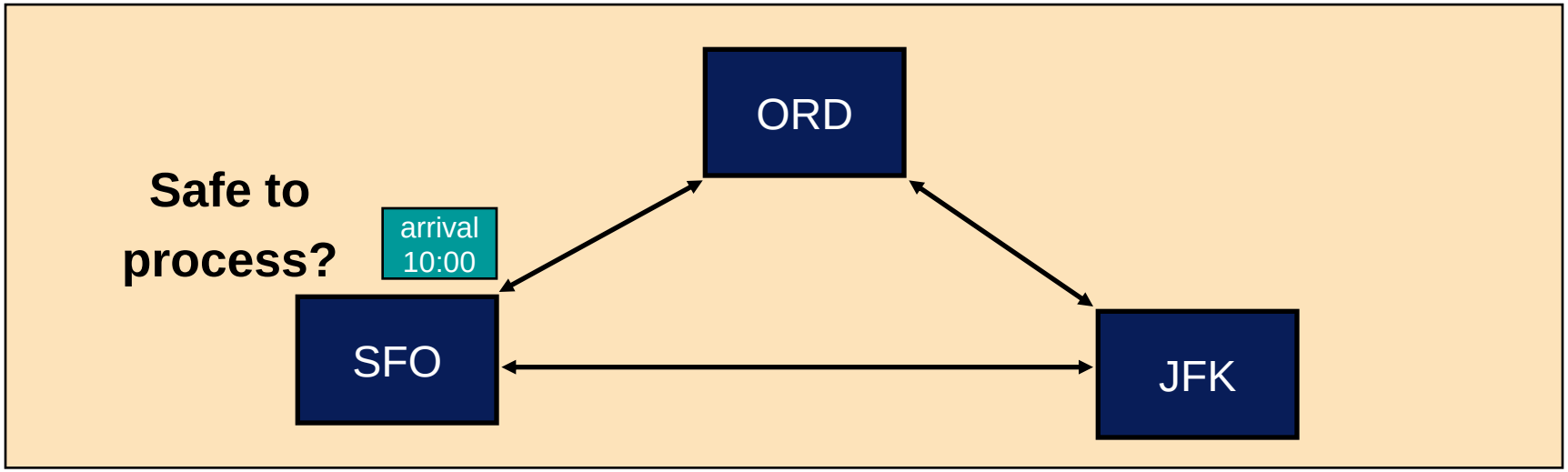
- Synchronization Problem:
  An algorithm is needed to ensure each LP processes events in time stamp order

- Local causality constraint:
  “Process incoming messages in time stamp order”
  (Golden rule for each process)

- Observation:
  Adherence to the local causality constraint is sufficient to ensure that the parallel simulation will produce exactly the same results as a sequential execution where all events across all LPs are processed in time stamp order [… if we ignore the case of events having the same time stamp] .

Local Causality

Golden rule for each process: “You shall process incoming messages in time stamp order” (local causality constraint)



Two Basic Synchronization Approaches

- Conservative synchronization: avoid violating the local causality constraint (wait until it is safe)
  - deadlock avoidance using null messages (Chandy/Misra/Bryant)
  - deadlock detection and recovery
  - synchronous algorithms (e.g., execute in “rounds”)

- Optimistic synchronization: allow violations of local causality to occur, but detect them at runtime and recover using a rollback mechanism
  - Time Warp (Jefferson)
  - numerous other approaches

Outline

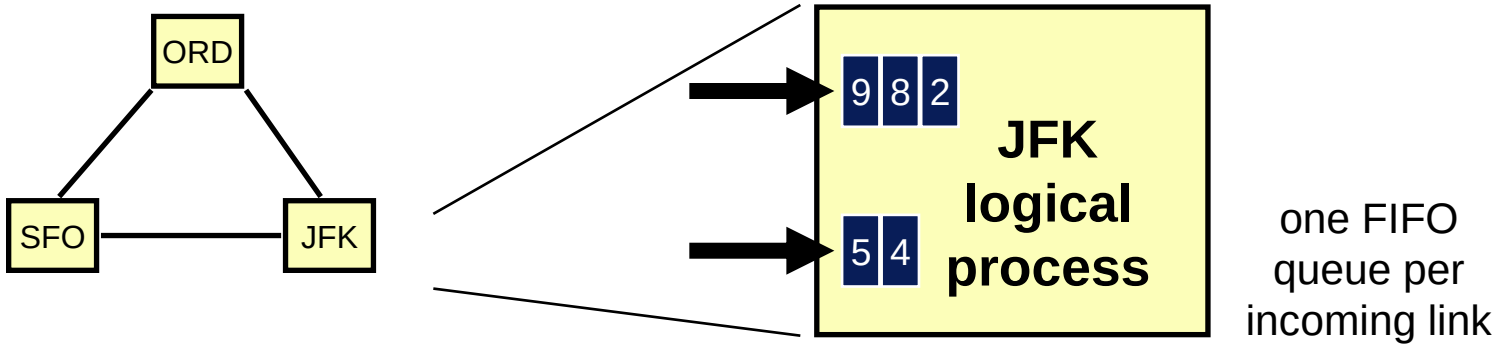
- Parallel / Distributed Computers
- Air Traffic Network Example
- Parallel Discrete Event Simulation
  - Logical processes
  - Local causality constraint
- Chandy/Misra/Bryant Null Message Algorithm
  - Ground rules
  - An algorithm that does not work
  - Deadlock avoidance using null messages

Chandy/Misra/Bryant “Null Message” Algorithm

Assumptions

- logical processes (LPs) exchanging time stamped events (messages)
- static network topology, no dynamic creation of LPs
- messages sent on each link are sent in time stamp order
- network provides reliable delivery, preserves order

Observation: The above assumptions imply the time stamp of the last message received on a link is a **lower bound on the time stamp (LBTS)** of subsequent messages received on that link (messages and thus events cannot overtake one another!)



Goal: Ensure LP processes events in time stamp order

Naive Approach:

Algorithm A (executed by each LP):

Goal: Ensure events are processed in time stamp order:

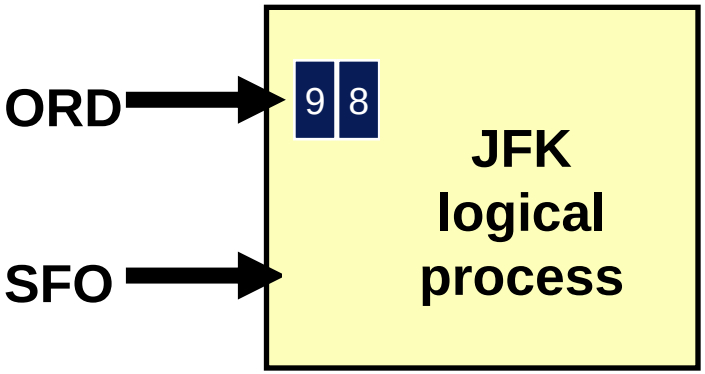
WHILE (simulation is not over)

wait until each FIFO contains at least one message

remove smallest time stamped event from its FIFO

process that event

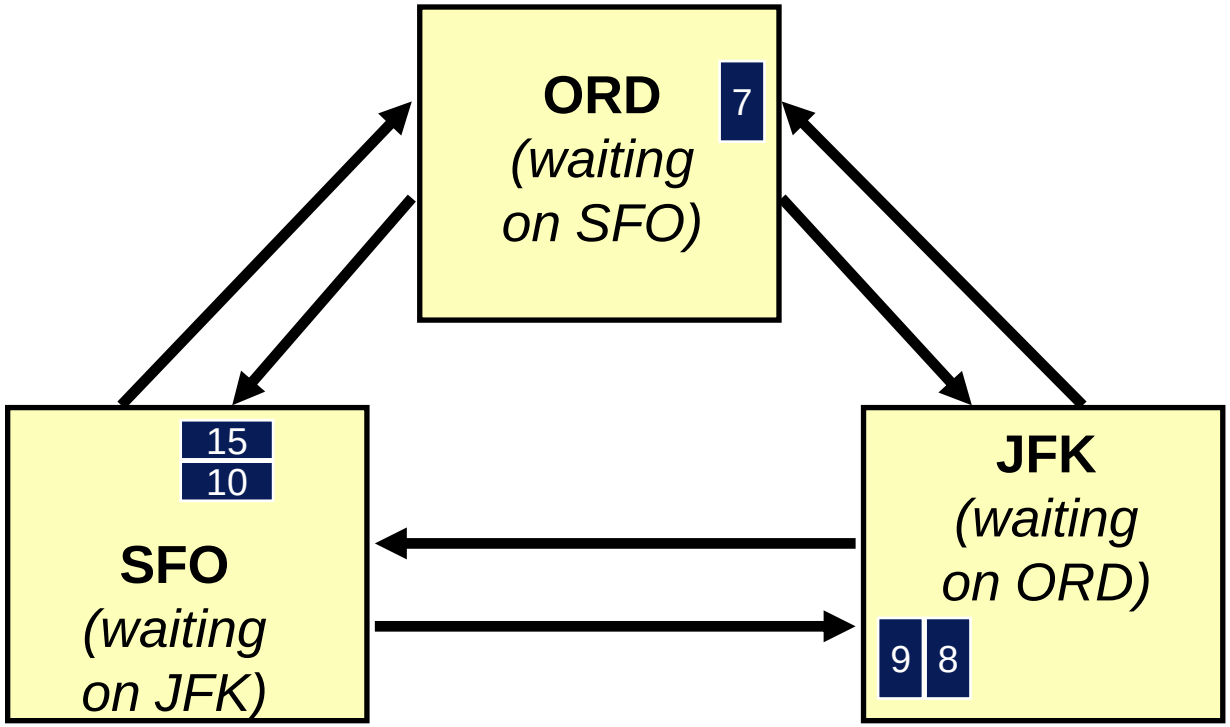
END-LOOP



- process time stamp 2 event
- process time stamp 4 event
- process time stamp 5 event
- wait until message is received from SFO

Observation: Algorithm A is prone to deadlock!

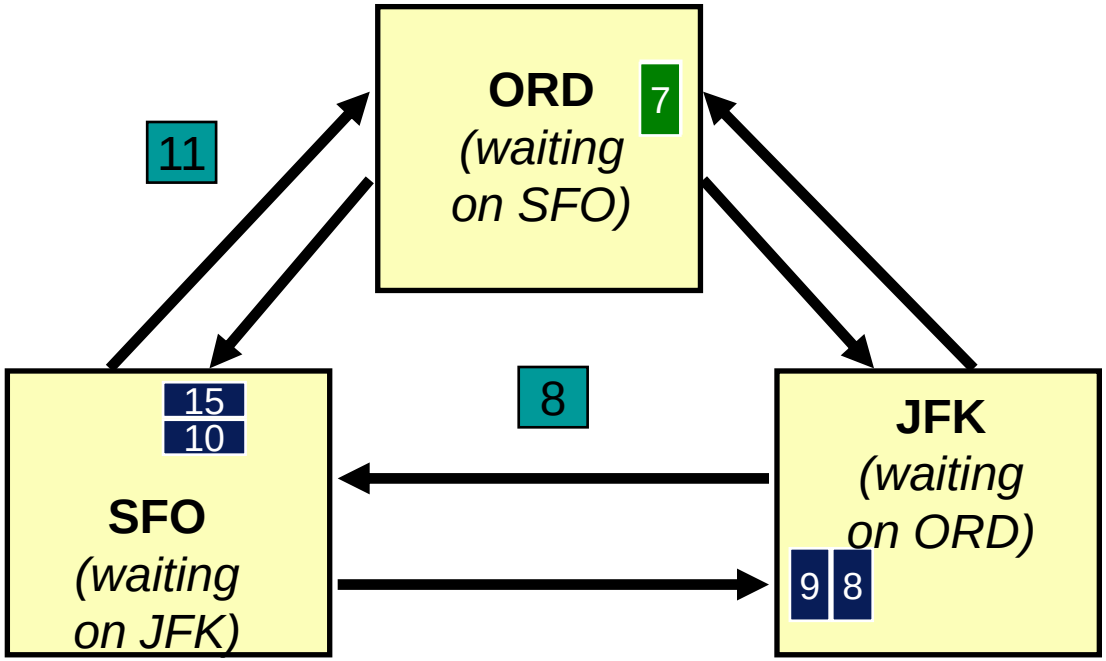
Deadlock Example



- A cycle of LPs forms where each is waiting on the next LP in the cycle.
- No LP can advance; the simulation is deadlocked.

Deadlock Avoidance Using Null Messages (Example)

Break deadlock: each LP sends “null” messages that indicate a lower bound on the time stamp of future messages.



Assume minimum delay between airports is 3 units of time

- JFK initially at time 5
- JFK sends null message to SFO with time stamp 8
- SFO sends null message to ORD with time stamp 11
- ORD may now process message with time stamp 7

Deadlock Avoidance Using Null Messages (algorithm)

Null Message Algorithm (executed by each LP):

Goal: Ensure events are processed in time stamp order and avoid deadlock

WHILE (simulation is not over)

wait until each FIFO contains at least one message

remove smallest time stamped event from its FIFO

process that event

send null messages to neighboring LPs with time stamp indicating a lower bound on future messages sent to that LP

(current time plus lookahead)

END-LOOP

The null message algorithm relies on a “lookahead” ability.

Deadlock Avoidance Using Null Messages

Null Message Algorithm (executed by each LP):

Goal: Ensure events are processed in time stamp order and avoid deadlock

WHILE (simulation is not over)

wait until each FIFO contains at least one message

remove smallest time stamped event from its FIFO

process that event

send null messages to neighboring LPs with time stamp indicating a lower bound on future messages sent to that LP

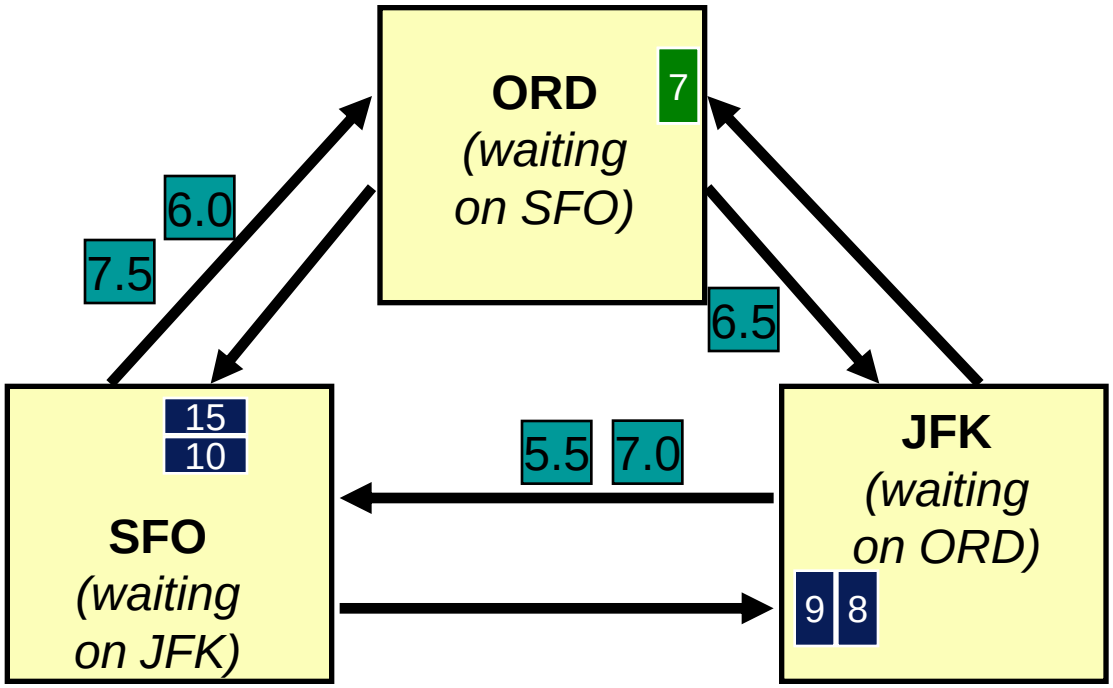
(current time plus minimum transit time between airports)

END-LOOP

Variation: LP *requests* null message when FIFO becomes empty

- Fewer null messages
- Delay to get time stamp information

The Time Creep Problem



Null messages:

JFK: timestamp = 5.5

SFO: timestamp = 6.0

ORD: timestamp = 6.5

JFK: timestamp = 7.0

SFO: timestamp = 7.5

ORD: process time stamp 7 message

Five "unnecessary" null messages to process a single event!

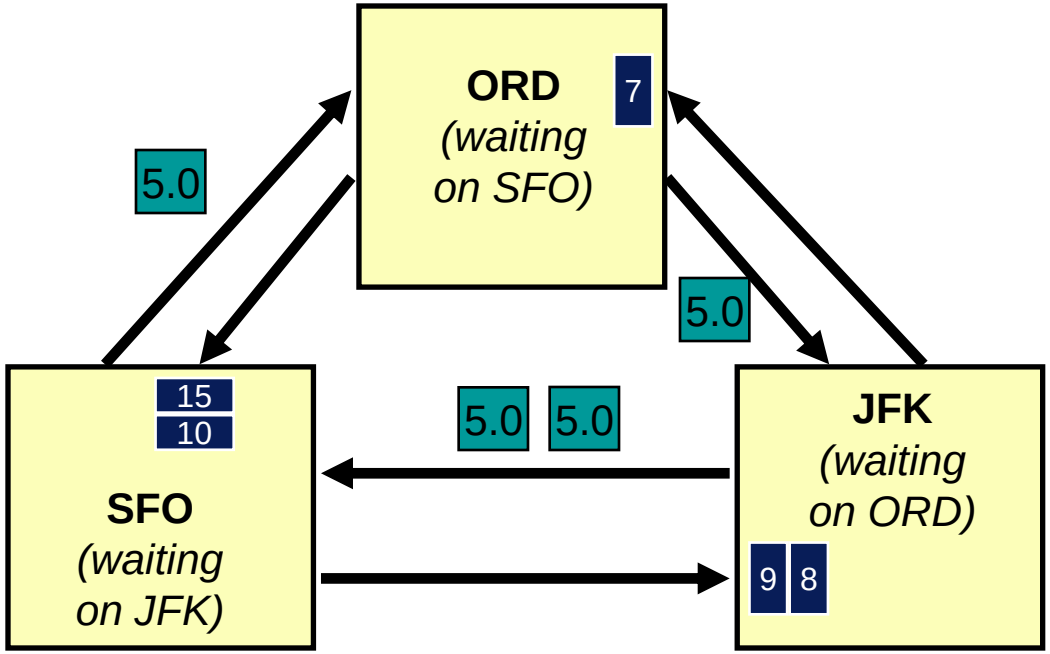
Assume minimum delay between airports is ~~3~~ 0.5 units of time

JFK initially at time 5

Many null messages if minimum flight time is small!

Worse yet: Livelock can occur!

Suppose the minimum delay between airports is zero…



Livelock: un-ending cycle of null messages where no LP can advance its simulation time

There cannot be a cycle where for each LP in the cycle, an incoming message with time stamp T results in a new message sent to the next LP in the cycle with time stamp T (zero lookahead cycle)

Outline

- Null message algorithm: The Time Creep Problem
- Lookahead
  - What is it and why is it important?
  - Writing simulations to maximize lookahead
- Changing lookahead
- Avoiding Time Creep

Lookahead

The null message algorithm relies on a "prediction" ability referred to as *lookahead*

- "ORD at simulation time 5, minimum transit time between airports is 3, so the next message sent by ORD must have a time stamp of *at least* 8"

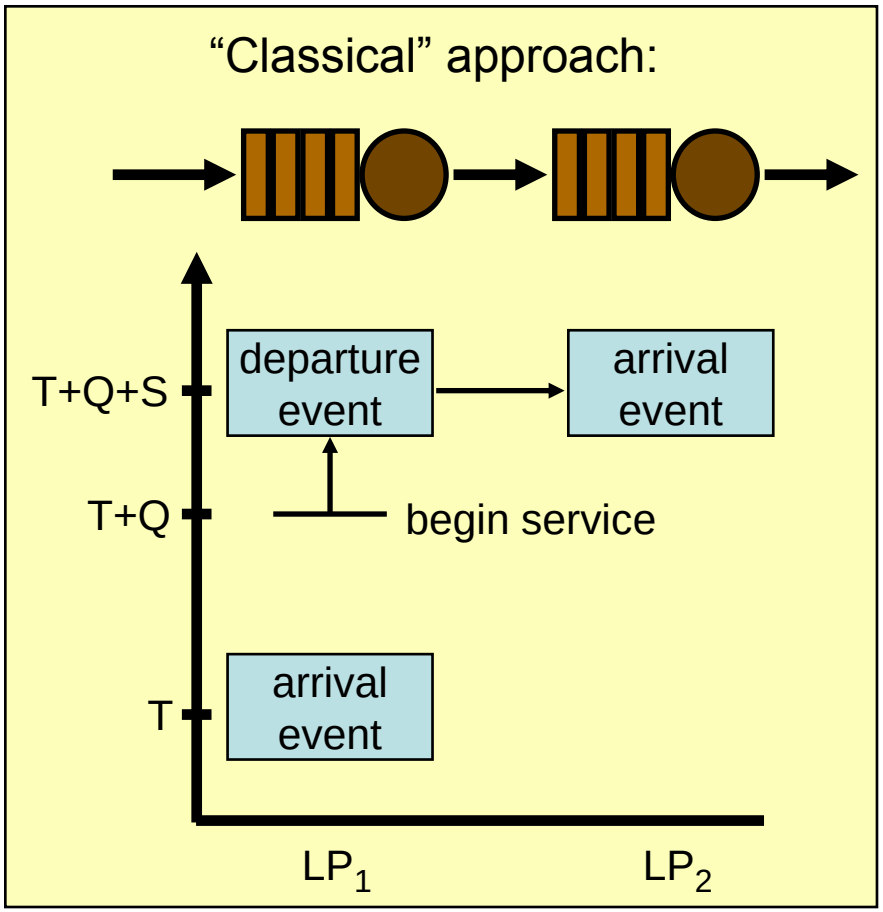
Lookahead is a constraint on LP's behaviour

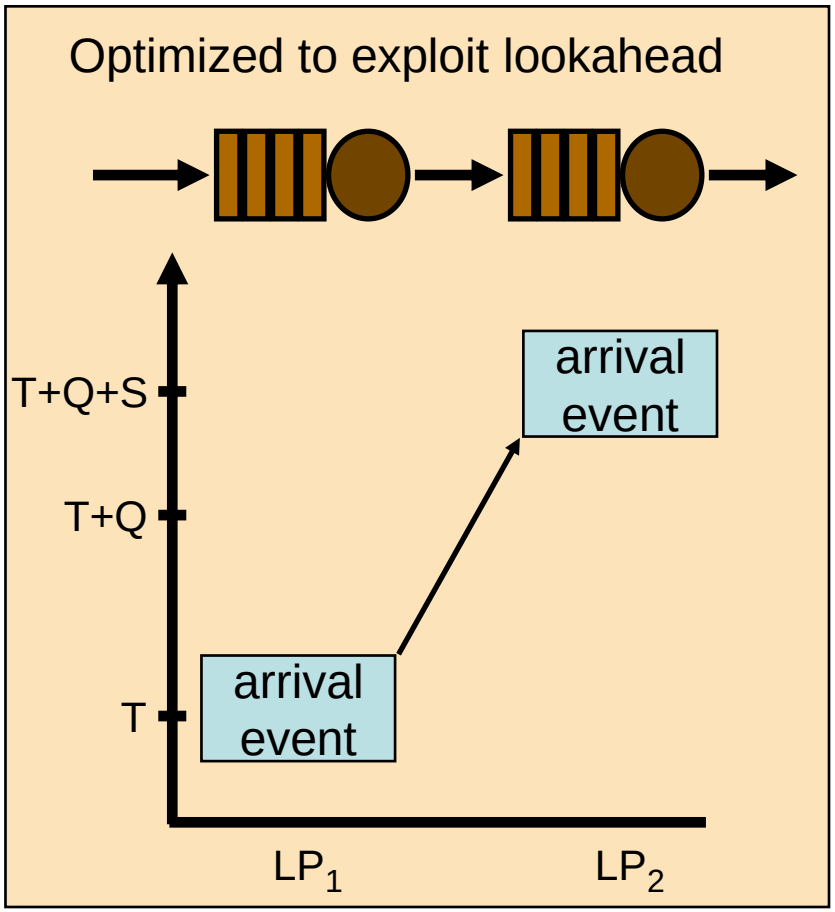
- **Link lookahead**: If an LP is at simulation time T, and an outgoing link has lookahead L_i, then any message sent on that link must have a time stamp of at least $T+L_i$
- **LP Lookahead**: If an LP is at simulation time T, and has a lookahead of L, then any message sent by that LP must have a time stamp of at least T+L
  - Equivalent to link lookahead where the lookahead on each outgoing link is the same

Exploiting Lookahead in Applications

Example: Tandem first-come-first-serve queues

T = arrival time of job
Q = waiting time in queue
S = service time



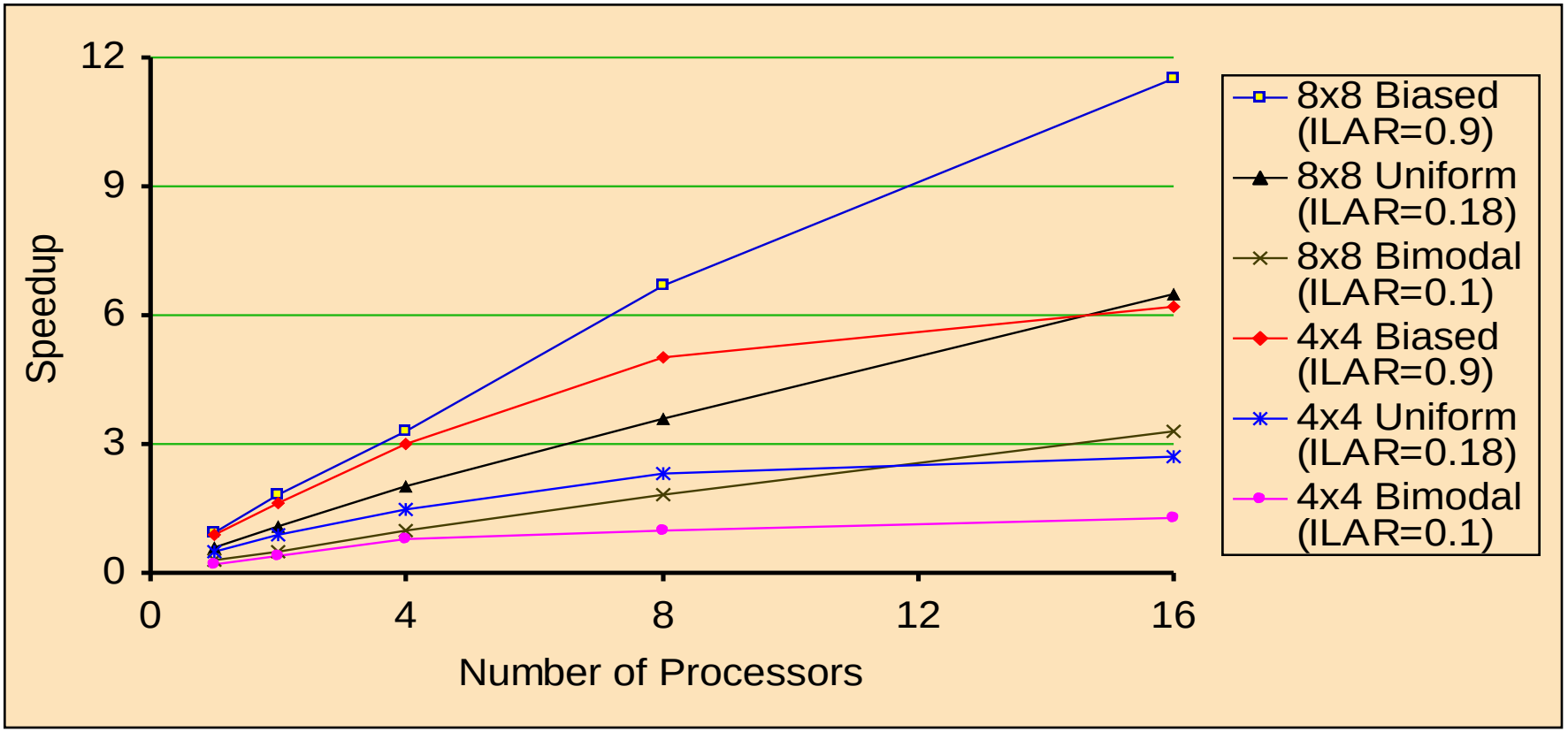


The degree to which the program can exploit lookahead is critical for good performance

Null Message Algorithm: Speedup depends on various factors

- toroid topology
- message density: 4 per LP
- 1 millisecond computation per event

- vary time stamp increment distribution
- ILAR=lookahead / average time stamp increment



Conservative algorithms live or die by their lookahead!

Lookahead and the Simulation Model

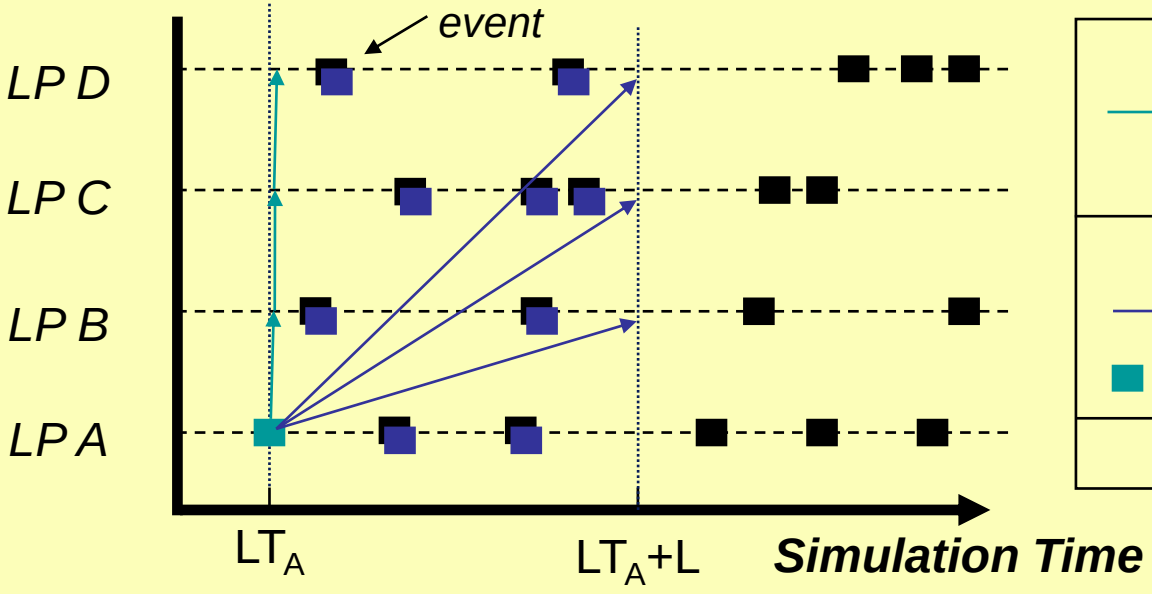
Lookahead is clearly dependent on the simulation model:

- Could be derived from physical constraints in the system being modeled, such as minimum simulation time for one entity to affect another (e.g., a weapon fired from a tank requires L units of time to reach another tank, or maximum speed of the tank places lower bound on how soon it can affect another entity)
- Could be derived from characteristics of the simulation entities, such as *non-preemptable* behavior (e.g., a tank is traveling north at 30 mph, and nothing in the federation model can cause its behavior to change over the next 10 minutes, so all output from the tank simulator can be generated immediately up to time “local clock + 10 minutes”)
- Could be derived from tolerance to temporal inaccuracies (e.g., users cannot perceive temporal difference of 100 milliseconds, so messages may be timestamped 100 milliseconds into the future).
- Simulations may be able to *precompute* when its next interaction with another simulation will be (e.g., if time until next interaction is stochastic, pre-sample random number generator to determine time of next interaction).

Observation: time-stepped simulations implicitly use lookahead; events in current time step are considered independent (and can be processed concurrently), new events are generated for the next time step, or later.

Why Lookahead is important

problem: limited concurrency
each LP must process events in time stamp order



without lookahead
possible message
OK to process

with lookahead
possible message
OK to process

not OK to process yet

Each LP A using logical time declares a lookahead value L; the time stamp of any event generated by the LP must be ≥ LT_A + L

- Lookahead is used in virtually all conservative synchronization protocols
- Essential to allow concurrent processing of events

Lookahead is necessary to allow concurrent processing of events with different time stamps (unless optimistic event processing is used)

Outline

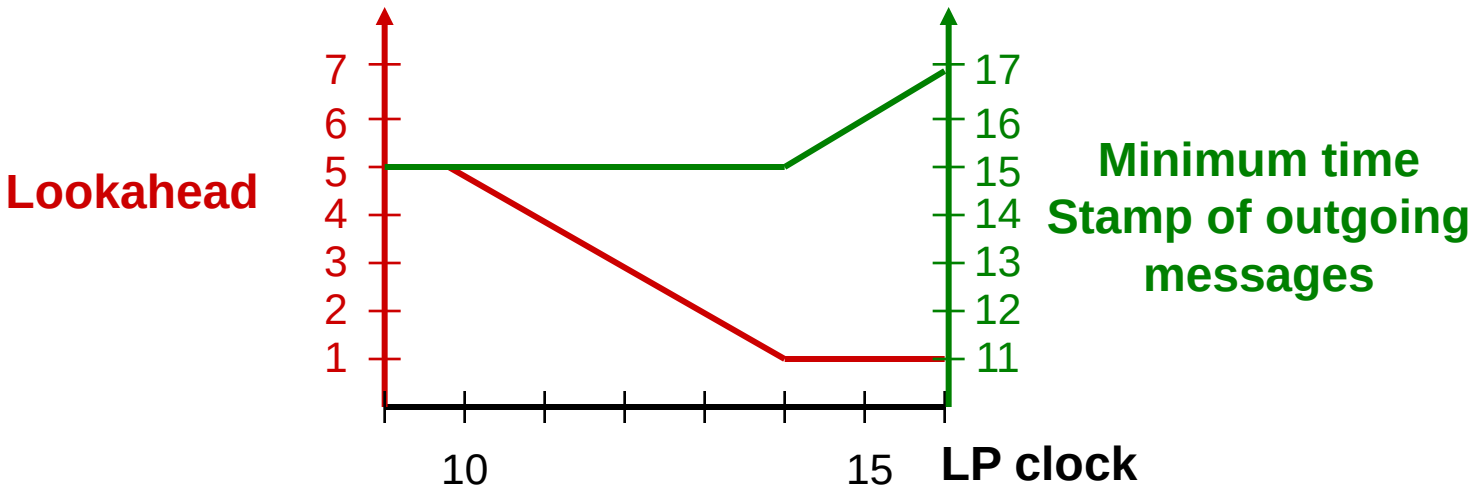
- Null message algorithm: The Time Creep Problem
- Lookahead
  - What is it and why is it important?
  - Writing simulations to maximize lookahead
- Changing lookahead
- Avoiding Time Creep

Changing Lookahead Values

- Increasing lookahead
  - No problem; lookahead can immediately be changed
- Decreasing lookahead
  - Previous time stamp guarantees must be honored!
  - Lookahead thus cannot immediately be decreased
    - If an LP is at simulation time 10 and lookahead is 5, it has promised subsequent messages will have a time stamp of at least 15
    - If lookahead were *immediately* set to 1, it could generate a message with time stamp 11
  - Lookahead can decrease by k units of simulation time only after the LP has advanced k units of simulation time

Example: Decreasing Lookahead

- SFO: simulation time = 10, lookahead = 5
- Future messages sent on link must have time stamp ≥ 15
- SFO: request its lookahead be reduced to 1



Summary

- Null message algorithm
  - Lookahead creep problem
  - No zero lookahead cycles allowed
- Lookahead
  - Constraint on time stamps of subsequent messages
  - Has large effect on performance: essential for concurrent processing of events for conservative algorithms
  - Programs/models must be coded to exploit lookahead!
- Use time of next event to avoid lookahead creep

Parallel Simulation: Summary/Outline

- Motivation: Why to use parallel simulators and why it's difficult
- Conservative algorithms: Only do what is allowed
  - Introduction
  - Null messages: deadlock avoidance
  - Deadlock detection and recovery
  - Barrier synchronization and LBTS (lower bound on time stamp) calculation
- Optimistic algorithms: Do anything, possibly roll back
  - Time Warp algorithm
  - Anti-Messages for rollback
  - Global Virtual Time (GVT) for fossil collection
  - State Saving Techniques
  - (Issues with Zero lookahead; Wide Virtual Time)
  - (Artificial Rollbacks to save memory)
- Summary
  - Pros and Cons of conservative and optimistic algorithms
  - Why not to use parallel simulators

Outline

- Deadlock Detection and Recovery Algorithm (Chandy and Misra)
  - Basic Approach
  - Deadlock Detection
    - Diffusing distributed computations
    - Dijkstra/Scholten algorithm (signaling protocol)
  - Deadlock Recovery

Deadlock Detection & Recovery: Idea

Algorithm A (executed by each LP):

Goal: Ensure events are processed in time stamp order:

```
WHILE (simulation is not over)
   wait until each FIFO contains at least one
   message
   remove smallest time stamped event from its FIFO
   process that event
END-LOOP
```

- But: No null messages!
- Allow simulation to execute until deadlock occurs
- Provide a mechanism to detect deadlock
- Provide a mechanism to recover from deadlocks

Deadlock Detection

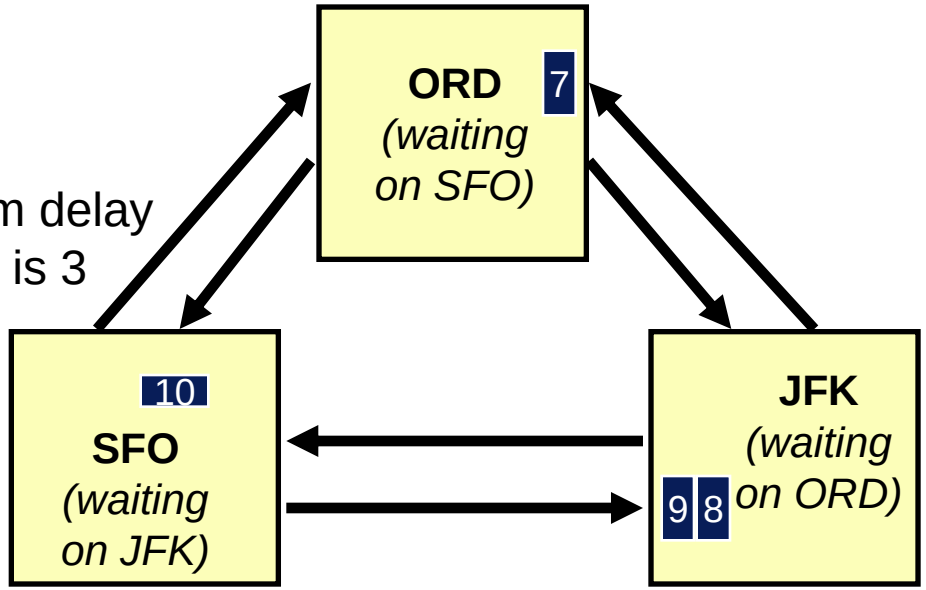
- There are algorithms that can detect when the entire system hangs (i.e., no LP can go on)
  - Even distributed algorithms exist
  - For example, Dijkstra/Schoten algorithm (not shown)
  - Of course, detection incurs additional processing overhead
- Once a deadlock has been detected, we resolve it (recovery)
  - Do not confuse this with optimistic simulation –
    we will not roll back any actions!

Deadlock Recovery

Deadlock recovery: identify “safe” events (events that can be processed without violating local causality)

deadlock state

Assume minimum delay between airports is 3



Which events are safe?

- Time stamp 7: smallest time stamped event in system
- Time stamp 8, 9: safe because of lookahead constraint
- Time stamp 10: OK if events with the same time stamp can be processed in any order
- No time creep!

Summary

- Deadlock Detection
  - Diffusing computation: Dijkstra/Scholten algorithm
  - Simple signaling protocol detects deadlock
  - Does not detect partial (local) deadlocks
- Deadlock Recovery
  - Smallest time stamp event safe to process
  - Others may also be safe (requires additional work to determine this)

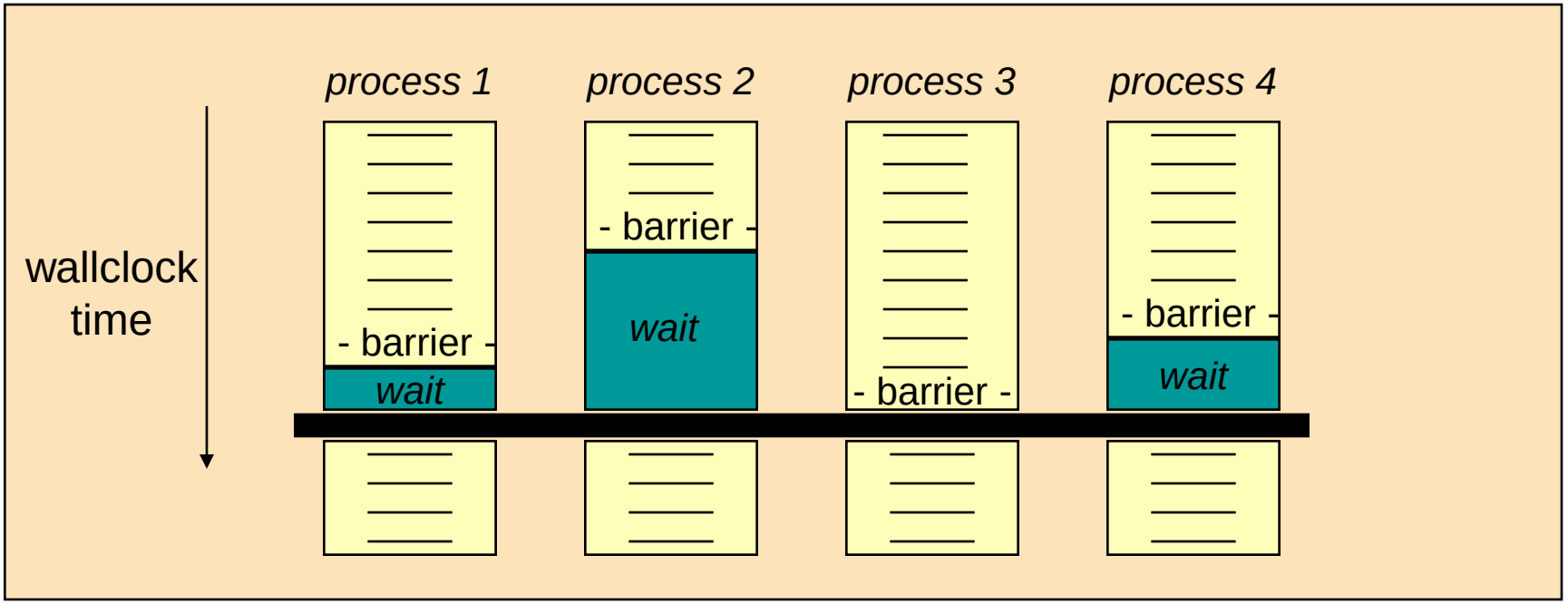
Parallel Simulation: Summary/Outline

- Motivation: Why to use parallel simulators and why it's difficult
- Conservative algorithms: Only do what is allowed
  - Introduction
  - Null messages: deadlock avoidance
  - Deadlock detection and recovery
  - Barrier synchronization and LBTS (lower bound on time stamp) calculation
- Optimistic algorithms: Do anything, possibly roll back
  - Time Warp algorithm
  - Anti-Messages for rollback
  - Global Virtual Time (GVT) for fossil collection
  - State Saving Techniques
  - (Issues with Zero lookahead; Wide Virtual Time)
  - (Artificial Rollbacks to save memory)
- Summary
  - Pros and Cons of conservative and optimistic algorithms
  - Why not to use parallel simulators

Outline

- Barrier synchronizations and a simple synchronous algorithm
- Implementation of Barrier mechanisms
  - Centralized Barriers
  - Tree Barrier
  - Butterfly Barrier
- Computing LBTS (lower bound on timestamp)

Barrier Synchronization



- Barrier Synchronization: When a process invokes the barrier primitive, it will block until all other processes have also invoked the barrier primitive.
- When the last process invokes the barrier, all processes can resume execution

Synchronous Execution

- Recall the goal is to ensure that each LP processes the events in timestamp order
- Basic idea:
  Each process cycles through the following steps:
  - Determine the events that are safe to process
    - Compute a Lower Bound on the Time Stamp ($LBTS_i$) of events that LP_i might later receive
    - Events with time stamp ≤ LBTS are safe to process
  - Process safe events, exchange messages
  - Global synchronization (barrier)

- Messages generated in one cycle are not eligible for processing until the next cycle

A Simple Synchronous Algorithm

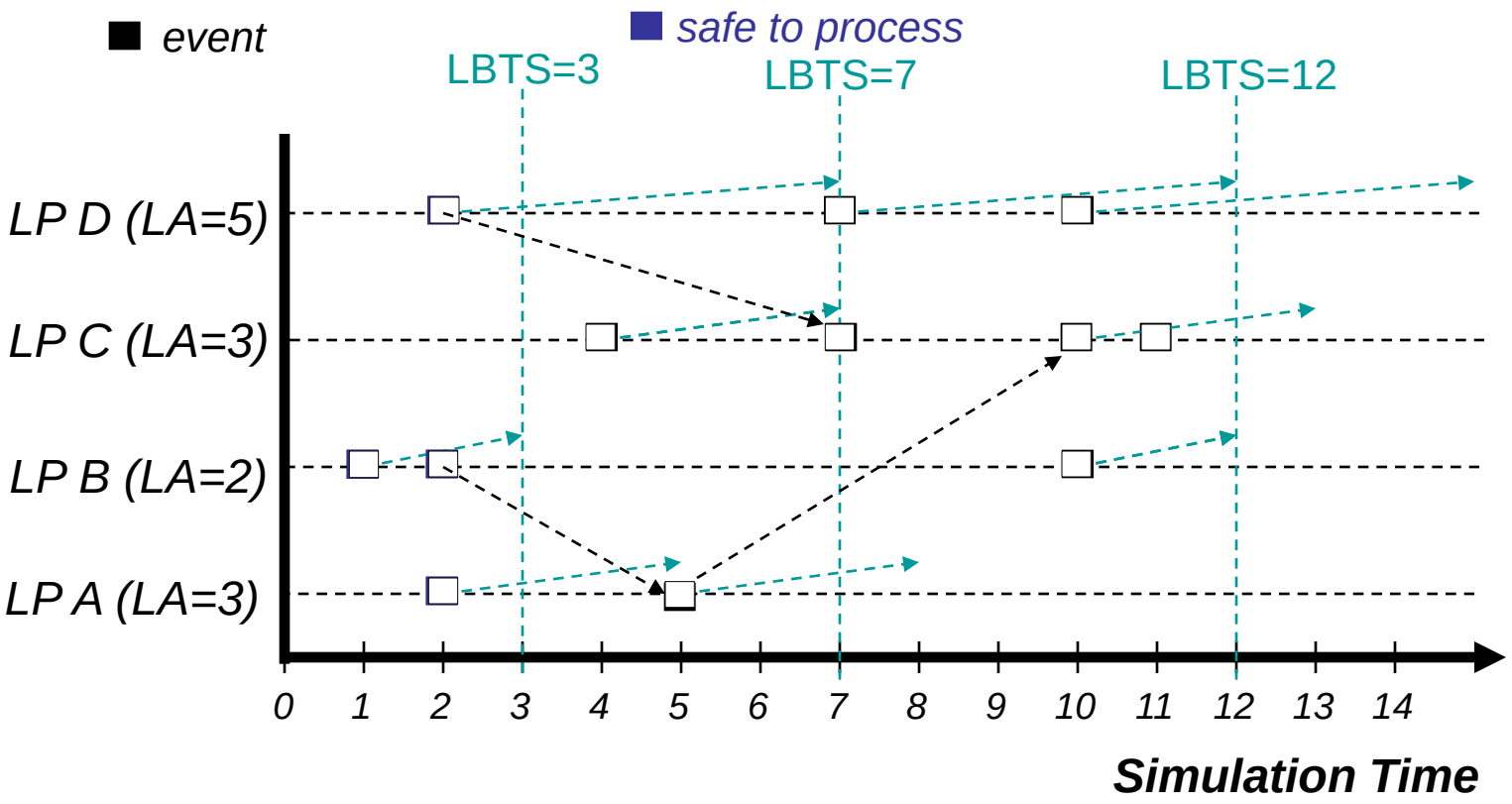
- Assume any LP can communicate with any other LP
- Assume instantaneous message transmission (revisit later)
- N_i = time of next event in LP_i
- LA_i = lookahead of LP_i

```
WHILE (unprocessed events remain)
   receive messages generated in previous iteration
   LBTS = min (Ni + LAi)
   process events with time stamp ≤ LBTS
  barrier synchronization
END
```

$LBTS_i$ = a lower bound on the time stamp of the messages that LP_i might receive in the future

If $LBTS_i$ is the same for all LPs, then it is simply called LBTS

Synchronous Execution Example



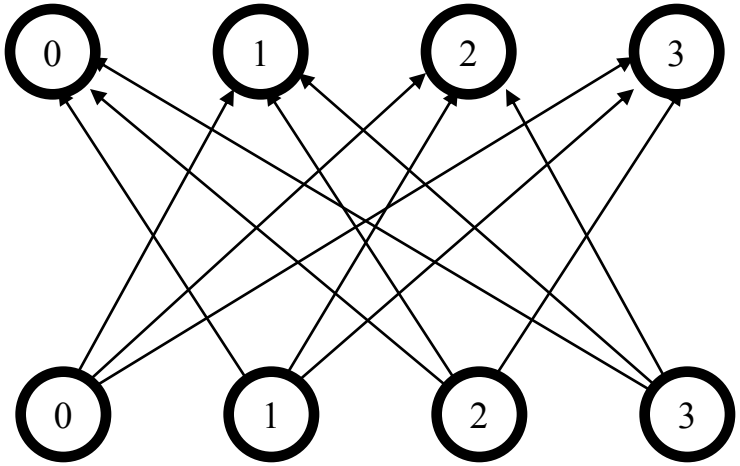
Issues

- Implementing the barrier mechanism
  - Centralized
  - Broadcast
  - Trees
  - Butterfly and other more sophisticated infrastructures (not shown)
- Computing LBTS (global minimum)
- Transient messages (i.e., messaging is not instantaneous – will be discussed later)

Method 1: Barrier Using a Centralized Controller

- Central controller process used to implement barrier
- Overall, a two step process
  - Controller determines when barrier reached
  - Broadcast message to release processes from the barrier
- Barrier primitive for non-controller processes:
  - Send a message to central controller
  - Wait for a reply
- Barrier primitive for controller process
  - Receive barrier messages from other processes
  - When a message is received from each process, broadcast message to release barrier
- Performance
  - Controller must send and receive N-1 messages
  - Controller is…
    - … either potential bottleneck
    - … or idles around most of the time, waiting for barrier messages

Method 2: Broadcast Barrier

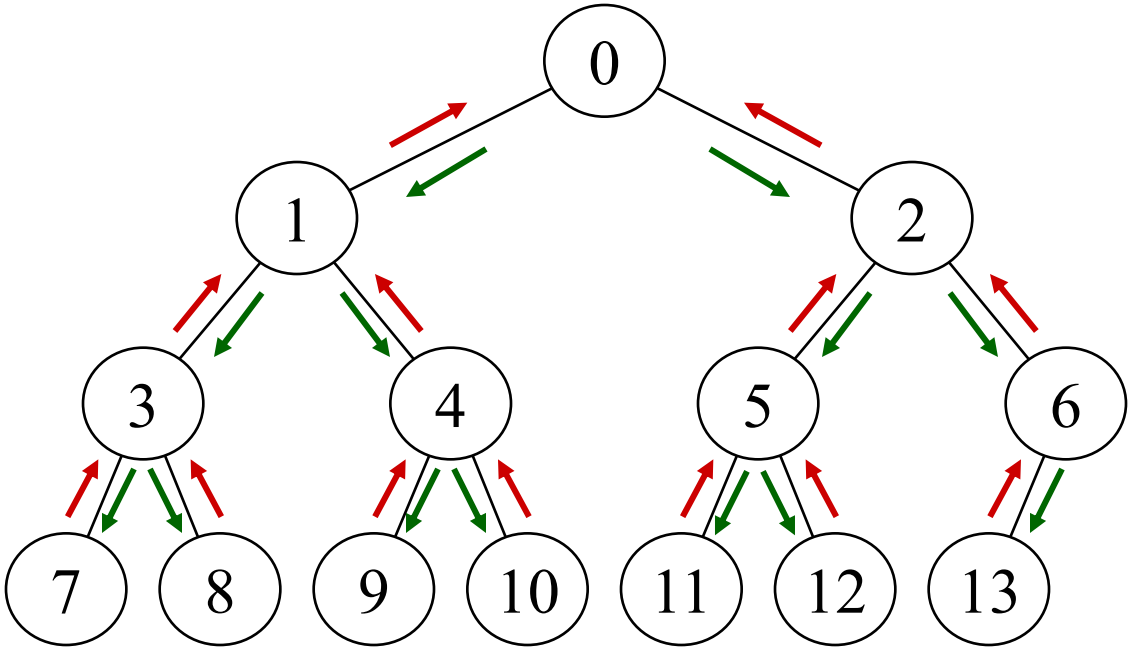


One step approach; no central controller

Each process:

- Broadcast message when barrier primitive is invoked
- Wait until a message is received from each other process
- N·(N-1) messages – that's quadratic!

Method 3: Tree Barrier

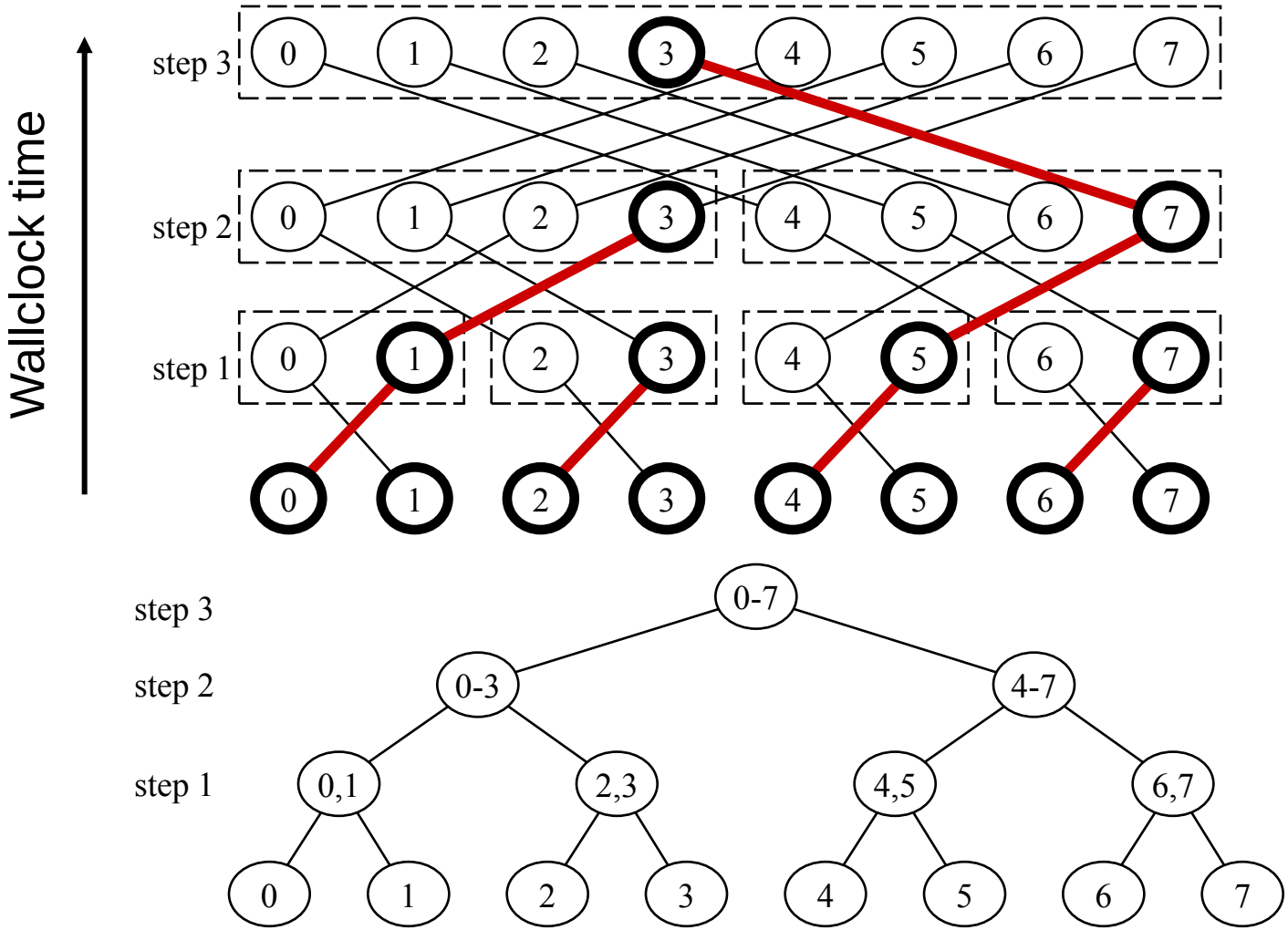


- Organize processes into a tree
- A process sends a message to its parent process when
  - The process has reached the barrier point, and
  - A message has been received from each of its children processes
- Root detects completion of barrier, broadcast message to release processes (e.g., send messages down tree)
- 2 log N time if all processes reach barrier at same time

Method 4: Butterfly Barrier

- Will touch this only briefly…
- N processes (here, assume N is a power of 2)
- Sequence of $\log_2 N$ pairwise barriers (let $k = \log_2 N$)
- Pairwise barrier:
  - Send message to partner process
  - Wait until message is received from that process
- Process p: $b_k b_{k-1} \ldots b_1$ = binary representation of p
- Step i: perform barrier with process $b_k \ldots b_i' \ldots b_1$ (complement i-th bit of the binary representation)
- Example: Process 3 (011)
  - Step 1: pairwise barrier with process 2 (010)
  - Step 2: pairwise barrier with process 1 (001)
  - Step 3: pairwise barrier with process 7 (111)

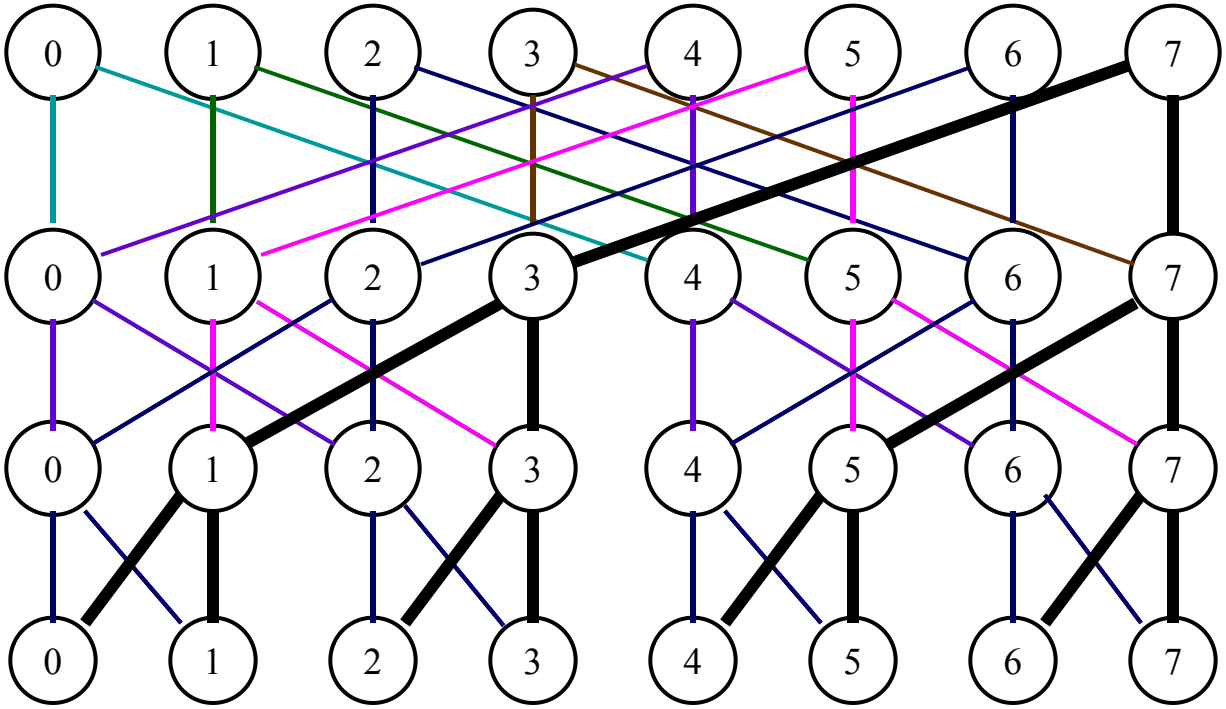
Method 4: Butterfly Barrier (won't go into details)



- The communication pattern forms a tree from the perspective of *any* process

Butterfly: Superimposed Trees

- An N node butterfly can be viewed as N trees superimposed over each other



- After $\log_2$ N steps each process is notified that the barrier operation has completed

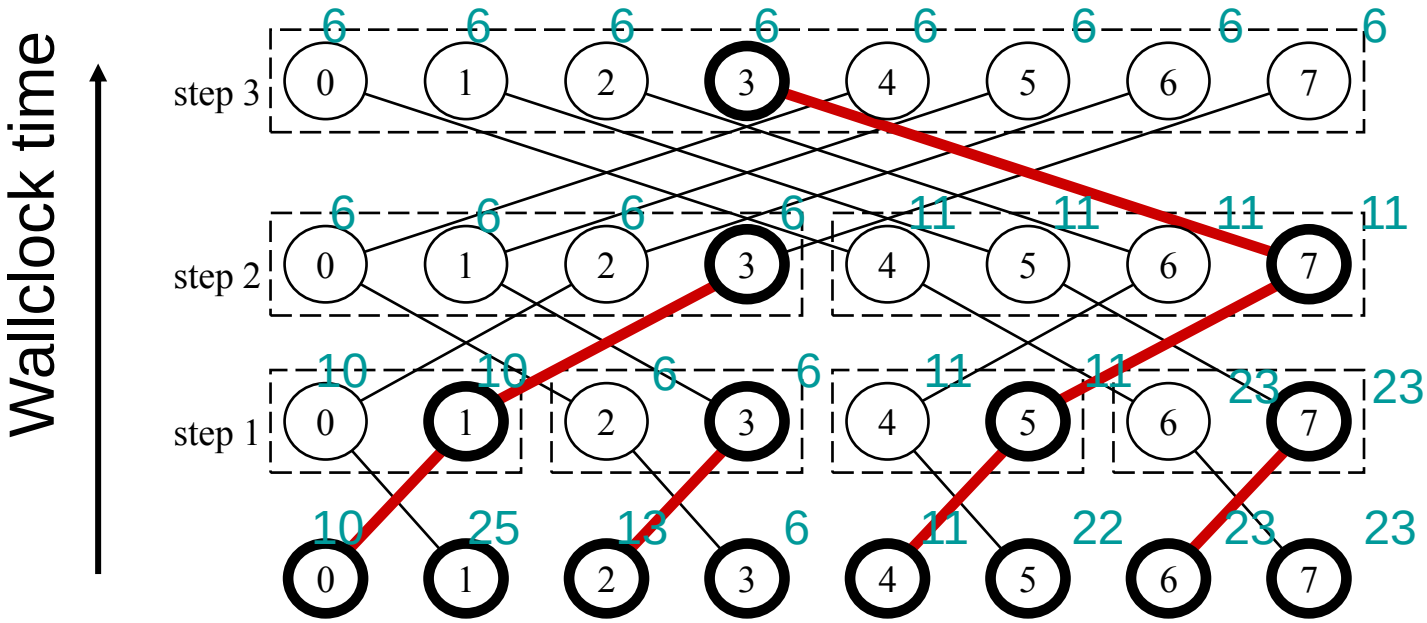
Outline

- Barrier synchronizations and a simple synchronous algorithm
- Implementation of Barrier mechanisms
  - Centralized Barriers
  - Tree Barrier
  - (Butterfly Barrier – not shown)
- Computing LBTS

Computing LBTS

It is trivial to extend any of these barrier algorithms to also compute a global minimum (LBTS)

- Piggyback local time value on each barrier message
- Compute new minimum among { local value, incoming message(s) }
- Transmit new minimum in next step



- After log N steps, (1) LBTS has been computed, (2) each process has the LBTS value

Summary

- Synchronous algorithms use a global barrier to ensure events are processed in time stamp order
  - Requires computation of a Lower Bound on Time Stamp (LBTS) on messages each LP might later receive
- There are several ways to implement barriers
  - Central controller
  - Broadcast
  - Tree
  - Butterfly (touched only briefly)
- The LBTS computation can be “piggybacked” onto the barrier synchronization algorithm

Outline

- Transient Messages
  - Transient Message Problem
  - Flush Barrier
  - Tree Implementation
  - Butterfly Implementation
- Distance Between Processes
  - Potential Performance Improvement
  - Distance Matrix

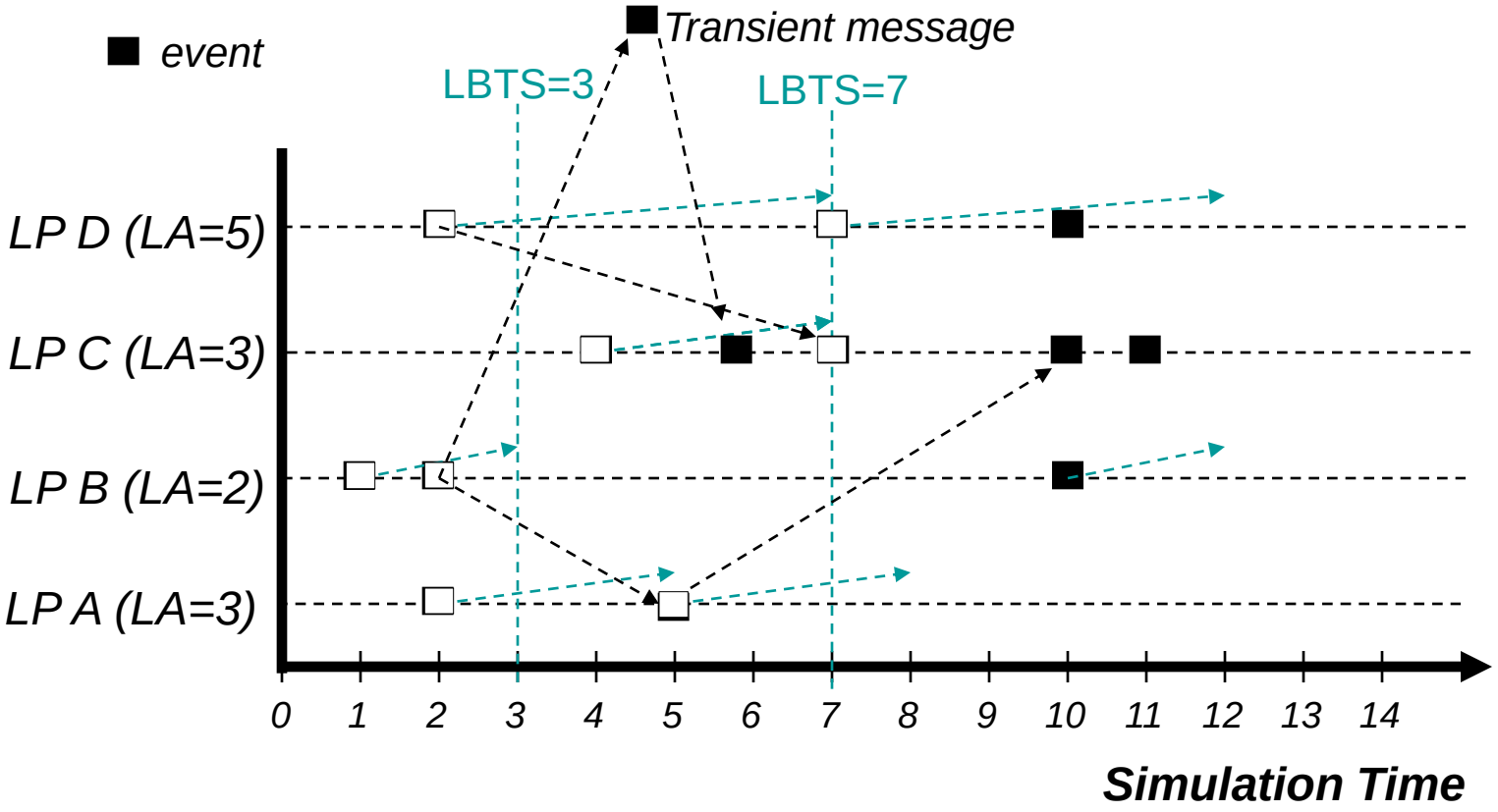
The Transient Message Problem

```
/* synchronous algorithm */

N_i = time of next event in LP_i
LA_i = lookahead of LP_i
WHILE (unprocessed events remain)
       receive messages generated in previous iteration
       LBTS = min (N_i + LA_i)
       process events in with time stamp ≤ LBTS
       barrier synchronization
ENDWHILE
```

- A transient message is a message that has been sent, but has not yet been received at its destination
- The message could be “in the network” or stored in an operating system buffer (waiting to be sent or delivered)
- The synchronous algorithm fails if there are transient message(s) remaining after the processes have been released from the barrier!

Transient Message Example



Message arrives in C's past!

Flush Barrier

No process will be released from the barrier until

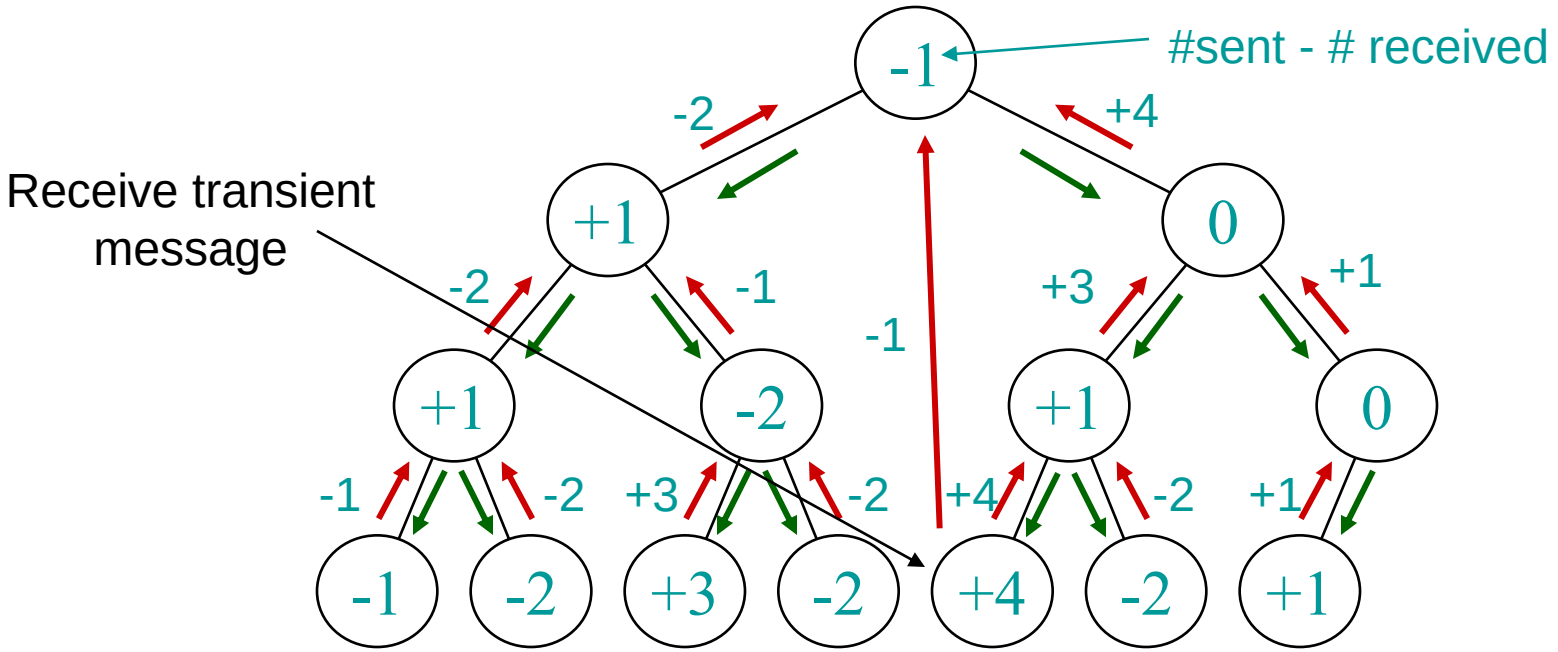
- All processes have reached the barrier
- Any message sent by a process before reaching the barrier has arrived at its destination

```
Revised algorithm:
WHILE (unprocessed events remain)
       receive messages generated in previous iteration
       LBTS = min (N_i + LA_i)
       process events in with time stamp ≤ LBTS
  flush barrier
END
```

Implementation

- Use FIFO communication channels
- Approach #1: Send a “dummy message” on each channel; wait until such a message is received on each incoming channel to guarantee transient messages have been received
  - May require a large number of messages
  - Again: Overhead…
- Approach #2: Message counters
  - $Send_i$ = number of messages sent by LP_i (this iteration)
  - Rec_i = number of messages received by LP_i (this iteration)
  - There are no transient messages when
    - All processes are blocked (i.e., at the barrier), and
    - $\sum Send_i = \sum Rec_i$
  - Again: Overhead…

Tree: Flush Barrier



- When a leaf process reaches flush barrier, include counter (#sent - #received) in messages sent to parent
- Parent adds counters in incoming messages with its own counter, sends sum in message sent to its parent
- If sum at root is zero, broadcast “go” message, else wait until sum is equal to zero
- Receive message after reporting sum: send update message to root

Outline

- Transient Messages
  - Transient Message Problem
  - Flush Barrier
  - Tree Implementation
  - Butterfly Implementation
- Distance Between Processes
  - Potential Performance Improvement
  - Distance Matrix

Identifying Safe Events

```
WHILE (unprocessed events remain)
   receive messages generated in previous iteration
   LBTS = min (N_i + LA_i)
       /* time of next event + lookahead */
   process events in with time stamp ≤ LBTS*flush barrier
      /* barrier + eliminate all transient messages */
END
```

- If all processes are blocked and there are no transient messages in the system, LBTS = min (N_i + LA_i) for each process where N_i and LA_i are the time of the next unprocessed event and lookahead, respectively, for LP_i
- Overly conservative estimate for LBTS
- Does not exploit “locality” in physical systems (things far away cannot affect you for some time into the future)

Improving the LBTS estimate

- We may calculate an overly conservative estimate for LBTS
  - Does not exploit “locality” in physical systems:
    Things far away can’t affect you for some time into the future
- Possible optimization: Exploit distance between LPs
  - Exploit locality in physical systems to improve concurrency in the simulation execution
  - Increased complexity, overhead
  - Lookahead and topology changes introduce additional complexities

Summary

- Transient messages must be accounted for by the synchronization algorithm
  - Flush barrier
  - FIFO and empty messages or send and receive counters
  - Additional overhead
- Possible optimization: Exploit distance between LPs

Parallel simulation: Summary/Outline

- Motivation: Why to use parallel simulators and why it is difficult
- Conservative algorithms: Only do what is allowed
  - Introduction
  - Null messages: deadlock avoidance
  - Deadlock detection and recovery
  - Barrier synchronization and LBTS (lower bound on time stamp) calculation
- Optimistic algorithms: Do anything, possibly roll back
  - Time Warp algorithm
  - Anti-Messages for rollback
  - Global Virtual Time (GVT) for fossil collection
  - State Saving Techniques
  - (Issues with Zero lookahead; Wide Virtual Time)
  - (Artificial Rollbacks to save memory)
- Summary
  - Pros and Cons of conservative and optimistic algorithms
  - Why not to use parallel simulators

Outline

- Optimistic Synchronization
- Time Warp
  - Local Control Mechanism
    - Rollback
    - Event cancellation
  - Global Control Mechanism
    - Global Virtual Time
    - Fossil Collection

The Synchronization Problem

Local causality constraint: Events within each logical process must be processed in time stamp order

Observation: Adherence to the local causality constraint is sufficient to ensure that the parallel simulation will produce exactly the same results as the corresponding sequential simulation*

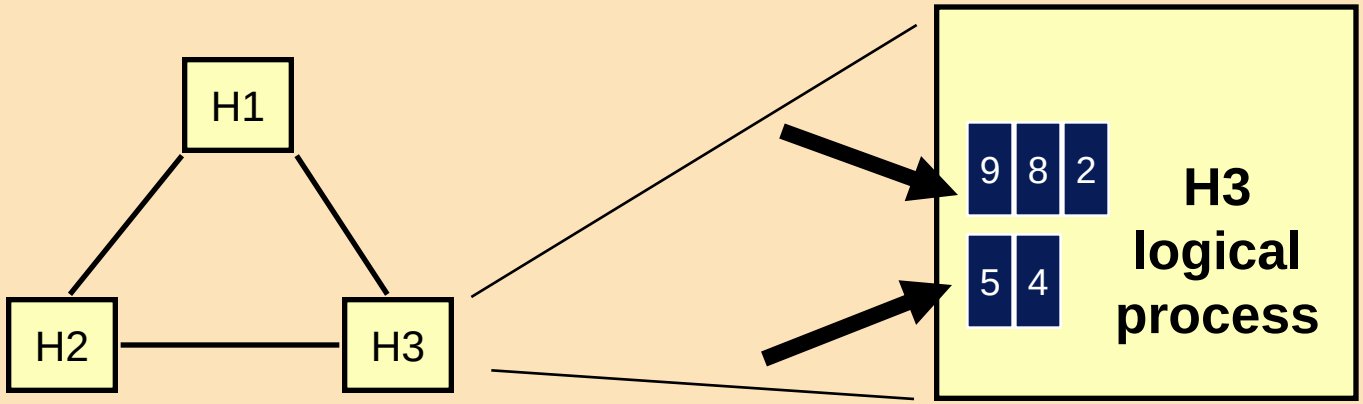
Synchronization Algorithms

- Conservative synchronization: Avoid violating the local causality constraint (wait until it is safe)
  - 1st generation: null messages (Chandy/Misra/Bryant)
  - 2nd generation: time stamp of next event
- Optimistic synchronization: Allow violations of local causality to occur, but detect them at runtime and recover using a rollback mechanism
  - Time Warp (Jefferson)
  - approaches limiting amount of optimistic execution

* provided events with the same time stamp are processed in the same order as in the sequential execution

Time Warp Algorithm (Jefferson)

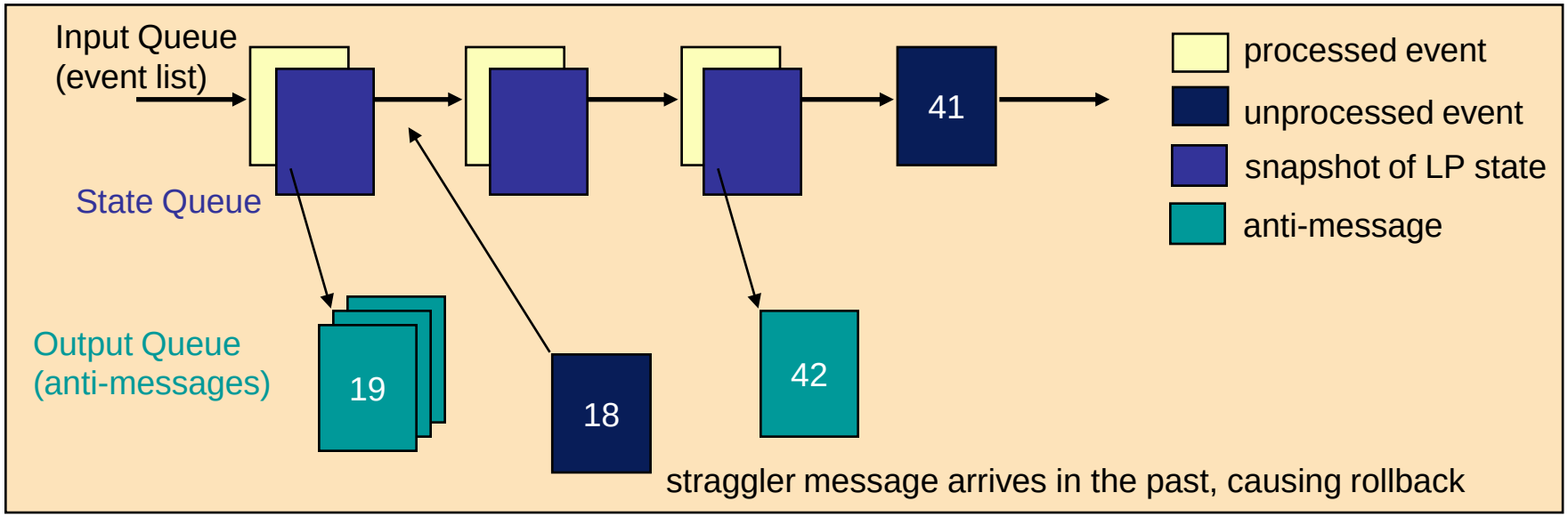
- **Assumptions**
  - Logical processes (LPs) exchanging time stamped events (messages)
  - Dynamic network topology is OK; dynamic creation of LPs is OK
  - Messages sent on each link need not be sent in time stamp order (!)
  - Network provides reliable delivery, but does not need to preserve order
- **Basic idea:**
  - Just go ahead and process events without worrying about messages that will arrive later
  - Detect out-of-order execution; in this case: recover using rollback



process all available events (2, 4, 5, 8, 9) in time stamp order

Time Warp: Local Control Mechanism

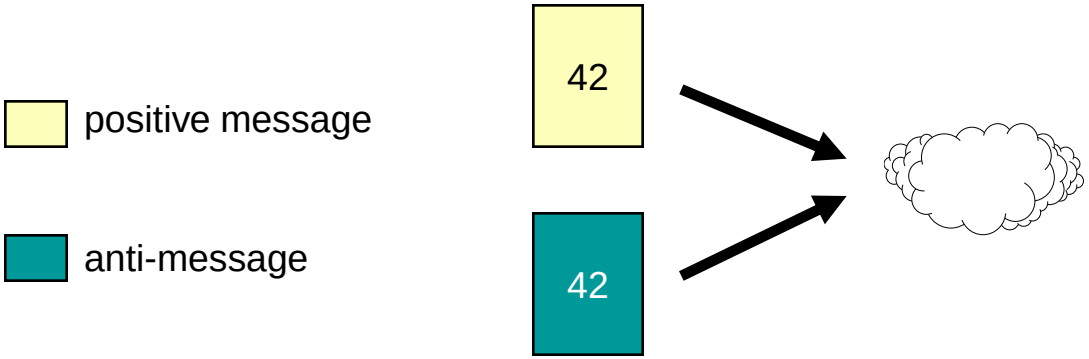
Each LP: process events in time stamp order, like a sequential simulator, except: (1) do NOT discard processed events and (2) add a rollback mechanism



Adding rollback:

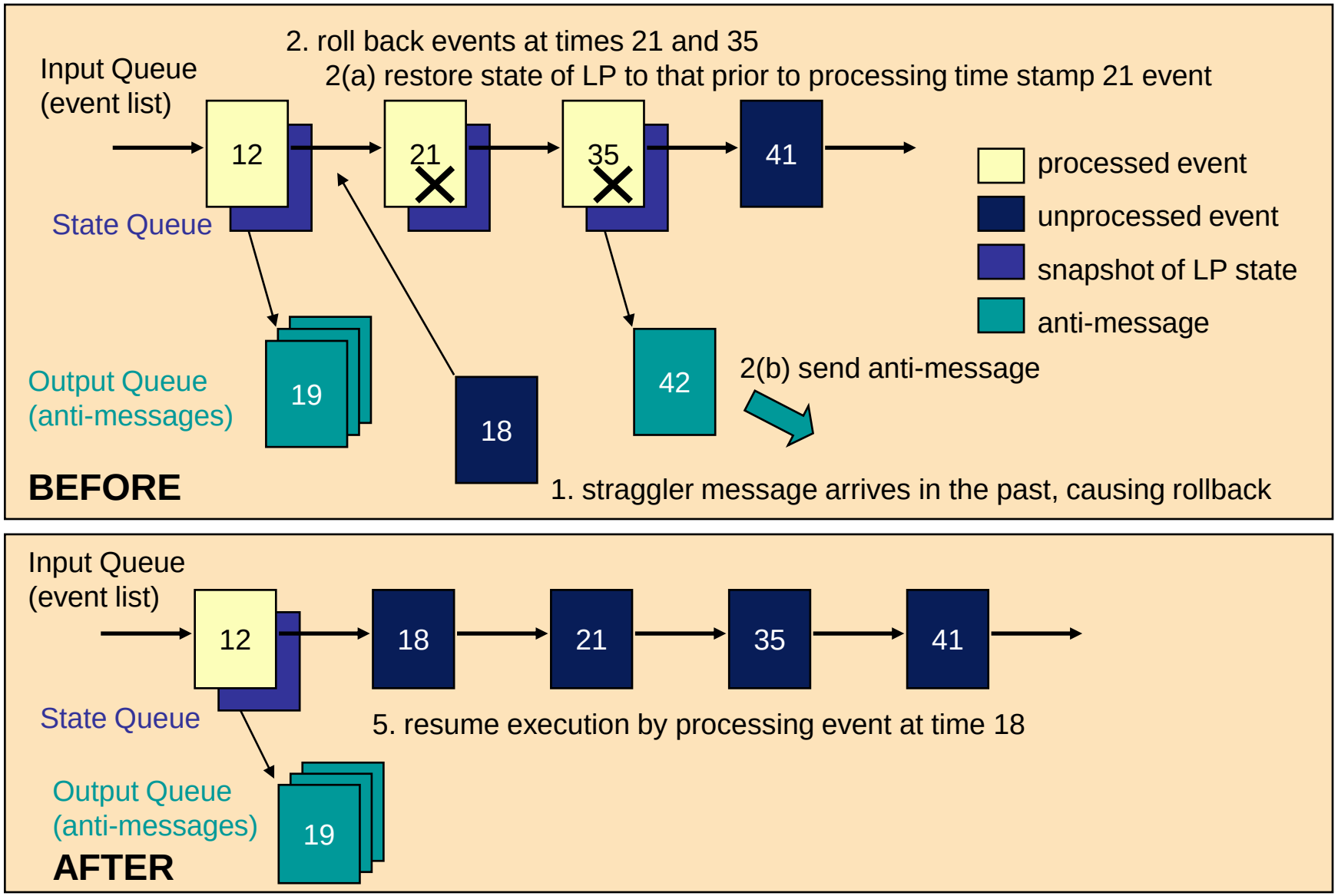
- a message arriving in the LP's past initiates rollback
- to roll back an event computation we must undo:
  - changes to state variables performed by the event;
    solution: checkpoint state or use incremental state saving (state queue)
  - message sends
    solution: anti-messages and message annihilation (output queue)

Anti-Messages



- Used to cancel a previously sent message
- Each positive message sent by an LP has a corresponding anti-message
- Anti-message is identical to positive message, except for a sign bit
- When an anti-message and its matching positive message meet in the same queue, the two annihilate each other (analogous to matter and anti-matter)
- To undo the effects of a previously sent (positive) message, the LP just needs to send the corresponding anti-message
- Message send: in addition to sending the message, leave a copy of the corresponding anti-message in a data structure in the sending LP called the output queue.

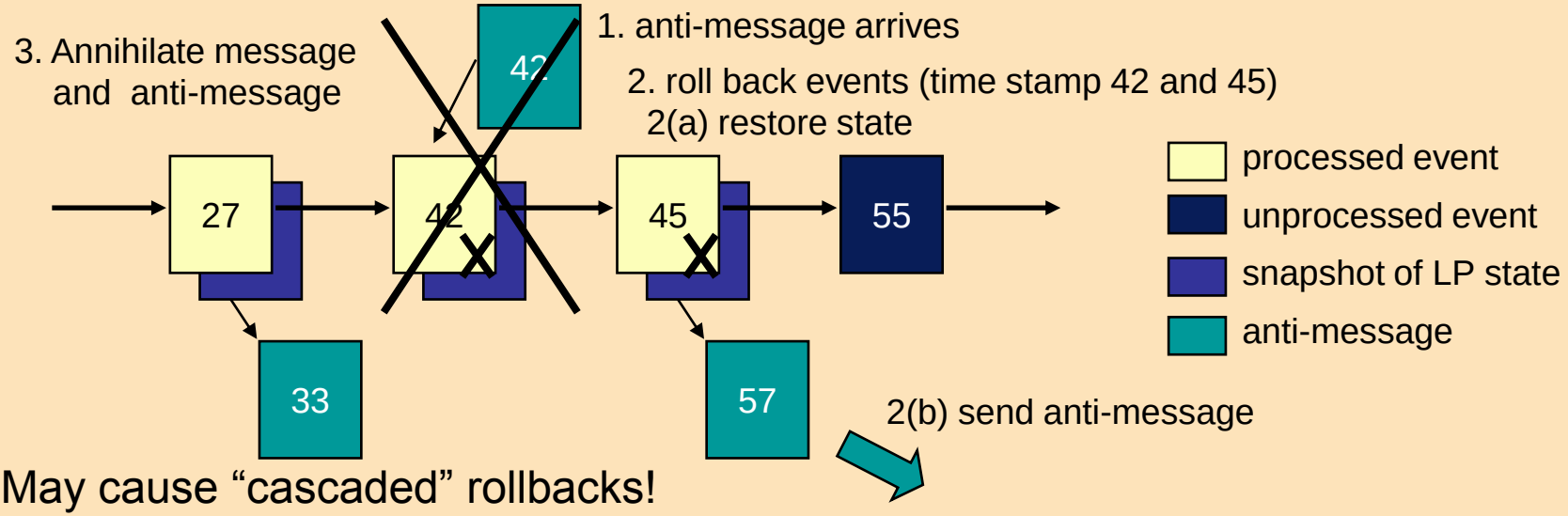
Rollback: Receiving a Straggler Message



Processing Incoming Anti-Messages

Case I: corresponding message has not yet been processed
- Simply annihilate message/anti-message pair; nothing else to do

Case II: corresponding message has already been processed
- roll back to time prior to processing message (secondary rollback)
- annihilate message/anti-message pair



- May cause “cascaded” rollbacks!
- Recursively applying eliminates all effects of error

Case III: corresponding message has not yet been received
- queue anti-message
- annihilate message/anti-message pair when message is received

Global Virtual Time and Fossil Collection

A mechanism is needed to:

- reclaim memory resources (e.g., old state and events)
  - Call this “fossil collection” (similar to garbage collection)
- perform irrevocable operations (e.g., I/O)

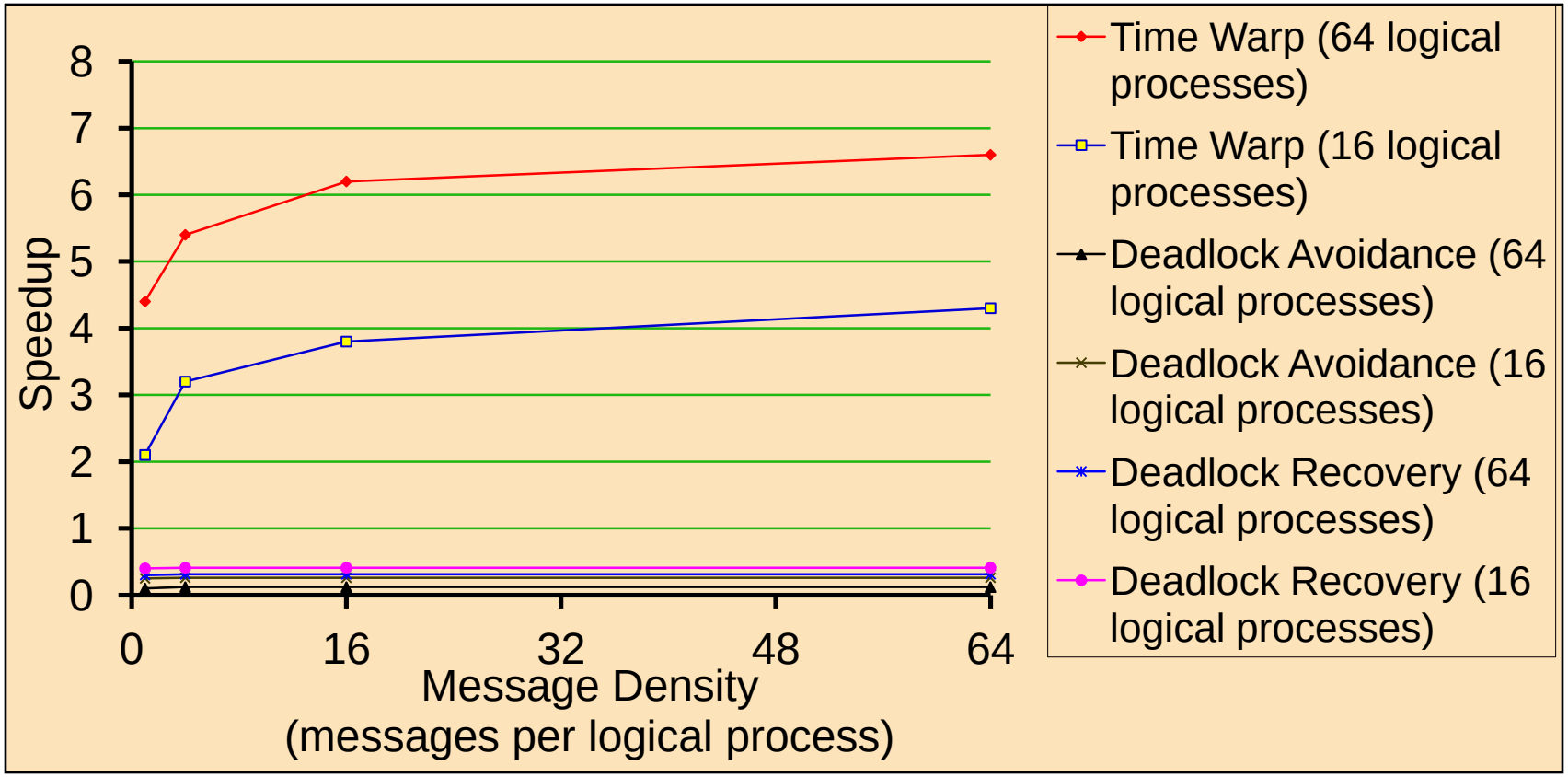
Observation: A lower bound on the time stamp of any rollback that can occur in the future is needed.

Global Virtual Time (GVT) is defined as the minimum time stamp of any unprocessed (or partially processed) message or anti-message in the system. GVT provides a lower bound on the time stamp of any future rollback.

- storage for events and state vectors older than GVT (except one state vector) can be reclaimed
- I/O operations with time stamp less than GVT can be performed.

Observation: The computation corresponding to GVT will not be rolled back, guaranteeing forward progress.

Time Warp and Chandy/Misra Performance



- eight processors
- closed queueing network, hypercube topology
- high priority jobs preempt service from low priority jobs (1% high priority)
- exponential service time (poor lookahead)

Summary

- Optimistic synchronization: detect and recover from synchronization errors rather than prevent them
- Time Warp
  - Local control mechanism
    - Rollback
    - State saving
    - Anti-messages
    - Cascaded rollbacks
  - Global control mechanism
    - Global Virtual Time (GVT)
    - Fossil collection to reclaim memory
    - Commit irrevocable operations (e.g., I/O)

Parallel simulation: Summary/Outline

- Motivation: Why to use parallel simulators and why it is difficult
- Conservative algorithms: Only do what is allowed
  - Introduction
  - Null messages: deadlock avoidance
  - Deadlock detection and recovery
  - Barrier synchronization and LBTS (lower bound on time stamp) calculation
- Optimistic algorithms: Do anything, possibly roll back
  - Time Warp algorithm
  - Anti-Messages for rollback
  - Global Virtual Time (GVT) for fossil collection
  - State Saving Techniques
  - (Issues with Zero lookahead; Wide Virtual Time)
  - (Artificial Rollbacks to save memory)
- Summary
  - Pros and Cons of conservative and optimistic algorithms
  - Why not to use parallel simulators

Outline

- GVT Computations: Introduction
  - Synchronous vs. Asynchronous
  - GVT vs. LBTS
- Computing Global Virtual Time
  - Transient Message Problem
  - Simultaneous Reporting Problem
- Samadi Algorithm (not shown)
  - Message Acknowledgements
  - Marked Acknowledgment Messages

Global Virtual Time

GVT(t): minimum time stamp among *all* unprocessed or partially processed messages at wallclock time t.

- Needed to commit I/O operations (e.g., write definite simulation output into logfile) and reclaim memory
- Computing GVT trivial if an instantaneous snapshot of the computation could be obtained: compute minimum time stamp among
  - Unprocessed events & anti-messages within each LP
  - Transient messages (messages sent before time t that are received after time t)
- Synchronous vs. Asynchronous GVT computation
  - Synchronous GVT algorithms: LPs stop processing events once a GVT computation has been detected
  - Asynchronous GVT algorithms: LPs can continue processing events and schedule new events while the GVT computation proceeds “in background”

GVT vs. LBTS

Observation:
Computing GVT is similar to computing the lower bound on time stamp (LBTS) of future events in conservative algorithms

- GVT algorithms can be used to compute LBTS and vice versa
- Both determine the minimum time stamp of messages (or anti-messages) that may later arrive
  - Historically, developed separately
  - Often developed using different assumptions (lookahead, topology, etc.)
- Time Warp
  - Latency to compute GVT typically less critical than the latency to compute LBTS
  - Asynchronous execution of GVT computation preferred to allow optimistic event processing to continue

Asynchronous GVT

An incorrect GVT algorithm:

- Controller process: broadcast “compute GVT request”
- upon receiving the GVT request, each process computes its local minimum and reports it back to the controller
- Controller computes global minimum, broadcast to others

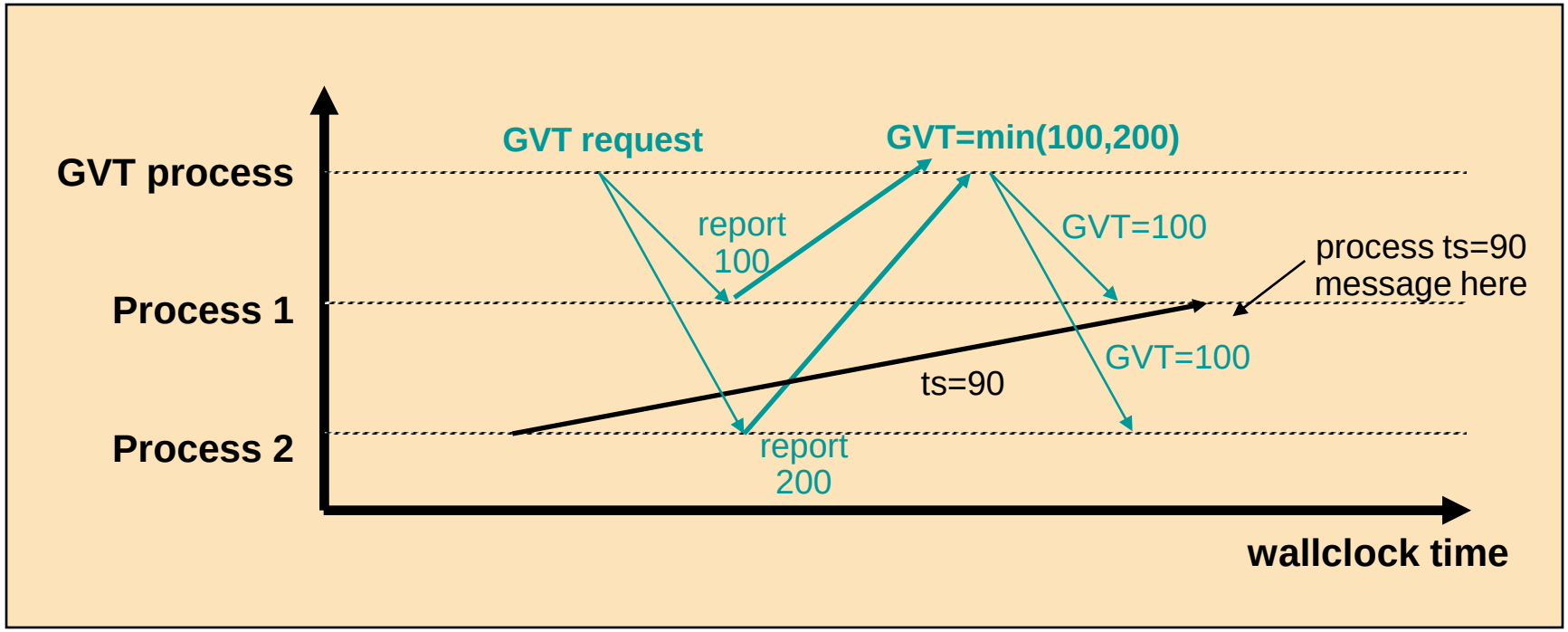
Difficulties:

- *transient message problem*: messages sent, but not yet received must be considered in computing GVT
- *simultaneous reporting problem*: different processors report their local minima at different points in wallclock times, leading to an incorrect GVT value

Solutions exist (e.g., Sammadi algorithm), but won’t show them here

The Transient Message Problem

- Transient message: A message that has been sent, but has not yet been received at its destination
- Erroneous values of GVT may be computed if the algorithm does not take into account transient messages



Samadi Algorithm

- Calculates GVT
- Requires acknowledgements on event messages
- Transient message problem:
  Handled by message acknowledgements
- Simultaneous reporting problem:
  Mark acknowledgements sent after reporting local minimum

Summary

- Global Virtual Time
  - Similar to lower bound on time stamp (LBTS)
    - Time Warp: GVT usually not as time critical as LBTS
    - Asynchronous GVT computation highly desirable to avoid unnecessary blocking
  - Transient message problem etc. handled by, e.g., Samadi algorithm

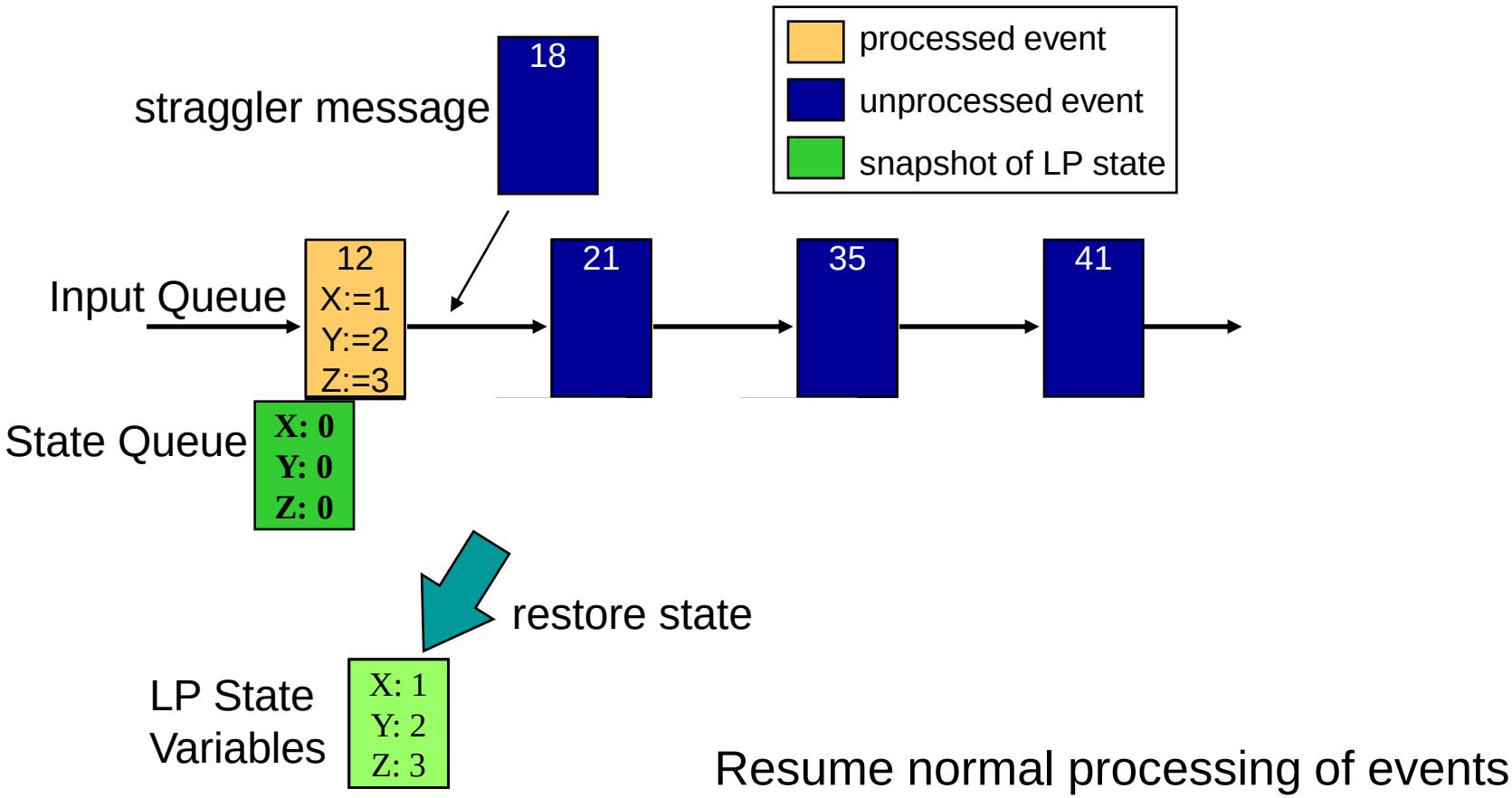
Parallel Simulation: Summary/Outline

- Motivation: Why to use parallel simulators and why it's difficult
- Conservative algorithms: Only do what is allowed
  - Introduction
  - Null messages: deadlock avoidance
  - Deadlock detection and recovery
  - Barrier synchronization and LBTS (lower bound on time stamp) calculation
- Optimistic algorithms: Do anything, possibly roll back
  - Time Warp algorithm
  - Anti-Messages for rollback
  - Global Virtual Time (GVT) for fossil collection
  - State Saving Techniques
  - (Issues with Zero lookahead; Wide Virtual Time)
  - (Artificial Rollbacks to save memory)
- Summary
  - Pros and Cons of conservative and optimistic algorithms
  - Why not to use parallel simulators

Outline

- State Saving Techniques
  - Copy State Saving
  - Infrequent State Saving
  - Incremental State Saving
  - Reverse Computation

Copy State Saving



- Checkpoint all modifiable state variables of the LP prior to processing each event
- Rollback: copy stored checkpoint state to LP state variables

Copy State Saving

Drawbacks

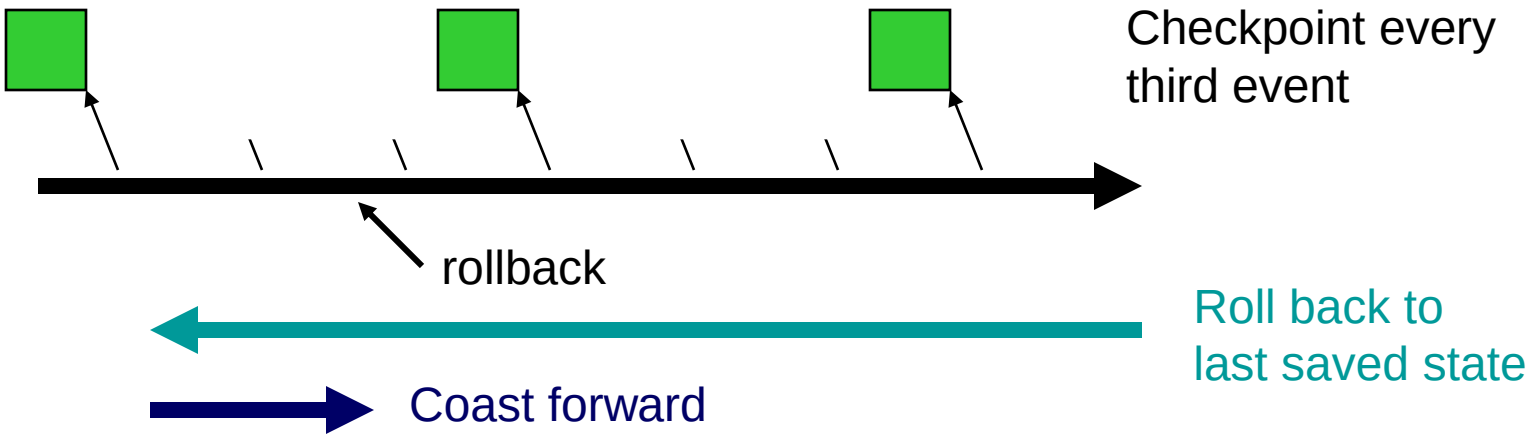
- Forward execution slowed by checkpointing
  - Must state save even if no rollbacks occur
  - Inefficient if most of the state variables are not modified by each event
- Consumes large amount of memory

⇒ Copy state saving is only practical for LPs that do not have a large state vector

⇒ Largely transparent to the simulation application (only need locations of LP state variables)

Infrequent State Saving

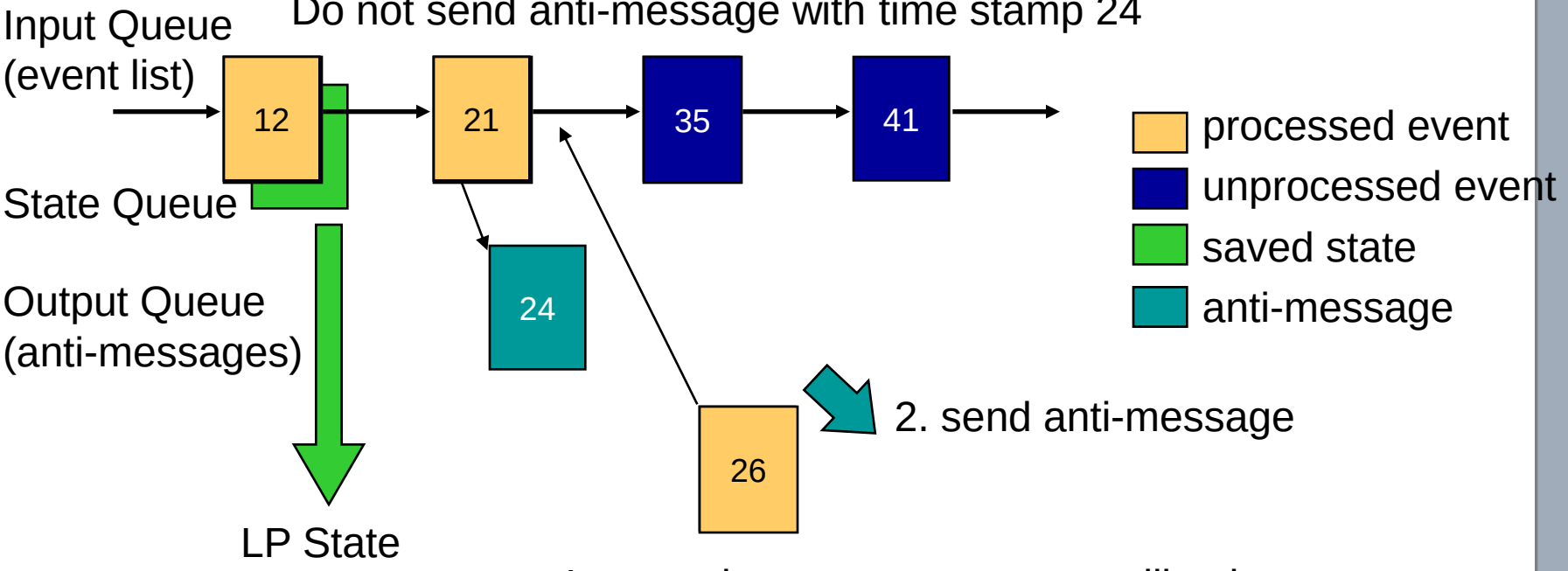
- Checkpoint LP periodically, e.g., every N-th event
- Rollback to sim time T – but may not have saved state at time T!
  - Roll back to most recent checkpointed state prior to time T
  - Execute forward (“coast forward”) to time T



- Coast forward phase
  - Only needed to recreate state of LP at simulation time T
  - Coast forward execution identical to the original execution
  - Must “turn off” message sends during coast forward, or else
    - rollback to T could cause new messages with time stamp < T, and roll backs to times earlier than T
    - Could lead to rollbacks earlier than GVT

Infrequent State Saving Example

3. Roll back to simulation time 12
   Restore state of LP to that prior to processing time stamp 12 event
   Do not send anti-message with time stamp 24



4. Coast forward: reprocess event with time stamp 12
5. Coast forward: reprocess event with time stamp 21,
   do not resend time stamp 24 message
6. Process straggler, continue normal event processing

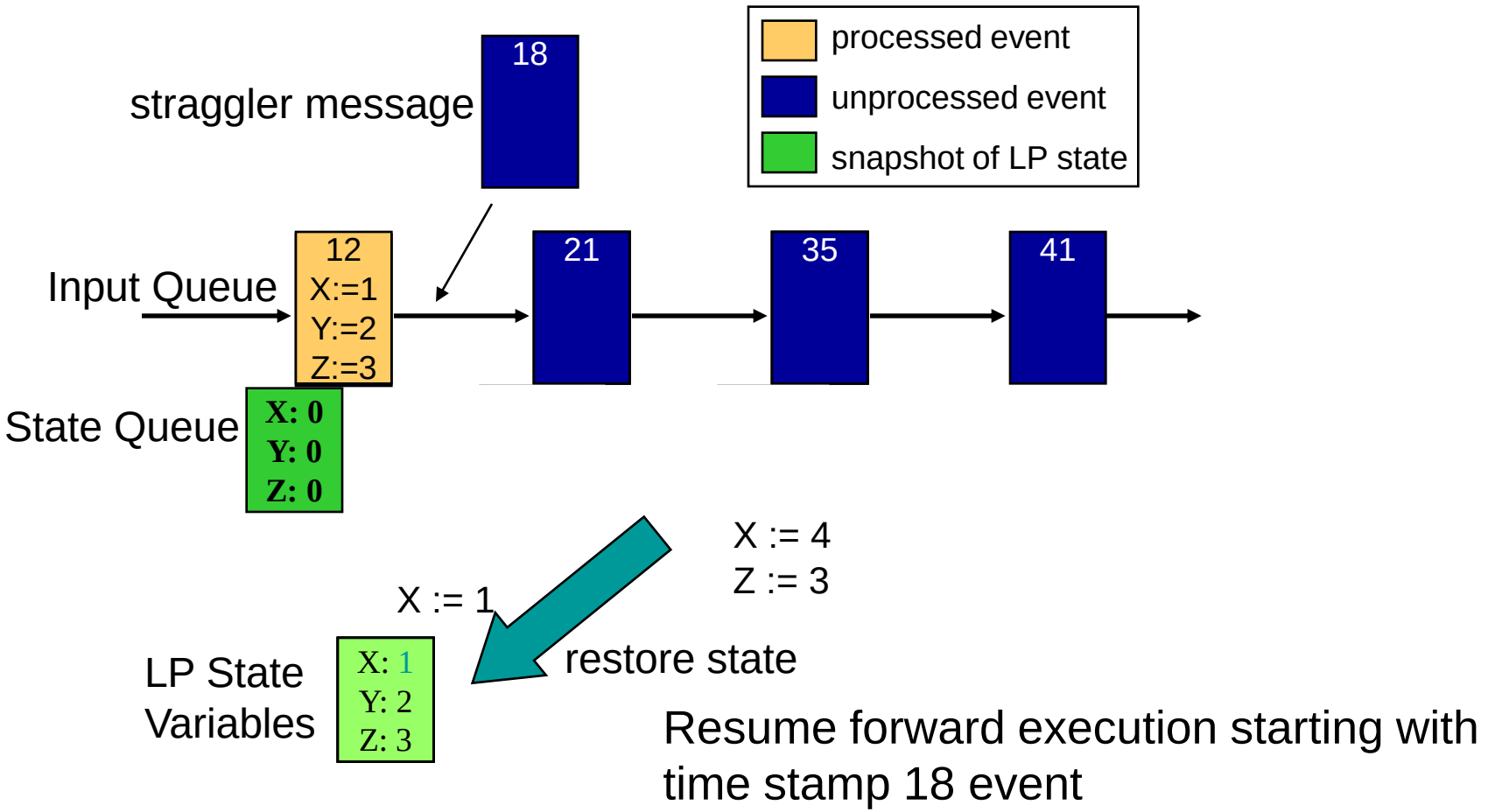
Infrequent State Saving: Pros and Cons

- Reduces time required for state saving
- Reduces memory requirements
- Increases time required to roll back LP
- Increases complexity of Time Warp executive
- Largely transparent to the simulation application (only need locations of LP state variables and frequency parameter)

Incremental State Saving

- Only state save variables modified by an event
  - Generate “change log” with each event indicating previous value of state variable before it was modified
- Rollback
  - Scan change log in reverse order, restoring old values of state variables

Incremental State Saving Example



- Before modifying a state variable, save current version in state queue
- Rollback: Scan state queue from back, restoring old values

Incremental State Saving

- Must log addresses of modified variables in addition to state
- More efficient than copy state save if most state variables are not modified by each event
- Can be used *in addition* to copy state save
  - Changing some variables may be faster than to change the entire state
  - Do full state copies every now and then;
    do incremental saving in between
    (similar to MPEG i-frames vs. p- and b-frames)

Reverse Computation

- Rather than state save, recompute prior state
  - For each event computation, need inverse computation
  - Instrument forward execution to enable reverse execution
- Advantages
  - Reduce overhead in forward computation path
  - Reduce memory requirements
- Disadvantages
  - Tedious to do by hand, requires automation

Parallel Simulation: Summary/Outline

- Motivation: Why to use parallel simulators and why it is difficult
- Conservative algorithms: Only do what is allowed
  - Introduction
  - Null messages: deadlock avoidance
  - Deadlock detection and recovery
  - Barrier synchronization and LBTS (lower bound on time stamp) calculation
- Optimistic algorithms: Do anything, possibly roll back
  - Time Warp algorithm
  - Anti-Messages for rollback
  - Global Virtual Time (GVT) for fossil collection
  - State Saving Techniques
  - (Issues with Zero lookahead; Wide Virtual Time)
  - (Artificial Rollbacks to save memory)
- Summary
  - Pros and Cons of conservative and optimistic algorithms
  - Why not to use parallel simulators

Issues

Zero lookahead: An LP has zero lookahead if it can schedule an event with time stamp equal to the current simulation time of the LP

Simultaneous events: events containing the same time stamp; in what order should they be processed?

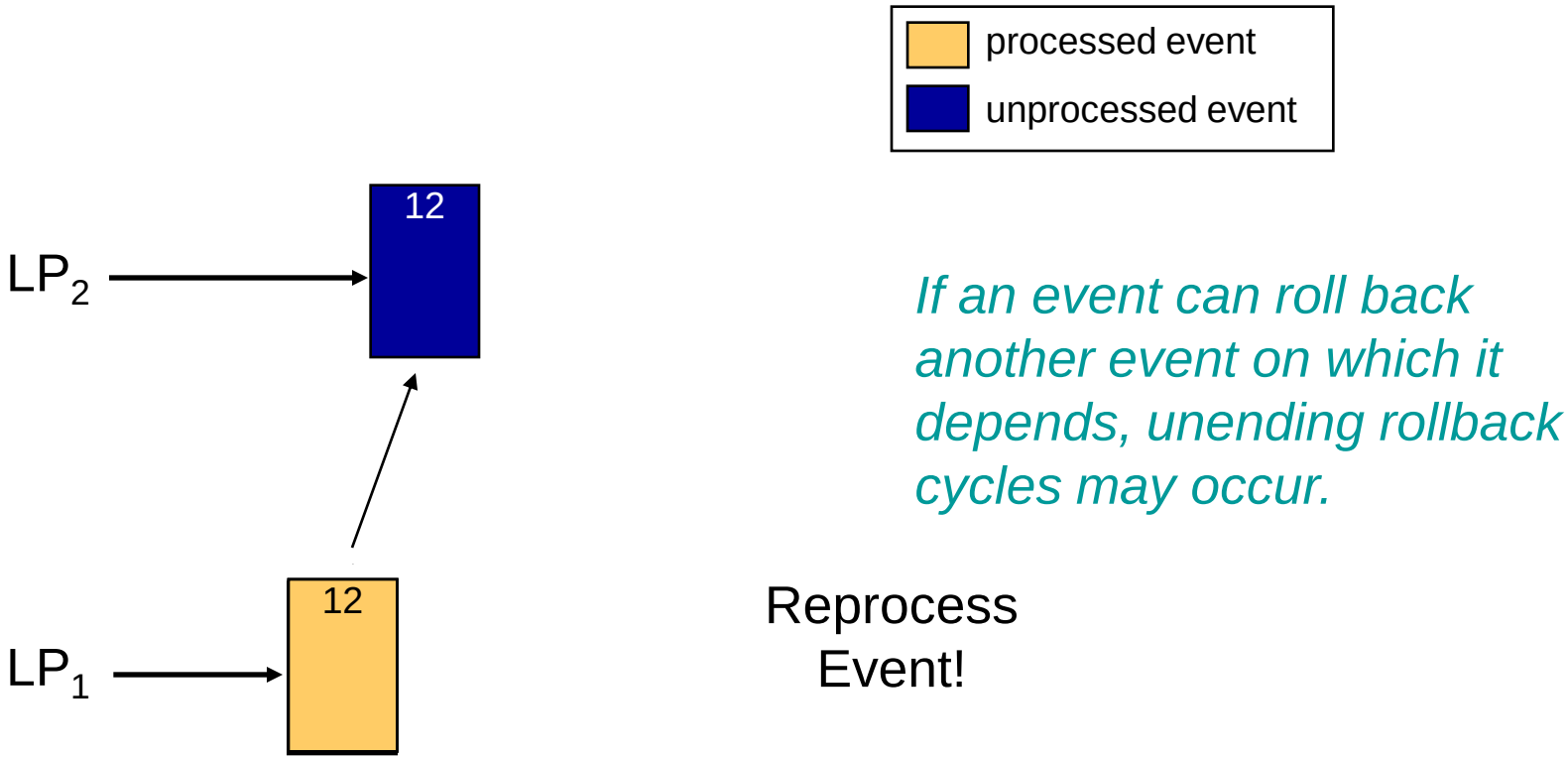
Repeatability: An execution mechanism (e.g., Time Warp) is repeatable if repeated executions produce exactly the same results

- Often a requirement
- Simplifies debugging

Zero Lookahead and Simultaneous Events

- Time Warp: Do simultaneous events cause rollback?
- A possible rule:

  If an LP processes an event at simulation time T and then receives a new event with time stamp T, roll back the event that has already been processed.

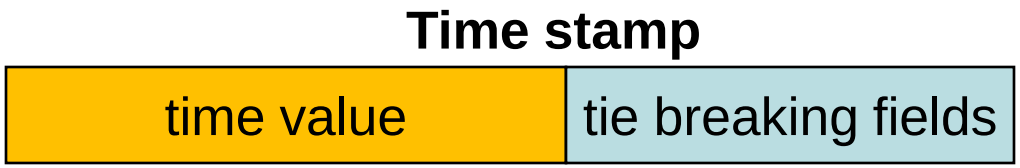


If an event can roll back another event on which it depends, unending rollback cycles may occur.

Wide Virtual Time (WVT)

Approach

- Application uses time value field to indicate “time when the event occurs”
- Tie breaking field used to order simultaneous events (events with same time value): Make time “artificially more precise”



- Tie breaking field can be viewed as low precision bits of time stamp
- Time definition applies to all simulation time values (e.g., current time of an LP)

An Approach Using WVT

Time stamp:

time value	priority	age	LP ID	Seq #

Application specified ordering of events:

- Application specified priority field
- Constraint on zero lookahead events

Avoid rollback cycles:

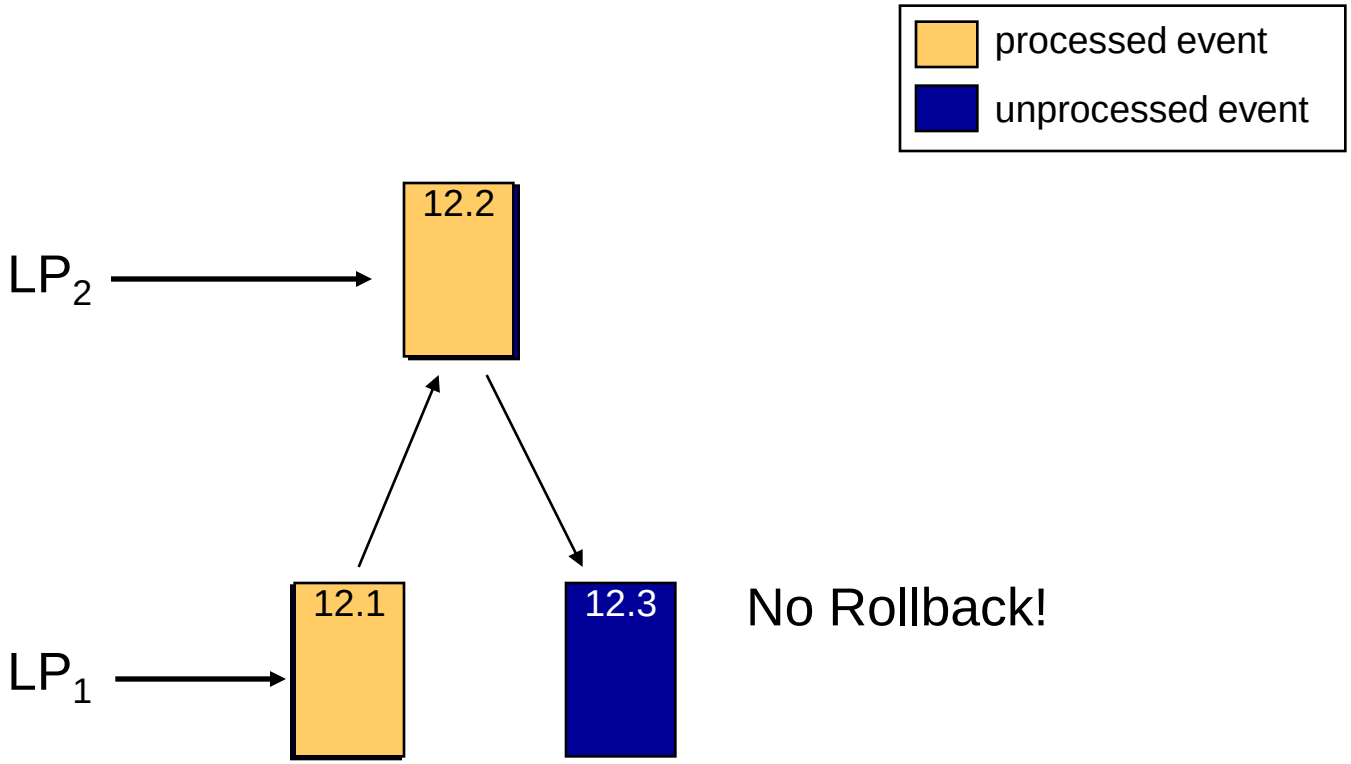
- Age field to order dependent zero lookahead events
- Non-zero lookahead events: Age=1
- Zero lookahead events: Age = Current Age + 1

Repeatable execution

- ID field identifying LP that scheduled the event
- Sequence number indicating # of events scheduled by LP

WVT Example

Avoid rollback cycles despite zero lookahead events



Summary

- Copy State Saving
  - Efficient if LP state small
  - Can be made transparent to application
- Infrequent state saving
  - Must turn off message sending during coast forward
  - Reduced memory requirements
  - less time for state saving
  - Increased rollback cost
- Incremental State Saving
  - Preferred approach if large state vectors
  - Means to simplify usage required
- Reverse computation
  - Efficient, requires automation
- Zero lookahead and simultaneous events
  - Can lead to unending rollbacks
  - Wide Virtual Time provides one solution

Parallel Simulation: Summary/Outline

- Motivation: Why to use parallel simulators and why it's difficult
- Conservative algorithms: Only do what is allowed
  - Introduction
  - Null messages: deadlock avoidance
  - Deadlock detection and recovery
  - Barrier synchronization and LBTS (lower bound on time stamp) calculation
- Optimistic algorithms: Do anything, possibly roll back
  - Time Warp algorithm
  - Anti-Messages for rollback
  - Global Virtual Time (GVT) for fossil collection
  - State Saving Techniques
  - (Issues with Zero lookahead; Wide Virtual Time)
  - (Artificial Rollbacks to save memory)
- Summary
  - Pros and Cons of conservative and optimistic algorithms
  - Why not to use parallel simulators

Observations

- In a sequential execution at simulation time T, the event list contains the events with
  - Receive time stamp greater than T,
  - Send time stamp less than T.
- Time Warp can restore the execution to a valid state if it retains events with
  - Send time less than GVT and receive time stamp greater than GVT.
  - All other events can be deleted (as well as their associated state vector, anti-messages, etc.)
- **Storage optimal protocols:** roll back LPs to reclaim all memory not required in corresponding sequential execution

Artificial Rollback

Salvage parameter: Amount of memory to be reclaimed when a processor runs out of memory

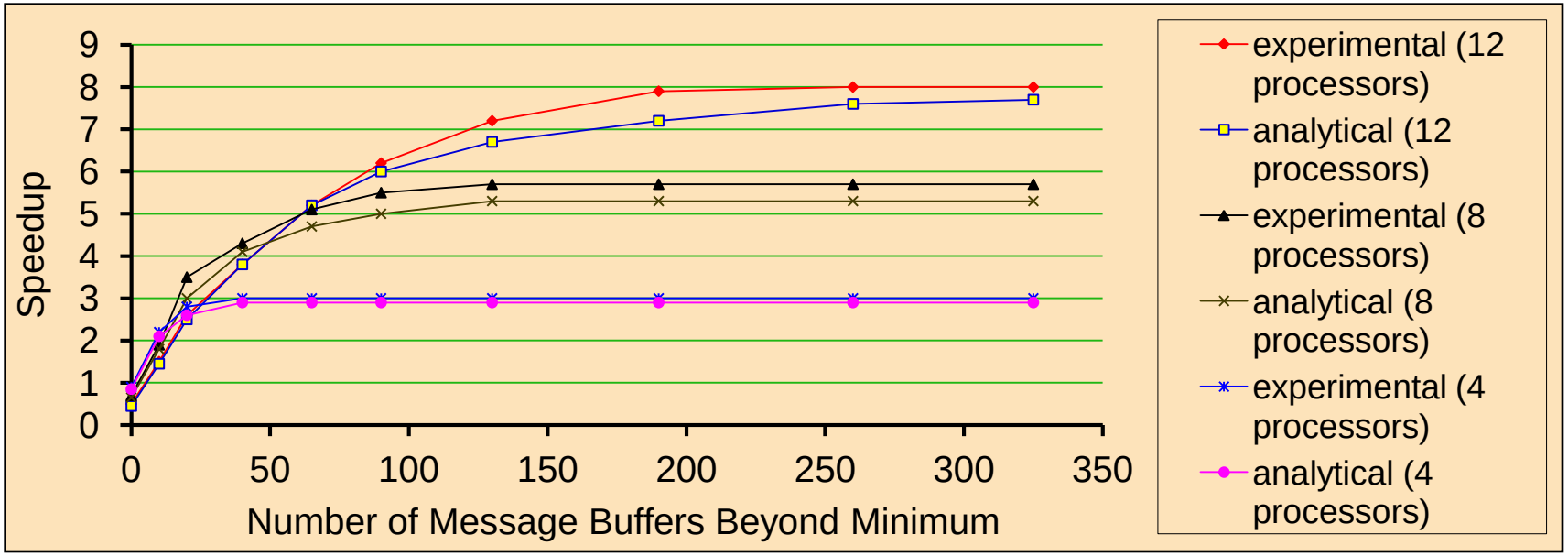
Algorithm: When system runs out of memory, then…:

- Sort LPs, in order of their current simulation time (largest to smallest): LP^1, LP^2, LP^3, …
- Roll back LP^1 to current simulation time of LP^2
- If additional memory must be reclaimed, roll back LP^1 and LP^2 to current simulation time of LP^3
- Repeat above process until sufficient memory has been reclaimed

Artificial rollback is storage optimal when executed on a shared memory multiprocessor with a shared buffer pool

Performance will be poor if too little memory is available

Effect of Limited Memory on Speedup



- symmetric synthetic workload (PHold)
- one logical processor per processor
- fixed message population
- KSR-1 multiprocessor
- sequential execution requires 128 (4 LPs), 256 (8 LPs), 384 (12 LPs) buffers

Other Optimistic Algorithms

Principal goal: avoid excessive optimistic execution

A variety of protocols have been proposed, among them:

- window-based approaches
  - only execute events in a moving window (simulated time, memory)
- risk-free execution
  - only send messages when they are guaranteed to be correct
- add optimism to conservative protocols
  - specify “optimistic” values for lookahead
- introduce additional rollbacks
  - triggered stochastically or by running out of memory
- hybrid approaches
  - mix conservative and optimistic LPs
- scheduling-based
  - discriminate against LPs rolling back too much
- adaptive protocols
  - dynamically adjust protocol during execution as workload changes

Parallel simulation: Summary/Outline

- Motivation: Why to use parallel simulators and why it's difficult
- Conservative algorithms: Only do what is allowed
  - Introduction
  - Null messages: deadlock avoidance
  - Deadlock detection and recovery
  - Barrier synchronization and LBTS (lower bound on time stamp) calculation
- Optimistic algorithms: Do anything, possibly roll back
  - Time Warp algorithm
  - Anti-Messages for rollback
  - Global Virtual Time (GVT) for fossil collection
  - State Saving Techniques
  - (Issues with Zero lookahead; Wide Virtual Time)
  - (Artificial Rollbacks to save memory)
- Summary
  - Pros and Cons of conservative and optimistic algorithms
  - Why not to use parallel simulators

Summary: Conservative vs. Optimistic

- Conservative systems:
  Only process events that are safe to process
- Optimistic systems:
  Go ahead processing, but be prepared to do a rollback

Conservative Algorithms

Pro:

- Good performance reported for many applications containing good lookahead (queueing networks, communication networks, war gaming)
- Relatively easy to implement
- Well suited for “federating” autonomous simulations, provided there is good lookahead

Con:

- Cannot fully exploit available parallelism in the simulation because they must protect against a “worst case scenario”
- Lookahead is essential to achieve good performance
- Writing simulation programs to have good lookahead can be very difficult or impossible, and can lead to code that is difficult to maintain

Optimistic Algorithms

Pro:

- good performance reported for a variety of application (queuing networks, communication networks, logic circuits, combat models, transportation systems)
- offers the best hope for “general purpose” parallel simulation software (not as dependent on lookahead as conservative methods)
- “Federating” autonomous simulations
- avoids specification of lookahead
- caveat: requires providing rollback capability in the simulation

Con:

- state saving overhead may severely degrade performance
- rollback thrashing may occur (though a variety of solutions exist)
- implementation is generally more complex and difficult to debug than conservative mechanisms; careful implementation is required or poor performance may result
- must be able to recover from exceptions (may be subsequently rolled back)

Further observations

- Simple operations in conservative systems (dynamic memory allocation, error handling) present non-trivial issues in Time Warp systems
- Solutions exist for most, but at the cost of increased complexity in the Time Warp executive

Parallel simulation: Summary/Outline

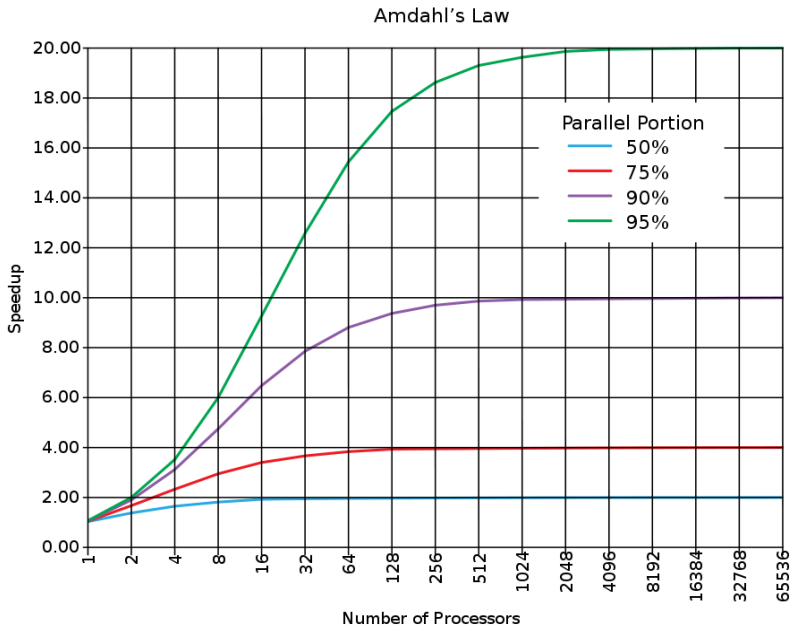
- Motivation: Why to use parallel simulators and why it's difficult
- Conservative algorithms: Only do what is allowed
  - Introduction
  - Null messages: deadlock avoidance
  - Deadlock detection and recovery
  - Barrier synchronization and LBTS (lower bound on time stamp) calculation
- Optimistic algorithms: Do anything, possibly roll back
  - Time Warp algorithm
  - Anti-Messages for rollback
  - Global Virtual Time (GVT) for fossil collection
  - State Saving Techniques
  - (Issues with Zero lookahead; Wide Virtual Time)
  - (Artificial Rollbacks to save memory)
- Summary
  - Pros and Cons of conservative and optimistic algorithms
  - Why not to use parallel simulators

Why not parallel / distributed simulation? 1/2

- Parallel computing is more complicated
  - More complicated = more code = more bugs
  - It is more complicated to avoid bugs in parallel / distributed programs
  - It is more complicated to debug parallel / distributed programs
- Parallel / distributed simulations are more complicated
  - Conservative?
    - Design model to be well-parallelizable
    - Need to think about lookahead
  - Optimistic?
    - Less difficult model; more difficult simulator
    - Needs more RAM
    - How often will I have to do rollbacks?

Why not parallel / distributed simulation? 2/2

- Parallel / distributed computing introduces overhead:
  - CPU overhead for time synchronisation, rollbacks, …
  - Memory overhead for keeping snapshots, message counters, …
  - Time / communication overhead: Waiting for messages, sending ACKs, message transit times, …
- Amdahl's law: Boundary for any parallel program
  - Speedup < #CPUs
  - Reason: There are always some non-parallelizable parts in the program



Another approach to parallelize…

- Often, you have to run multiple simulations anyway
  - Different random seeds to get more measurements
  - Try out different parameter sets
- In this case, the easiest way to parallelize your simulation is to run them in parallel…
  - ☺ Scales linearly with the number of CPUs (…unless RAM size, RAM accesses or disk I/O are bottlenecks)
  - ☺ No overhead for locking, synchronizing etc. means more efficient use of CPU time than a parallel simulator
  - ☺ Output always deterministic
  - ☺ Less complex than a parallel simulator
    - ☺Easier to debug
    - ☺No thoughts about improving lookahead in the model, etc.
  - ☹ Often needs more RAM than a parallel simulator (…although Time Warp may be a memory hog, too)
  - ☹ Does not work with hardware-in-the-loop, human-in-the-loop etc.

Parallel simulation: Summary/Outline

- Motivation: Why to use parallel simulators and why it's difficult
- Conservative algorithms: Only do what is allowed
  - Local causality constraint
  - Null messages: deadlock avoidance
    - Lookahead / Time creep problem
  - Deadlock detection and recovery
  - Barrier synchronization
    - LBTS (lower bound on time stamp) calculation
    - Transient message problem and flush barriers
  - Possible optimization: Exploit distance between processes
- Optimistic algorithms: Do anything, possibly roll back
  - Time Warp algorithm / Anti-Messages for rollback
  - Global Virtual Time (GVT) for fossil collection
  - State Saving Techniques / (Issues with Zero lookahead; Wide Virtual Time)
  - (Artificial Rollbacks to save memory)
- Summary
  - Pros and Cons of conservative and optimistic algorithms
  - Why not to use parallel simulators