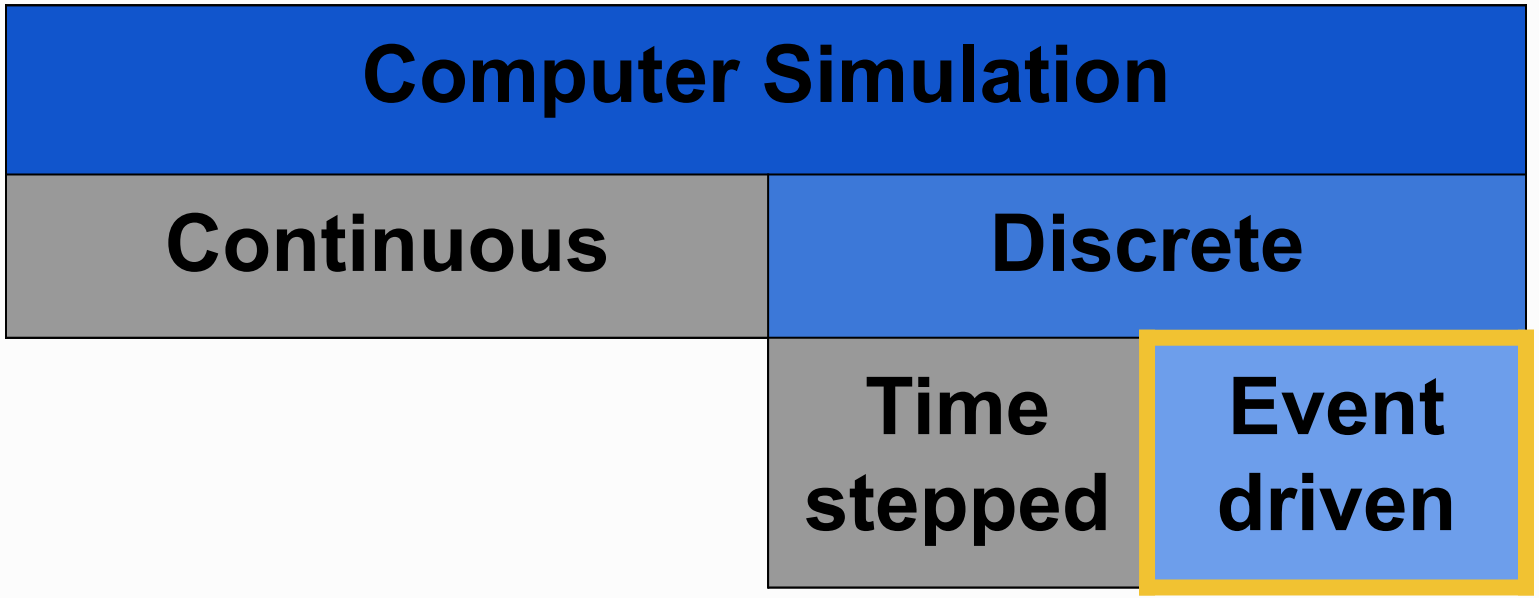
Parallel Discrete Event Simulation

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Classification of Simulations



Discrete Event Simulations

- Logical Processes (LPs) execute events
- Executing an event updates the LP's state
- Events have a virtual timestamp
- Events must be executed in order

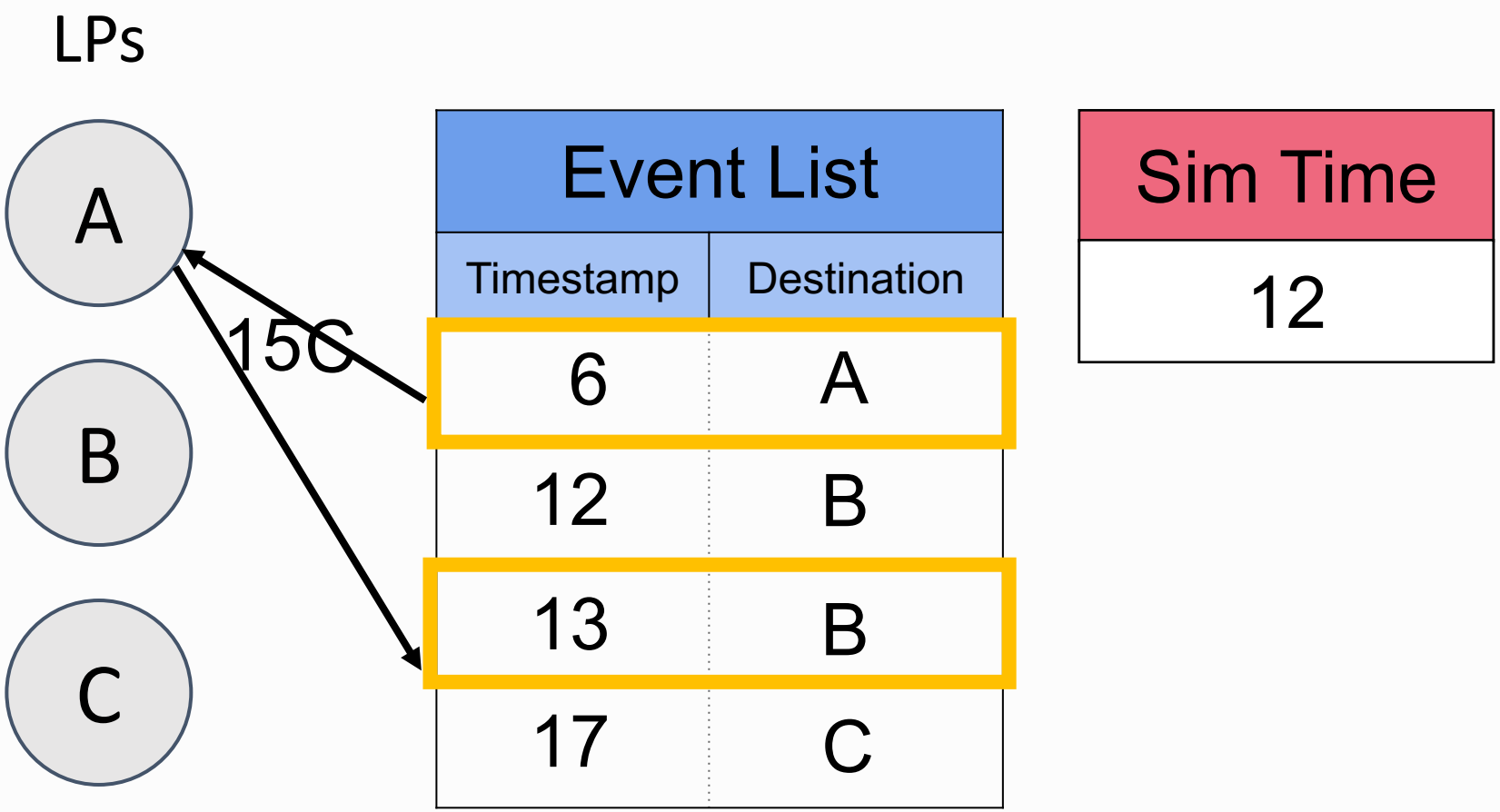
Applications

- Traffic analysis
- Military battles
- Networks
- Circuits
- Economic models
- and many more...

Sequential Implementation

- Single event queue
- Sorted by timestamp
- Loop over queue and execute events
- Efficiency depends on queue used
- Very simple

Discrete Event Simulations



Pseudocode

```
while (running) {
      Event* e = eventList.pop();
      LP* lp = e->destination();
      lp->execute(e);
}
```

Pretty simple right?

Limitations of Sequential DES

- Millions of LPs
- Billions of events to simulate
- Sequential simulations will take too long

Challenges to Parallelization

- Events are generally very small/fast
- Simulations can have of billions of events
- Order must be maintained

Benefits of parallelism come from executing many events at once

How do we parallelize DES?

- Distribute LPs across processors
- Each processor has its own event list and virtual clock
- How to synchronize event lists and clocks?

(Super) Naive Implementation

```
while (running) {
      AllReduce(simTime);
      if (events.top()->ts==simTime) {
            Event* e = events.pop();
            LP* lp = e->destination();
            lp->execute(e);
      }
}
```

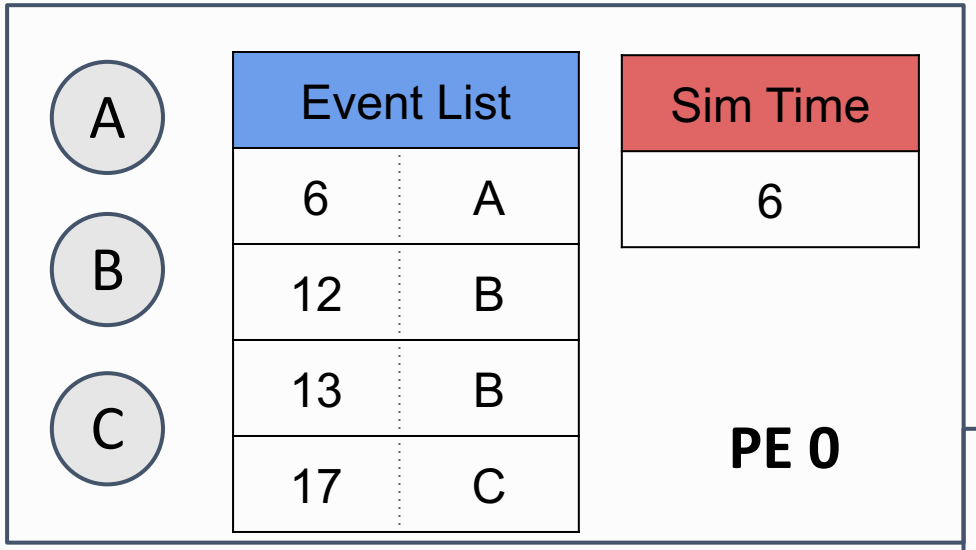
(Super) Naive Implementation

- Exactly matches sequential semantics
- NO PARALLELISM
- Way too much synchronization

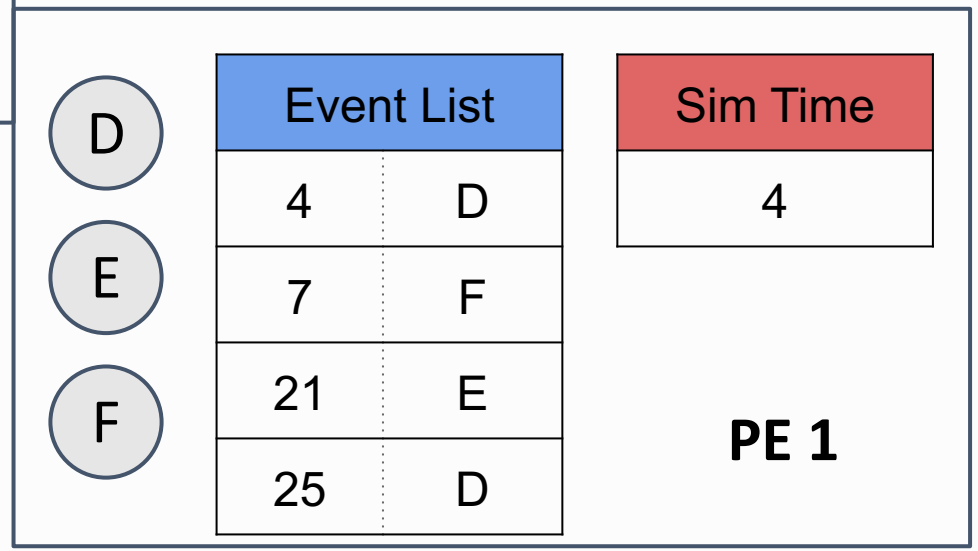
Moral of the Story:

We need to relax our ordering restrictions!

What is the fundamental problem?



Can PE 0 safely execute any events?

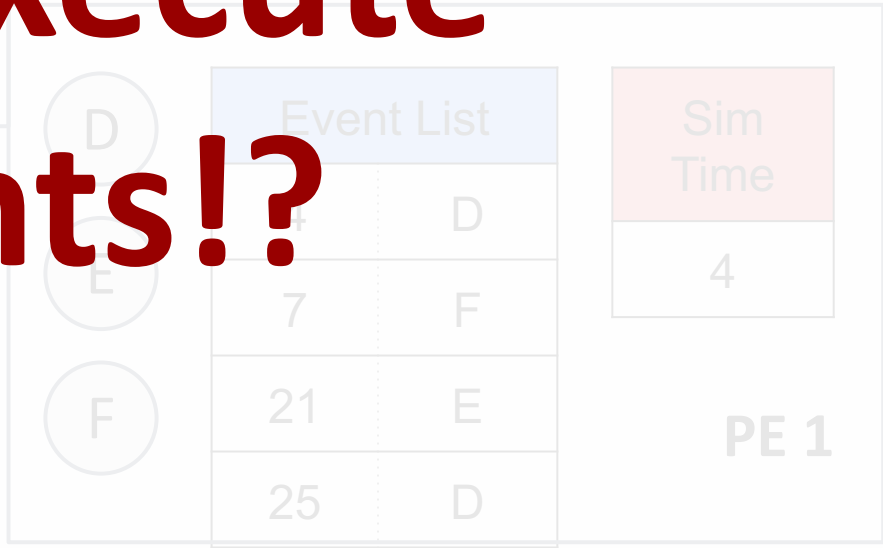


What if the first event on PE 1 generates event (5,A)?

Causality Error!

What is the fundamental problem?

Can PE 0 safely execute any events?



What if the first event on PE 1 generates event (5,A)?

How can we ever execute events!?

Two Approaches

Conservative

(don't screw up)

- Only execute events we know are safe
- Synchronize often to determine safe events

- More synchronization
- Less parallelism
- Inflexible
+ Simple

Optimistic

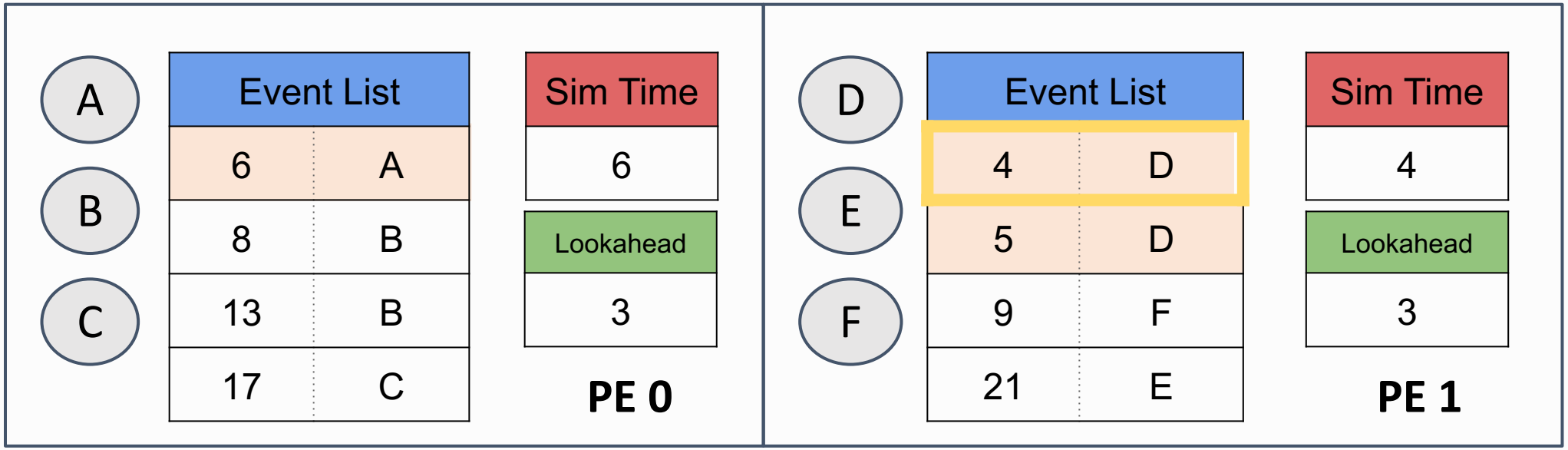
(if we screw up, we'll fix it)

- Execute events freely
- Rollback the processor if there's an error

+ Less synchronization
+ More parallelism
+ Flexible
- Complex

Conservative (Windowed)

- Requires a model-specific lookahead :3
- Determine the min timestamp in the system :4
- Execute events in (min + lookahead) window: (4+3)



(Simple) Event driven implementation

```
while (running) {
      AllReduce(simTime);
      while (events.top().ts >= simTime + lookahead) {
            Event* e = events.pop();
            LP* lp = e->destination();
            lp->execute(e);
      }
}
```

Windowed Analysis

- Very similar to naive solution
- Performance depends on lookahead
- Low lookahead = low parallelism and high synchronization
- Workload can be unbalanced

Parallel Discrete Event Simulation

Optimistic (Time Warp) synchronization

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Optimistic Concurrency Control

- Also called the time-warp method, or speculative parallelism
- Execute events freely
- When an event is received with a smaller timestamp than your clock, rollback
- How do we rollback efficiently?
- How do rollbacks affect performance?
- How many PEs will a rollback affect?

Rollbacks

- Save previous events (How many?)
- Revert your own state (How?)
- Cancel sent events (How?)

Saving Previous Events

- Events take up memory
  - Limits how many events we can save
  - Need to reclaim memory periodically
- What can we safely reclaim?
- Find the Global Virtual Time (GVT)
  - Minimum clock time of the system
  - Everything prior to this can be reclaimed
  - Events with observable effects can be committed

GVT

- Global synchronization required
- All events must be accounted for
- Can be synchronous or asynchronous

Reverting Your State

- State saving
  - Save the states of LPs after each event
  - Rolling back is equivalent to reverting states
  - High memory consumption
  - Need to reclaim memory more often
- Reverse computation
  - During rollback execute events in reverse
  - Better for memory
  - Overhead of executing in reverse
  - Reverse computation is complex
  - Can compilers help?

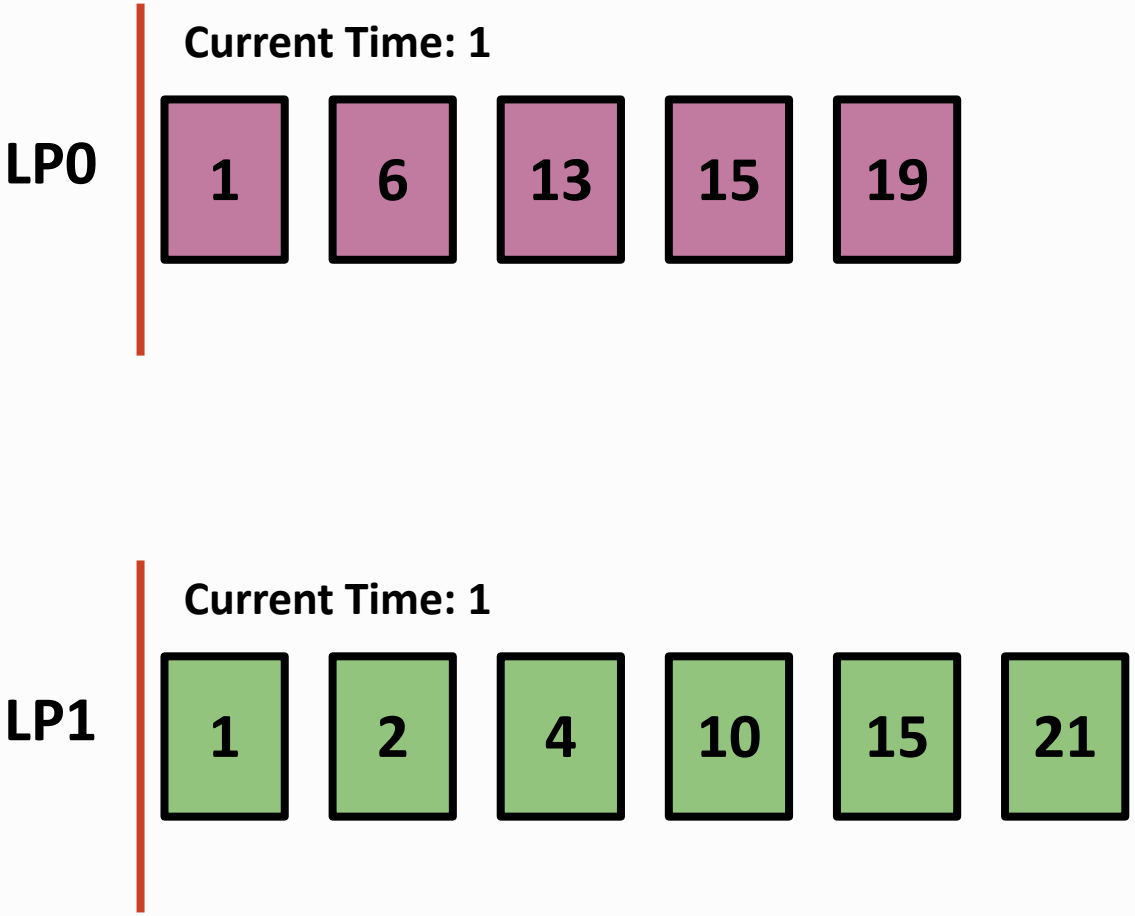
Cancelling Events

- Events sent erroneously must be cancelled
- First we must find the event
  - If it’s local, that’s easy
  - If it’s remote we need to send an anti-event
- Then we must cancel it
  - If they weren’t executed, just delete them
  - If they have been executed, do a rollback
- Rollbacks can snowball out of control

Pseudocode

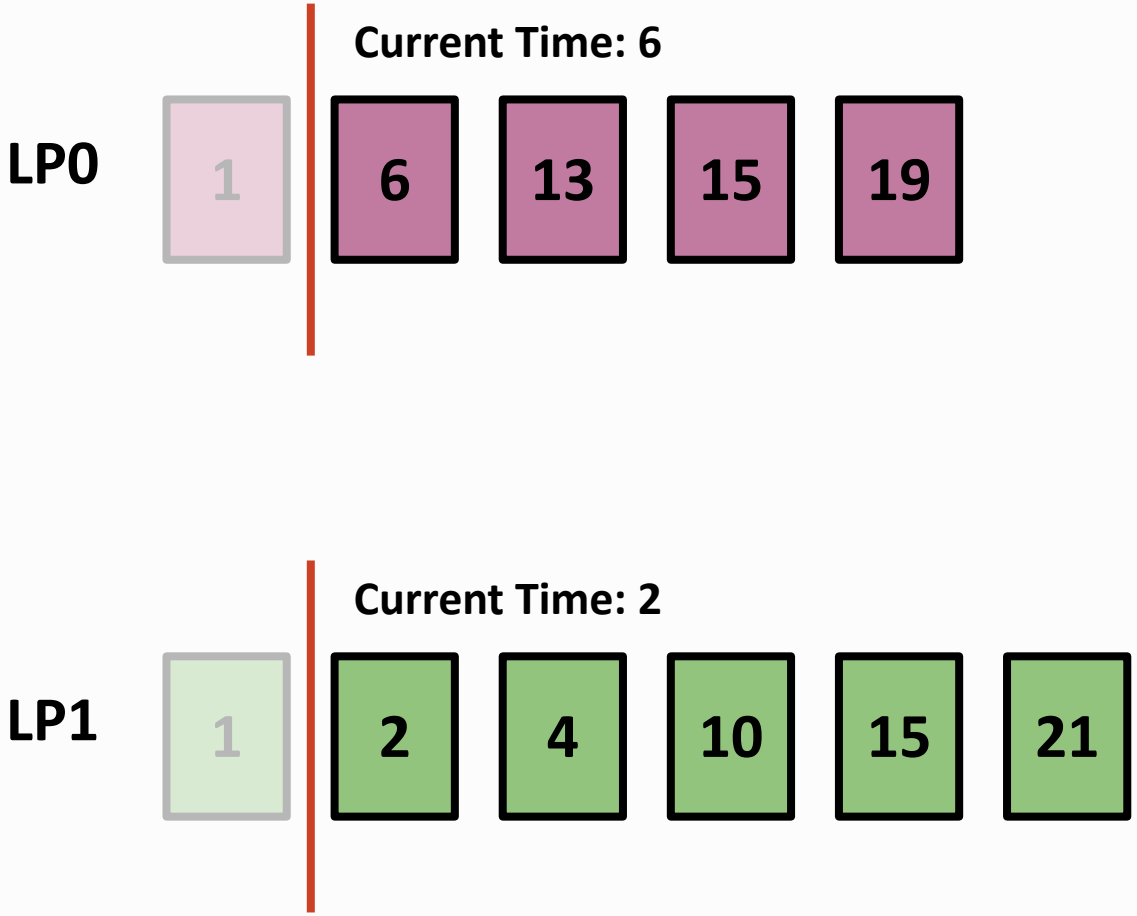
```
while (running) {
     while (executing_events) {
          check_for_rollbacks();
          Event* e = events.pop();
          LP* lp = e->destination();
          lp->execute(e);
     }
     compute_gvt();
}
```

A (small) Example



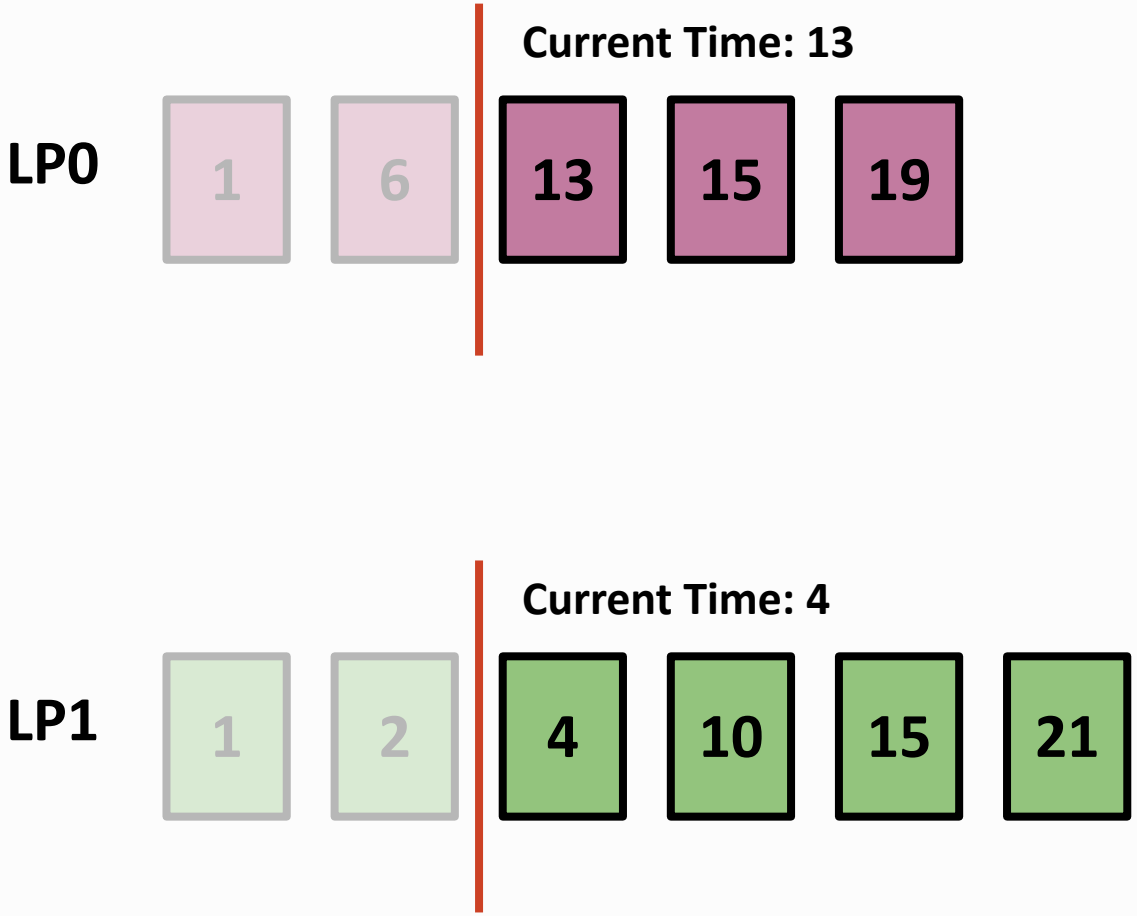
Total Events: 0
Total Rollbacks: 0
Committed Events: 0

A (small) Example



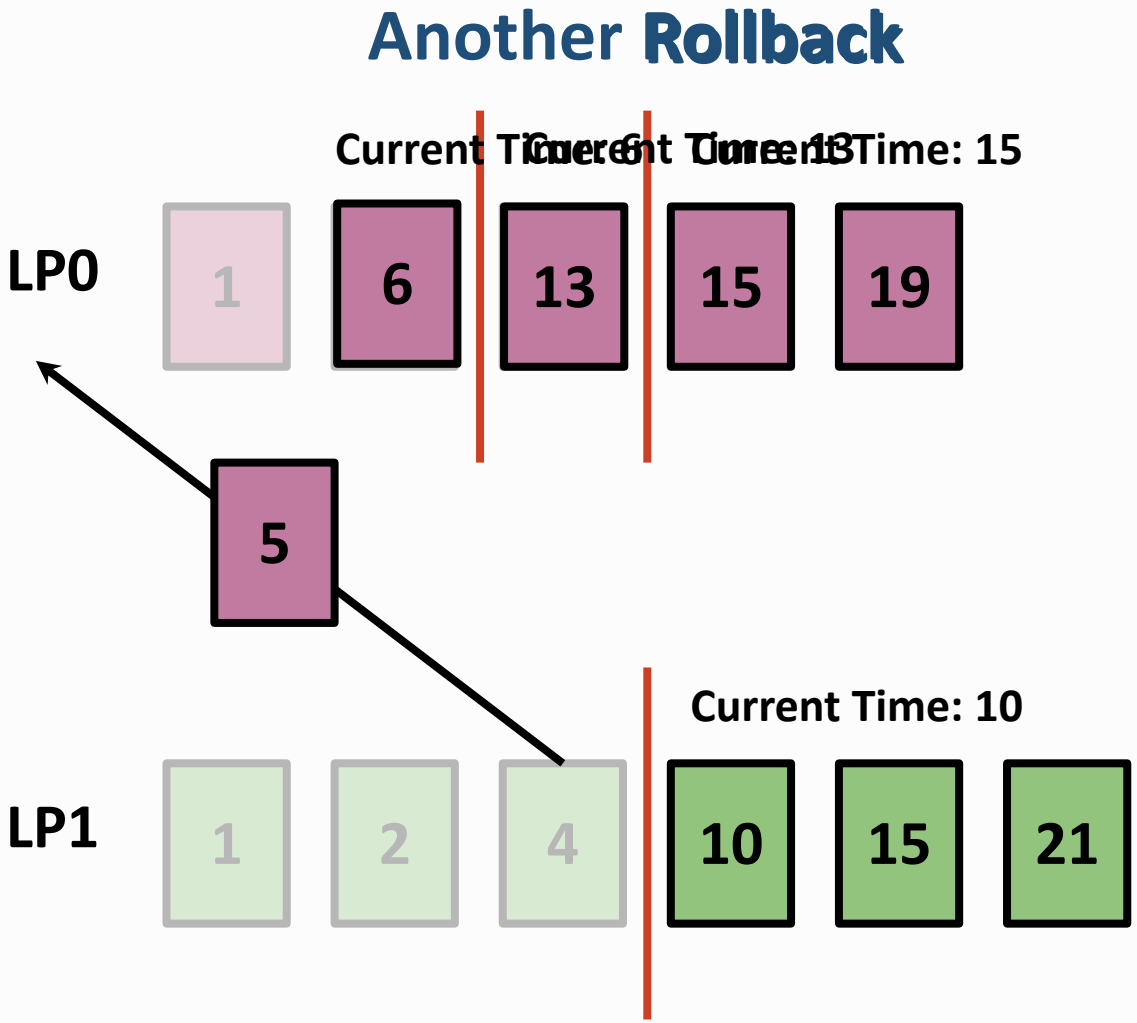
Total Events: 2
Total Rollbacks: 0
Committed Events: 0

A (small) Example



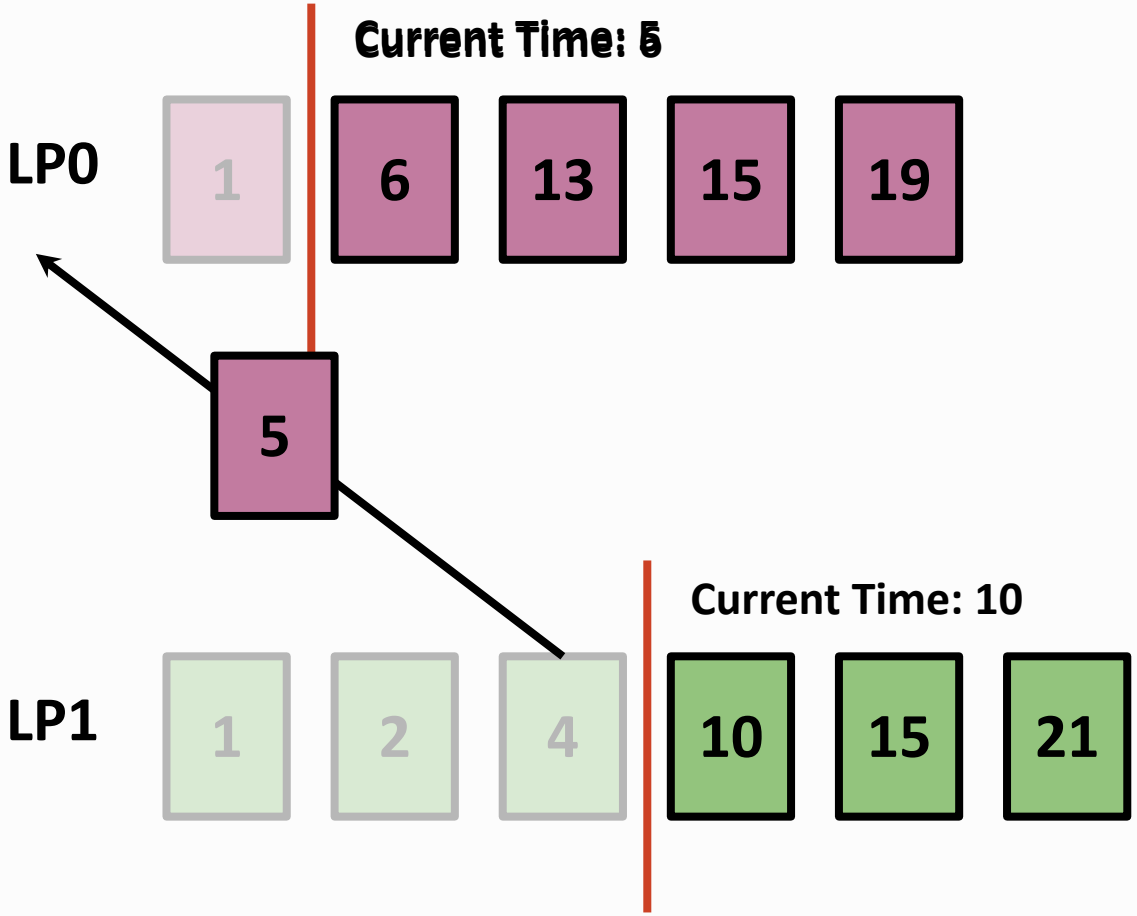
Total Events: 4
Total Rollbacks: 0
Committed Events: 0

A (small) Example



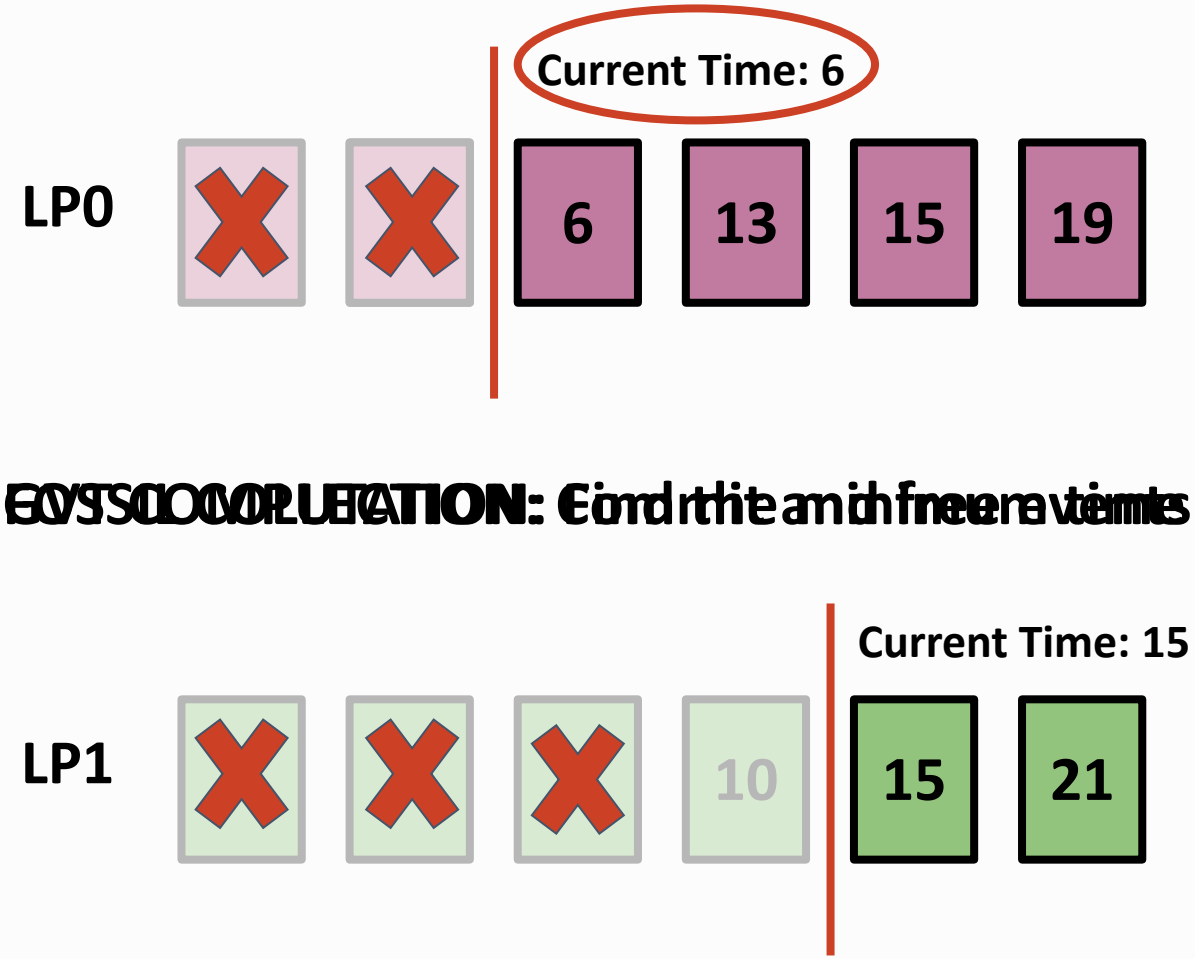
Total Events: 6
Total Rollbacks: 1
Committed Events: 0

A (small) Example



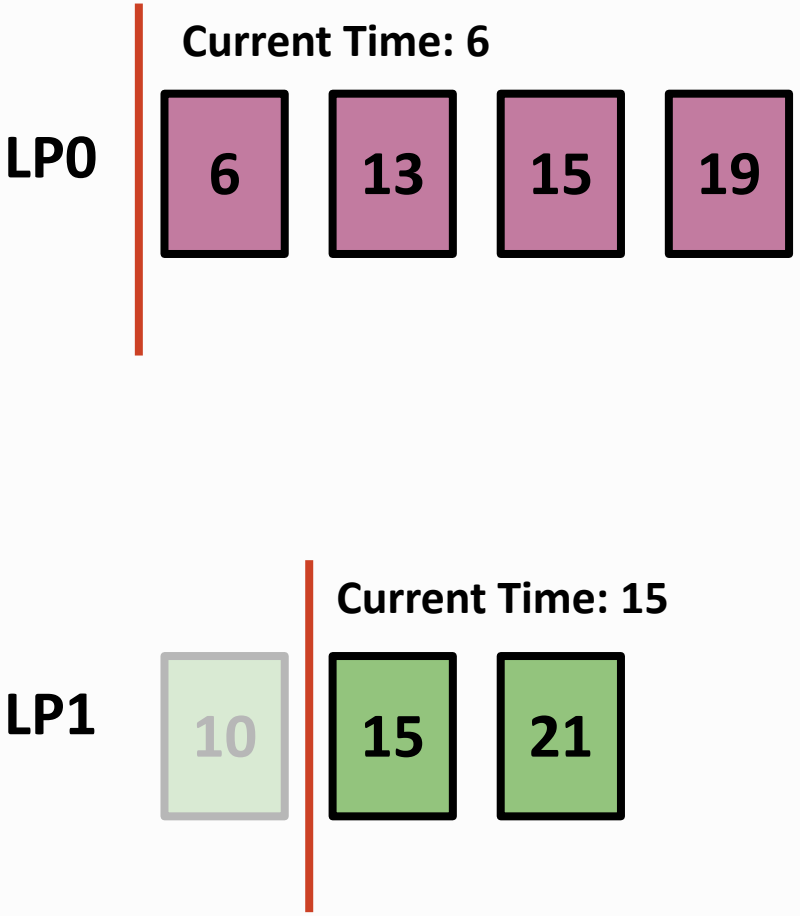
Total Events: 6
Total Rollbacks: 2
Committed Events: 0

A (small) Example



Total Events: 8
Total Rollbacks: 2
Committed Events: 6

A (small) Example



Total Events: 8
Total Rollbacks: 2
Committed Events: 5

Event Efficiency: E_C/E_T

5/8 = 62%

Summary

Two Main Classes of PDES:

- Conservative
  - Low parallelism/High synchronization cost
  - Model dependent
  - Low memory footprint
- Optimistic
  - High parallelism/Low synchronization cost
  - Model independent
  - Memory Hungry
  - Rollbacks can snowball