**UNIVERSITY OF WATERLOO**
**Department of Electrical and Computer Engineering**

# Fujimoto's Parallel Discrete Event Simulation

**Christopher Trudeau**
ctrudeau@etude.uwaterloo.ca

**Parallel and Distributed Systems Research Group**

slides by *SlickSlides*

# Outline

Richard M. Fujimoto, "Parallel Discrete Event Simulation". *Communications of the ACM*, October 1990, 33(10), pp31-53.

1. Background
2. Conservative Methods
3. Optimistic Methods
4. Comparison

# Background

- simulations inherently parallel
- simplest: parallel run of different batches
- often difficult to exploit parallelism in simulation
- **Amdahl's Law** – law of diminishing returns
- **Speed-up** – number of times faster the parallel application runs than the equivalent sequential application
- speed-up is always compared to the number of processors
- **Linear Speed-up** – "n" times speed up for "n" processors; theoretical limit
- **Super-linear Speed-up** – possible to obtain numbers that exceed "n" but it means that the sequential case is non-optimal

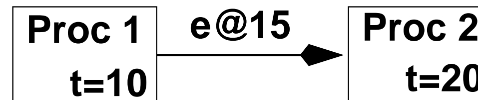
# Why is PDES Hard?

- sequential simulations typically use:
  1. state variables
  2. event list
  3. global clock
- events are time stamped & the program removes the smallest time event from the queue and activates it
- **CAUSALITY** – events in the future can not effect events in the past
- by having multiple processes running at a time it becomes difficult to determine what is the smallest time
- two solutions:
  1. conservative time – advance by the amount of time that works for all processes
  2. optimistic time – each process advances on its own and then there are corrective methods to keep causality

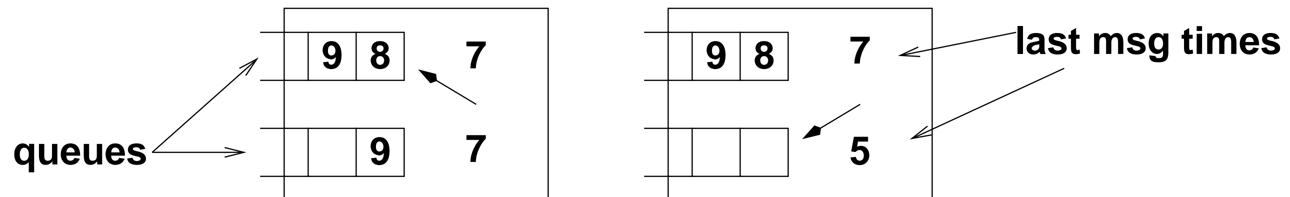
# Parallel Computing can be UGLY

- **Granularity** – how many logical processes for each physical processes/processor (compare circuits versus manufacturing)
- **Partitioning** – what aspects are to be parallelized and how to place these sections on the machine (battle simulation: sector vs troop partitioning)
- **Shared Memory** – variables that are to be accessed by separate processes (e.g. global state variables)
- how do you synchronize distinct process access to shared variables? $\Rightarrow$ read/write order, accidental over-writes, etc.
- how does the machine handle multiple processes?
- scheduling
- coordination between processes
- message passing
- process synchronization

# Causality

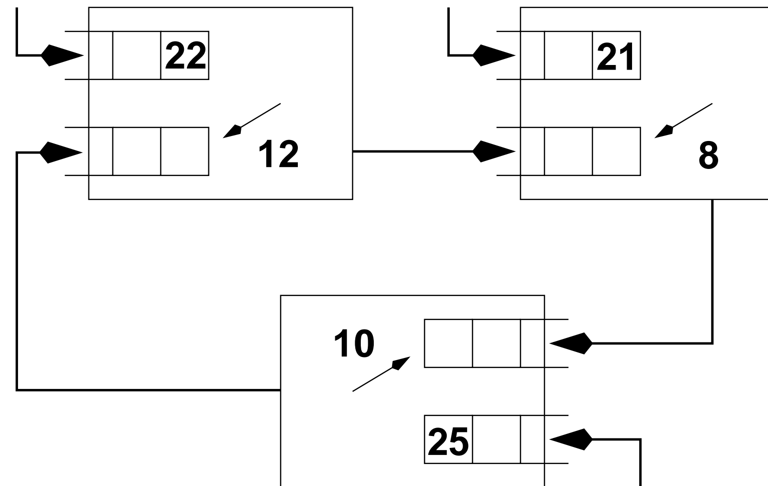



- process 1 & 2 are at local time "t"
- if process 1 sends an event to process 2 at a time *before* process 2's local time there is a *potential* causality problem



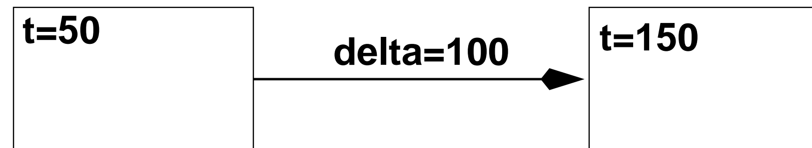

- process queue with smallest message
- if last seen message is smallest then block
- $\Rightarrow$ this ensures causality

# Deadlock



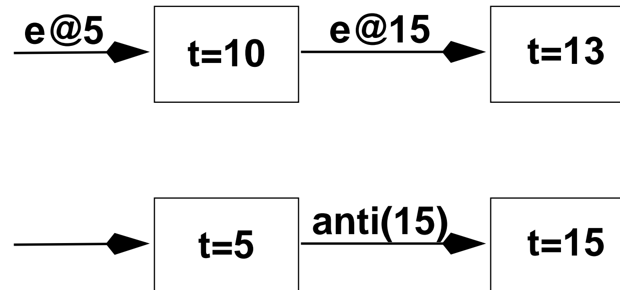

- cycles in simulations are common
- cycles can cause deadlocks in conservative protocol – two solutions:
  1. prevent it – send empty (*null*) message with smallest stamp
  2. detect it – when things stop simulation and give it a kick

# Time Windows




- using knowledge of processing time a “time window” can be determined
- events outside of the time window can be ignored as they must incur processing time that would put their local time past the global time
- i.e. constant “delta”
- **Lookahead** – a variable time window; can send lower bound null messages with local time + pre-computed processing time to the next process
- i.e. variable “delta”

# Time Warp




- **Time Warp Simulation** – each process advances as quickly as it can and non-causal events (*stragglers*) cause the system to redo part of the simulation
- **Roll Back** – processes of undoing the simulation which happens after the receipt of a straggler
- **Global Virtual Time** – the smallest time on an unprocessed event in the system; the point in the simulation guaranteed not to be rolled-back past
- **Anti-Message** – a cancelation which is sent out during roll back asking other processes to invalidate a previously sent message
- permanent records (print to screen or file) can only be output after GVT has passed

# Optimizations

- **Lazy Cancelation** – only send an anti-message if the recomputed time does not generate that message
  - extra overhead from comparison of messages (performance hit?)
  - no extra storage as must store messages in case of anti-message
  - longer potential propagation of bad messages
  - can be super-linear if different recalc produces same answer
- **Lazy Reevaluation** – store state variables in a vector and check for changes in state vector at roll back
  - if no changes then no recalculation required
  - useful for "read" messages which do not affect state
  - complicated to implement
  - additional storage space required
  - performance hit: comparisons of states

# Comparison

- optimistic potentially exploits more parallelism
- optimistic can cause *thrashing*: three steps forward, two steps back
- conservative relies on look ahead for performance improvement: selection of time window is critical and difficult to predict
- only optimistic supports dynamic process allocation
- optimistic easier to write a generic toolkit for (conservative tends to be more application specific)
- optimistic a lot harder to code & test
- optimistic requires more memory
- conservative is now typically used for well suited niche applications (e.g. circuits)