The CODES Project

Rob Ross, Phil Carns, Kevin Harms, John Jenkins, Misbah Mubarak and Shane Snyder

Argonne National Laboratory

Mathematics and Computer Science

rross@mcs.anl.gov

Chris Carothers, Elsa Gonsiorowski, Justin LaPre, Mark Plagge, Caitlin Ross and Noah Wolfe

Rensselaer Polytechnic Institute

Center for Computational Innovations

chrisc@cs.rpi.edu or ***chris.carothers@gmail.com***

Outline

- Part 1: PDES/ROSS Overview

- Part 2: ROSS Details

Motivation

Why Parallel Discrete-Event Simulation (DES)?

- Large-scale systems are difficult to understand
- Analytical models are often constrained

Parallel DES simulation offers:

- Dramatically shrinks model's execution-time
- Prediction of future "what-if" systems performance
- Potential for real-time decision support
  - Minutes instead of days
  - Analysis can be done right away
- Example models: national air space (NAS), ISP backbone(s), distributed content caches, next generation supercomputer systems.

Ex: Movies over the Internet

- Suppose we want to model 1 million home ISP customers downloading a 2 GB movie

- How long to compute?
  - Assume a nominal 100K ev/sec seq. simulator
  - Assume on avg. each packet takes 8 hops
  - 2GB movies yields 2 trillion 1K data packets.
  - @ 8 hops yields 16+ trillion events

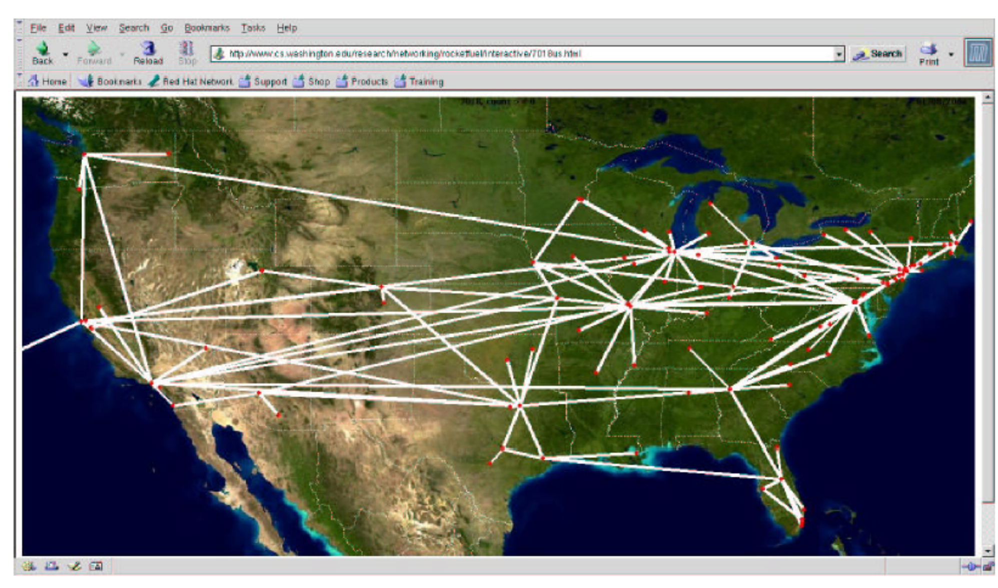


Fig. 5. AT&T Network Topology (AS 7118) from the Rocketfuel data bank for the continental US.

- 16+ trillion events @ 100K ev/sec

Over 1,900 days!!! Or

5+ years!!!

Need massively parallel simulation to make tractable

Discrete Event Simulation (DES)

Discrete event simulation: computer model for a system where changes in the state of the system occur at *discrete* points in simulation time.

Fundamental concepts:

- system state (state variables)
- state transitions (events)

A DES computation can be viewed as a sequence of event computations, with each event computation is assigned a (simulation time) time stamp

Each event computation can

- modify state variables
- schedule new events

DES Computation

example: air traffic at an airport
events: aircraft arrival, landing, departure

arrival 8:00

schedules

landed 8:05

schedules

departure 9:15

arrival 9:30

processed event
current event
unprocessed event

simulation time

- Unprocessed events are stored in a pending list
- Events are processed in time stamp order

Discrete Event Simulation System

model of the physical system

independent of the simulation application

Simulation Application
- state variables
- code modeling system behavior
- I/O and user interface software

calls to schedule events

calls to event handlers

Simulation Executive
- event list management
- managing advances in simulation time

Event-Oriented World View

state variables

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

Simulation application

Event handler procedures

```
Arrival
  Event
{
    …
}

Landed
  Event
{
    …
}

Departure
  Event
{
    …
}
```

Simulation executive

Now = 8:45

Pending Event List (PEL)

9:00
10:10
9:16

Event processing loop

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

Ex: Air traffic at an Airport

Model aircraft arrivals and departures, arrival queueing

Single runway model; ignores departure queueing

- `R` = time runway is used for each landing aircraft (const)
- `G` = time required on the ground before departing (const)

State Variables

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

Model Events

- `Arrival`: denotes aircraft arriving in air space of airport
- `Landed`: denotes aircraft landing
- `Departure`: denotes aircraft leaving

Arrival Events

New aircraft arrives at airport. If the runway is free, it will begin to land. Otherwise, the aircraft must circle, and wait to land.

- R = time runway is used for each landing aircraft
- G = time required on the ground before departing
- 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 @ Now + R;
```

Landed Event

An aircraft has completed its landing.

- R = time runway is used for each landing aircraft
- G = time required on the ground before departing
- Now: current simulation time
- InTheAir: number of aircraft landing or waiting to land
- OnTheGround: number of landed aircraft
- RunwayFree: Boolean, true if runway available

Landed Event:

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

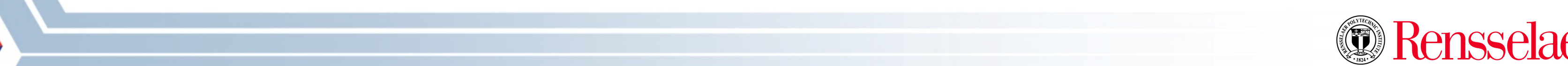
Departure Event

An aircraft now on the ground departs for a new dest.

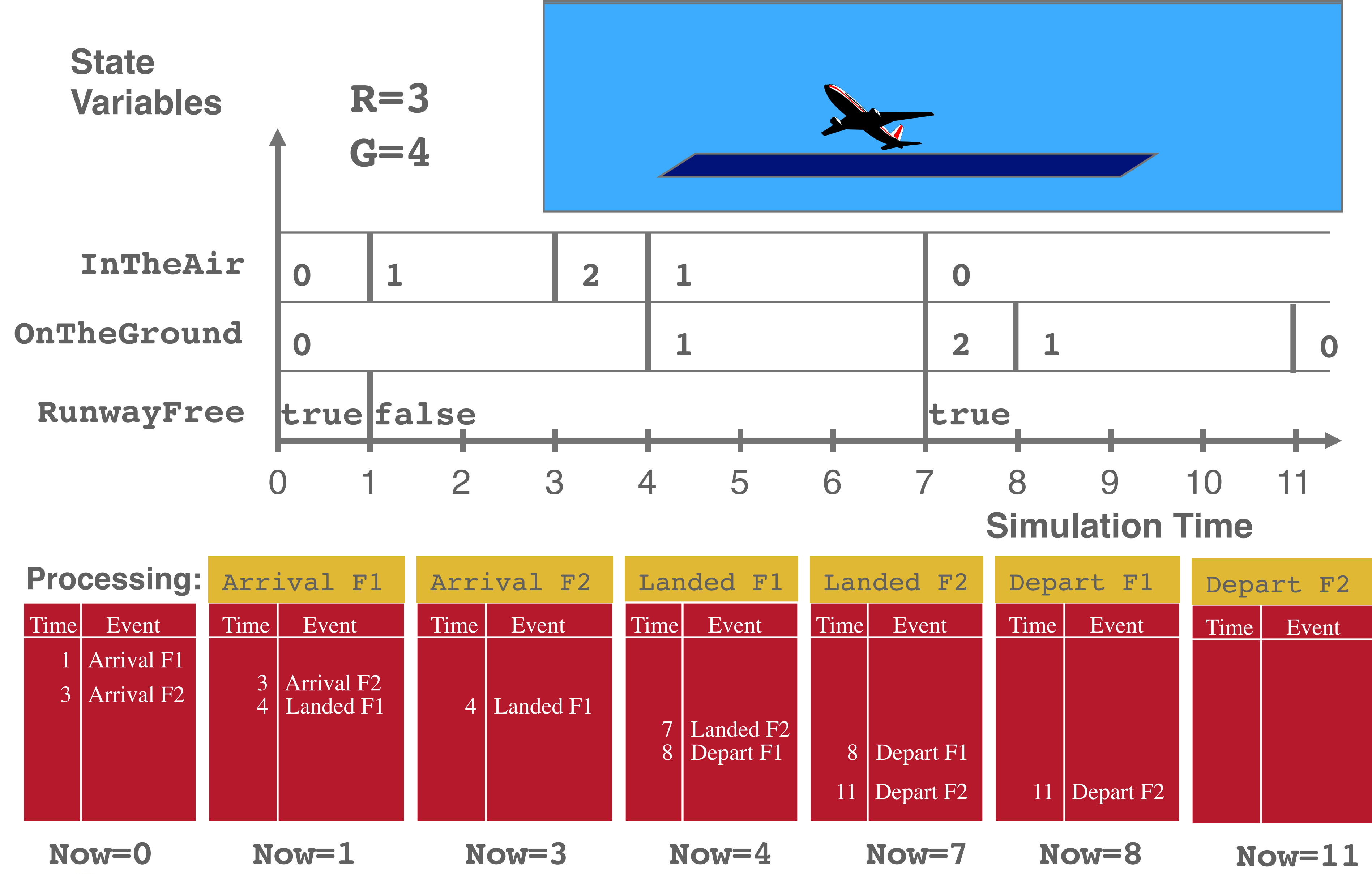
- R = time runway is used for each landing aircraft
- G = time required on the ground before departing
- Now: current simulation time
- InTheAir: number of aircraft landing or waiting to land
- OnTheGround: number of landed aircraft
- RunwayFree: Boolean, true if runway available

Departure Event:

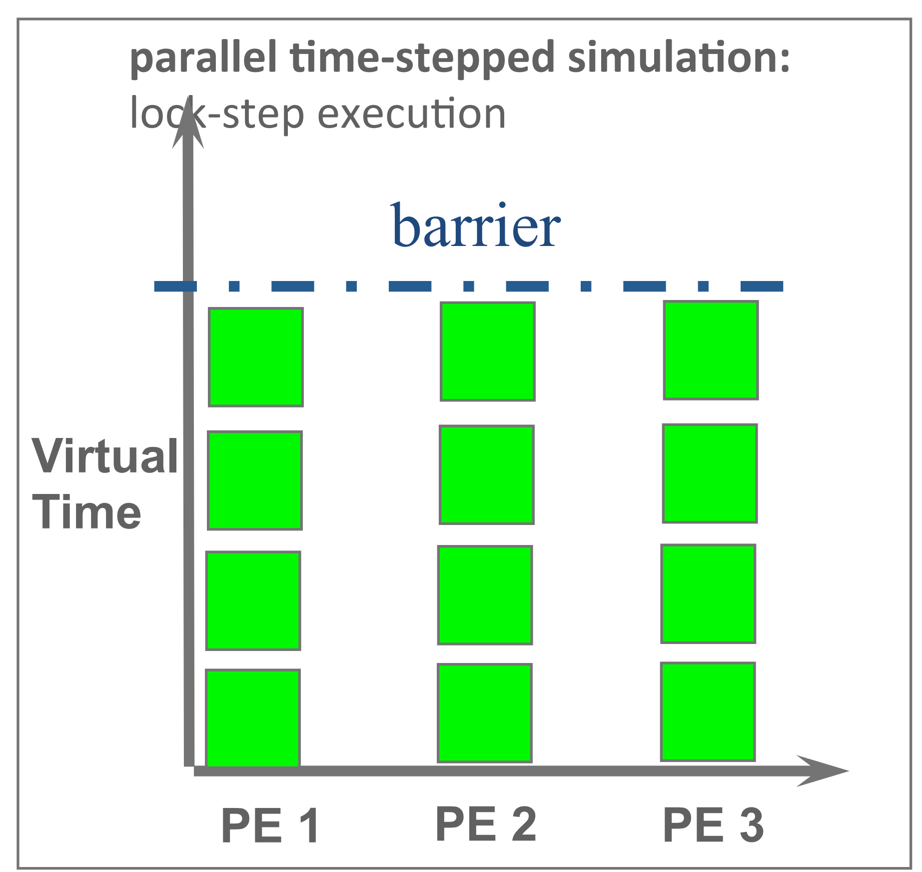
OnTheGround := OnTheGround - 1;

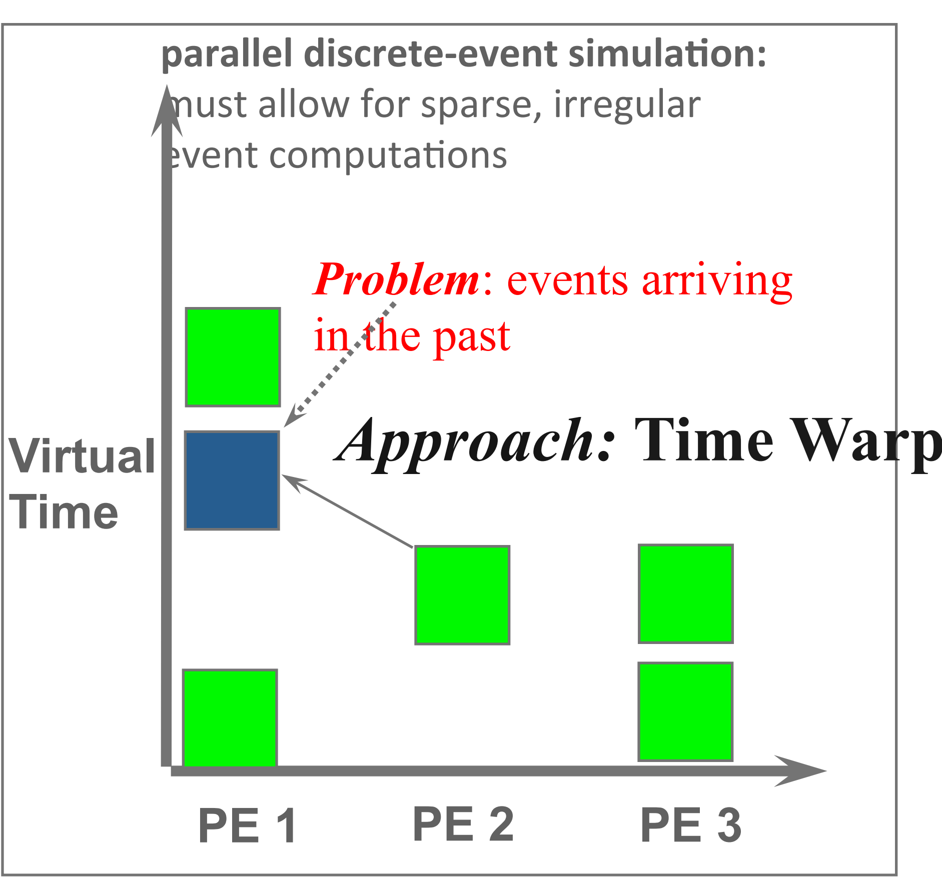


Execution Example

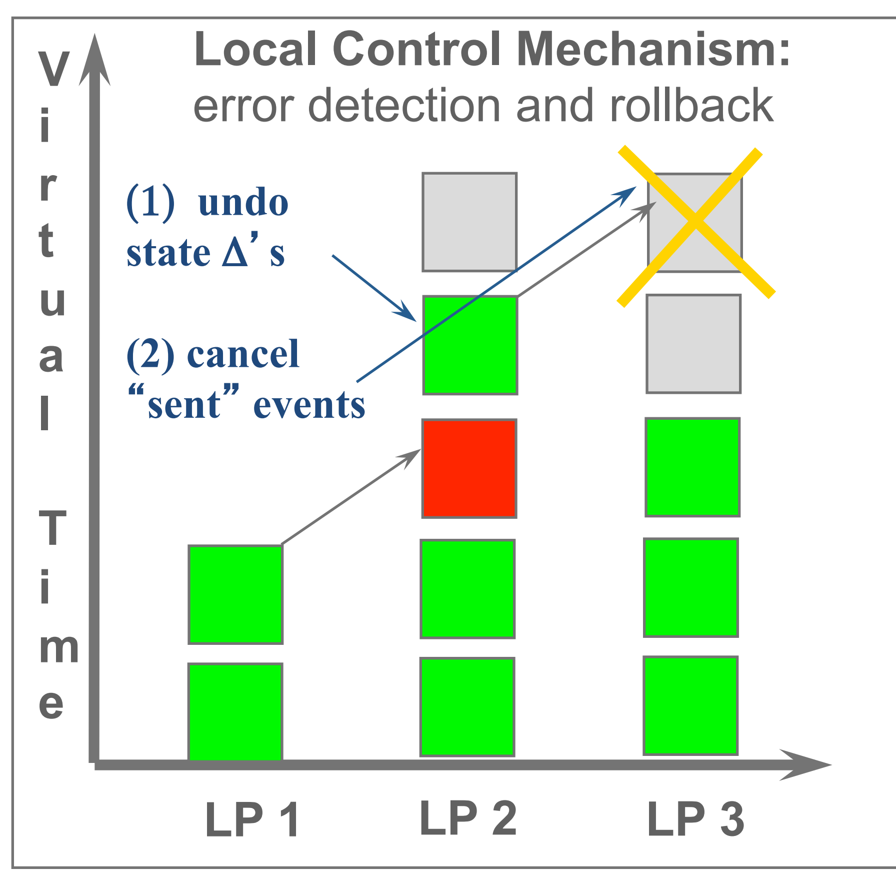


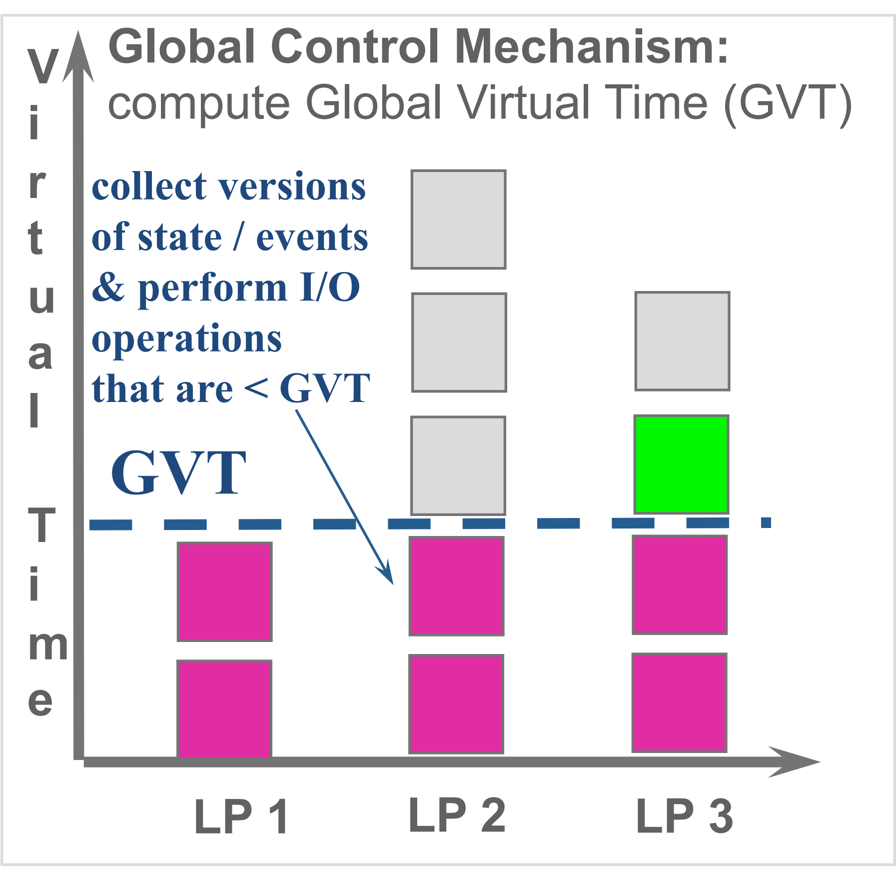
How to Synchronize Parallel Simulations?

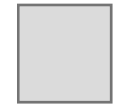




Massively Parallel Discrete-Event Simulation Via Time Warp







unprocessed event

"committed" event

Whew …Time Warp sounds expensive are there other PDES Schemes?…

- "Non-rollback" options:
  - Called "Conservative" because they disallow out of order event execution.
  - Deadlock Avoidance
    - NULL Message Algorithm
  - Deadlock Detection and Recovery

Outline

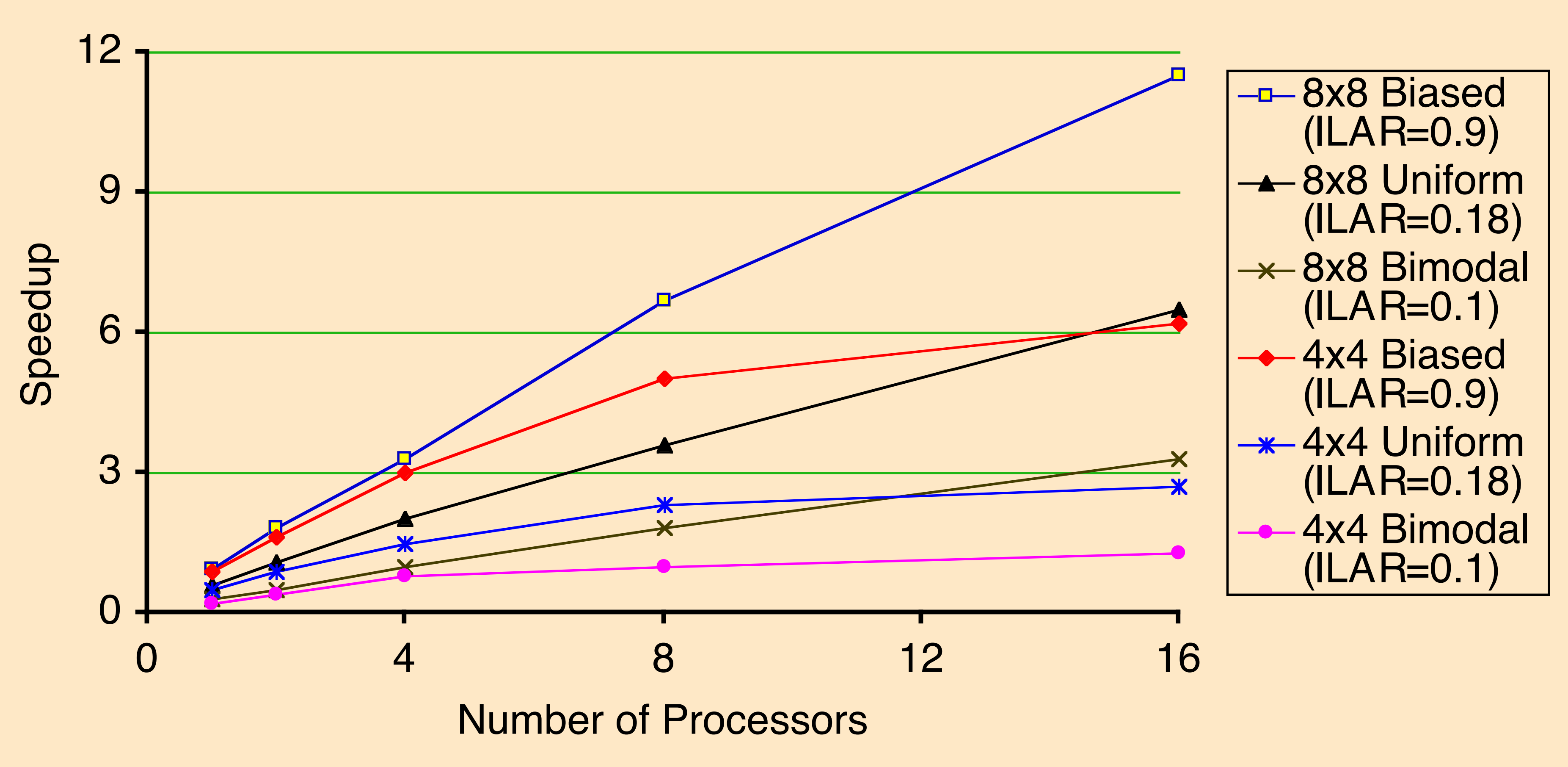
- Part 1: PDES/ROSS Overview

- Part 2: ROSS Details

Null Message Algorithm: Speed Up

- 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!

Deadlock Detection & Recovery

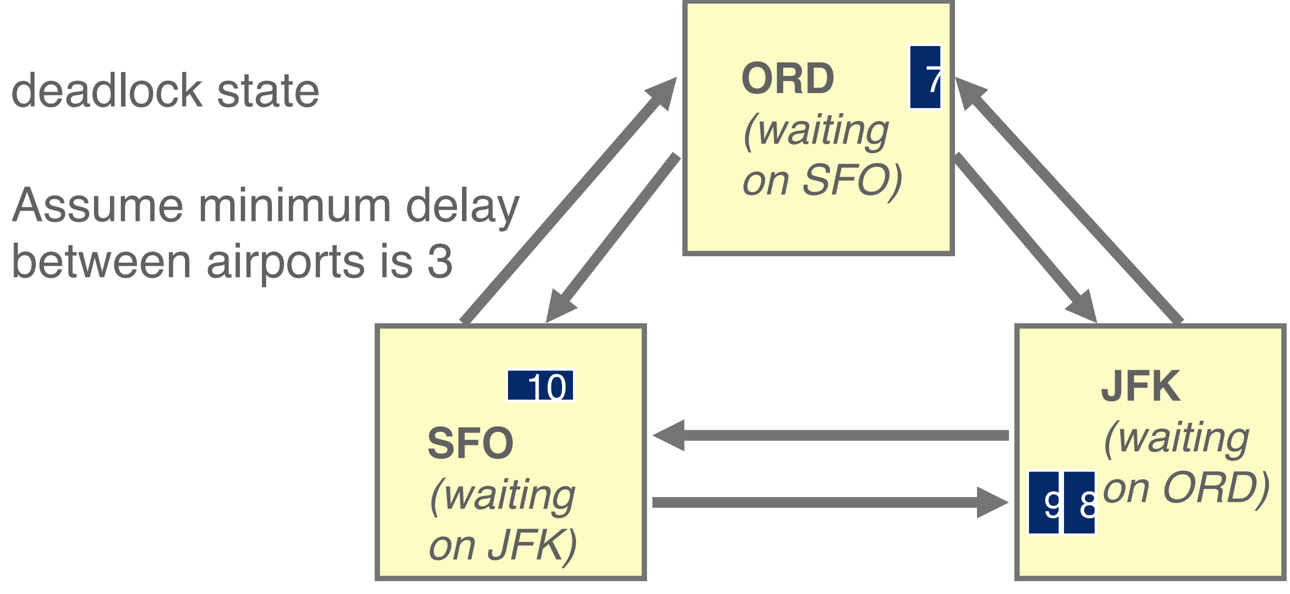
Algorithm A (executed by each LP):
Goal: Ensure events are processed in time stamp order:

```
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
```

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

Deadlock Recovery

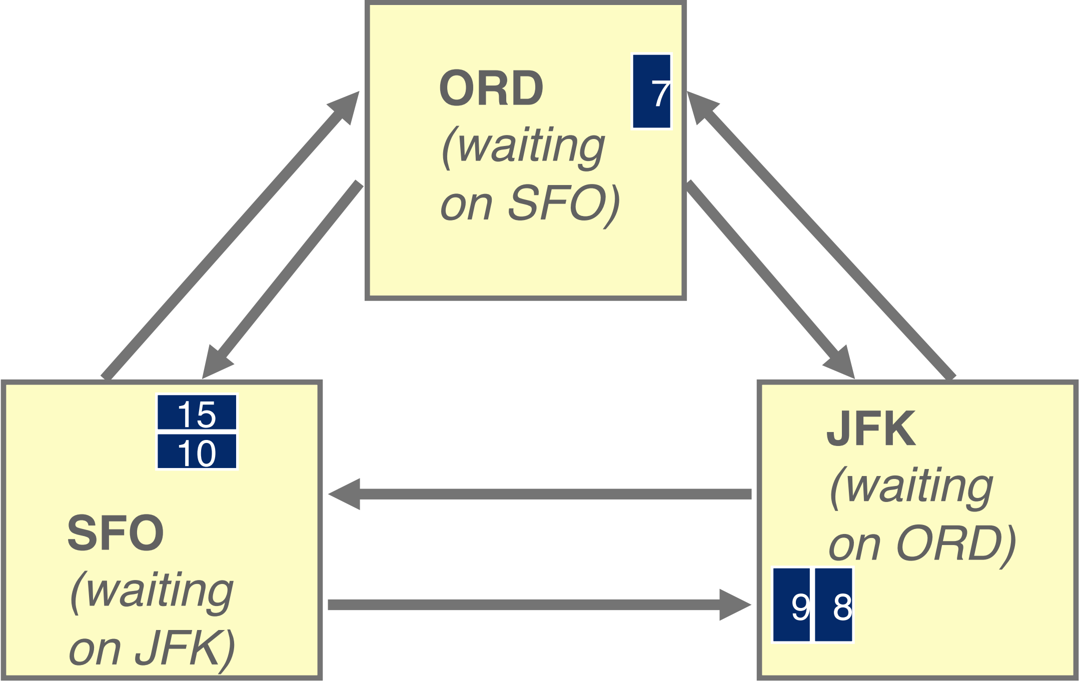
Deadlock recovery: identify "safe" events (events that can be processed w/o violating local causality),



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 lookahead creep!

Preventing LA Creep Using Next Event Time Info



Observation: smallest time stamped event is safe to process

- Lookahead creep avoided by allowing the synchronization algorithm to immediately advance to (global) time of the next event
- Synchronization algorithm must know time stamp of LP's next event
- Each LP guarantees a logical time T such that ***if no additional events are delivered to LP with TS < T, all subsequent messages that LP produces have a time stamp at least T+L (L = lookahead)***

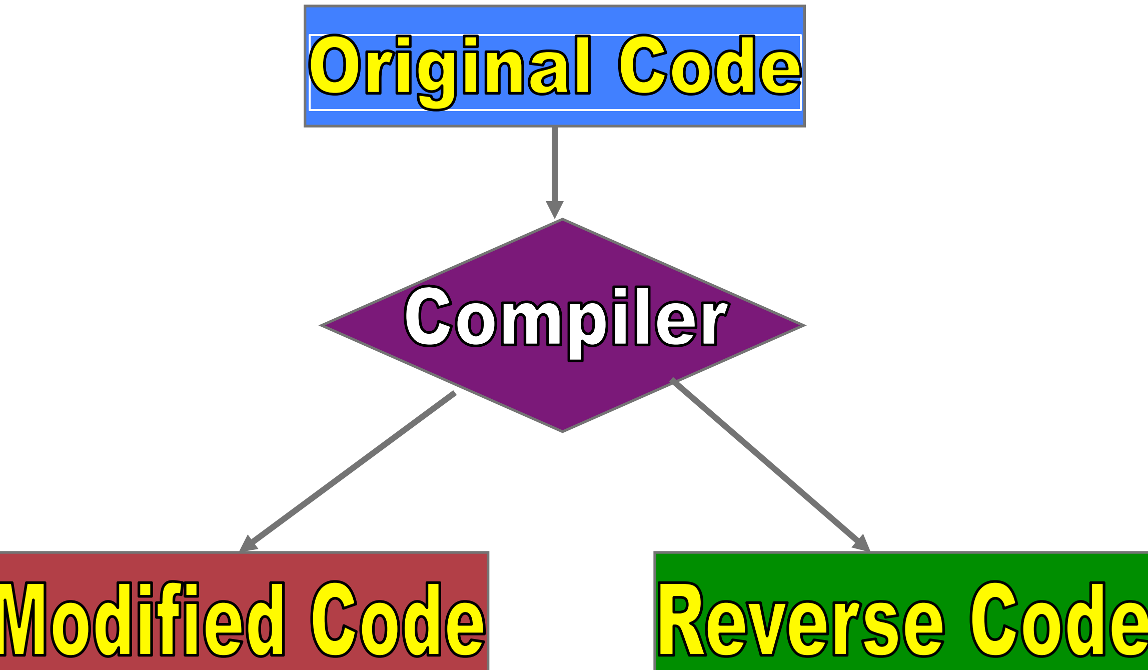
No Free Lunch for PDES!

- **Time Warp** → State saving overheads
- **Null message algorithm** → Lookahead creep problem
  - No zero lookahead cycles allowed
- **Lookahead** → Essential for concurrent processing of events for conservative algorithms
  - Has large effect on performance → need to program it
- **Deadlock Detection and Recovery** → Smallest time stamp event safe to process
  - Others may also be safe (requires additional work to determine this)
- Use time of next event to **avoid lookahead creep**, but hard to compute at scale…

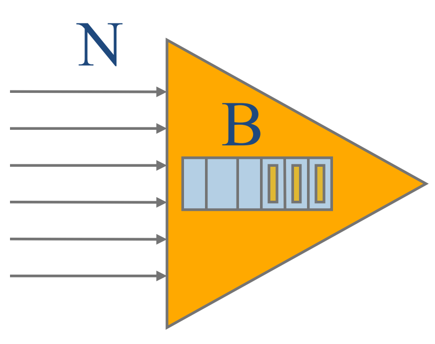
Can we avoid some of these overheads and complexities??

Our Solution: Reverse Computation...

- Use Reverse Computation (RC)
  - automatically generate reverse code from model source
  - undo by executing reverse code
- Delivers better performance
  - negligible overhead for forward computation
  - significantly lower memory utilization



Ex: Simple Network Switch



on packet arrival...

Original

```
if( qlen < B )
    qlen++
    delays[qlen]++
else
    lost++
```

Forward

```
if( qlen < B )
    b1 = 1
    qlen++
    delays[qlen]++
else
    b1 = 0
    lost++
```

Reverse

```
if( b1 == 1 )
    delays[qlen]--
    qlen--
else
    lost--
```

Benefits of Reverse Computation

- State size reduction
  - from B+2 words to 1 word
  - e.g. B=100 => 100x reduction!
- Negligible overhead in forward computation
  - removed from forward computation
  - moved to rollback phase
- Result
  - significant increase in speed
  - significant decrease in memory
- How?...

Beneficial Application Properties

1. Majority of operations are constructive
   – e.g., ++, --, etc.
2. Size of *control state* < size of *data state*
   – e.g., size of `b1` < size of `qlen`, `sent`, `lost`, etc.
3. Perfectly reversible high-level operations gleaned from irreversible smaller operations
   – e.g., random number generation

ROSS Rules for Automation...

Generation rules, and *upper-bounds* on bit requirements for various statement types

Type	Description	Application Code			Bit Requirements		
		Original	Translated	Reverse	Self	Child	Total
T0	simple choice	if() s1	if() {s1; b=1;}	if(b==1){inv(s1);}	1	x1,	1+
		else s2	else {s2; b=0;}	else{inv(s2);}		x2	max(x1,x2)
T1	compound choice	if () s1;	if() {s1; b=1;}	if(b==1) {inv(s1);}	lg(n)	x1,	lg(n) +
	(n-way)	elseif() s2;	elseif() {s2; b=2;}	elseif(b==2) {inv(s2);}		x2,	max(x1....xn)
		elseif() s3;	elseif() {s3; b=3;}	elseif(b==3) {inv(s3);}		....,	
		else() sn;	else {sn; b=n;}	else {inv(sn);}		xn	
T2	fixed iterations (n)	for(n)s;	for(n) s;	for(n) inv(s);	0	x	n*x
T3	variable iterations	while() s;	b=0;	for(b) inv(s);	lg(n)	x	lg(n) +n*x
	(maximum n)		while() {s; b++;}				
T4	function call	foo();	foo();	inv(foo)();	0	x	x
T5	constructive	v@ = w;	v@ = w;	v = @w;	0	0	0
	assignment						
T6	k-byte destructive	v = w;	{b =v; v = w;}	v = b;	8k	0	8k
	assignment						
T7	sequence	s1;	s1;	inv(sn);	0	x1+	x1+…+xn
		s2;	s2;	inv(s2);		....+	
		sn;	sn;	inv(s1);		xn	
T8	Nesting of T0-T7	Recursively apply the above			Recursively apply the above		

Rensselaer

Destructive Assignment...

- Destructive assignment (DA):
  - examples: `x = y;`
    `x %= y;`
  - requires all modified bytes to be saved
- Caveat:
  - reversing technique for DA's can degenerate to traditional *incremental state saving*
- ***Good news:***
  - certain collections of DA's are perfectly reversible!
  - queueing network models contain collections of easily/perfectly reversible DA's
    - queue handling (swap, shift, tree insert/delete, …)
    - statistics collection (increment, decrement, …)
    - ***random number generation (reversible RNGs)***

Reversing an RNG?

```
double RNGGenVal(Generator g)
{
   long k,s;
   double u;
   u = 0.0;

   s = Cg [0][g];  k = s / 46693;
   s = 45991 * (s - k * 46693) - k * 25884;
   if (s < 0) s = s + 2147483647;
   Cg [0][g] = s;
   u = u + 4.65661287524579692e-10 * s;

   s = Cg [1][g];  k = s / 10339;
   s = 207707 * (s - k * 10339) - k * 870;
   if (s < 0) s = s + 2147483543;
   Cg [1][g] = s;
   u = u - 4.65661310075985993e-10 * s;
   if (u < 0) u = u + 1.0;

   s = Cg [2][g];  k = s / 15499;
   s = 138556 * (s - k * 15499) - k * 3979;
   if (s < 0.0) s = s + 2147483423;
   Cg [2][g] = s;
   u = u + 4.65661336096842131e-10 * s;
   if (u >= 1.0) u = u - 1.0;

   s = Cg [3][g];  k = s / 43218;
   s = 49689 * (s - k * 43218) - k * 24121;
   if (s < 0) s = s + 2147483323;
   Cg [3][g] = s;
   u = u - 4.65661357780891134e-10 * s;
   if (u < 0) u = u + 1.0;

   return (u);
}
```

Observation: k = s / 46693 is a Destructive Assignment
Result: RC degrades to classic state-saving…can we do better?

ROSS RNGs: A Higher Level View

The previous RNG is based on the following recurrence….

$$x_{i,n} = a_i x_{i,n-1} \text{ mod } m_i$$

where $x_{i,n}$ one of the four seed values in the Nth set, m_i is one the four largest primes less than 2^{31}, and a_i is a ***primitive root of*** m_i.

Now, the above recurrence is in fact ***reversible….***

inverse of a_i ***modulo*** m_i is defined,

$$b_i = a_i^{m_i-2} \text{ mod } m_i$$

Using b_i, we can generate the reverse recurrence as follows:

$$x_{i,n-1} = b_i x_{i,n} \text{ mod } m_i$$

Reverse Code Efficiency…

- ***Property…***
  - Non-reversibility of indvidual steps ***DO NOT*** imply that the computation as a whole is not reversible.
  - ***Can we automatically find this "higher-level" reversibility?***
- ***Other Reversible Structures Include…***
  - Circular shift operation
  - Insertion & deletion operations on trees (i.e., priority queues).

Reverse computation is well-suited for small grain event models!

ROSS Data Structures – MPI rank or Processing Element (PE)

Ev Ev Ev Ev Ev

Processed Event Lists

LP w/ Model State

LP w/ Model State

LP w/ Model State

LP w/ Model State

Kernel Process (KP)

Right Rotation

Left Rotation

Pending events / Splay Tree

Remote event AVL Tree

Ev Ev Ev Ev Ev

Model Free Event List (FEL)

RNG lib

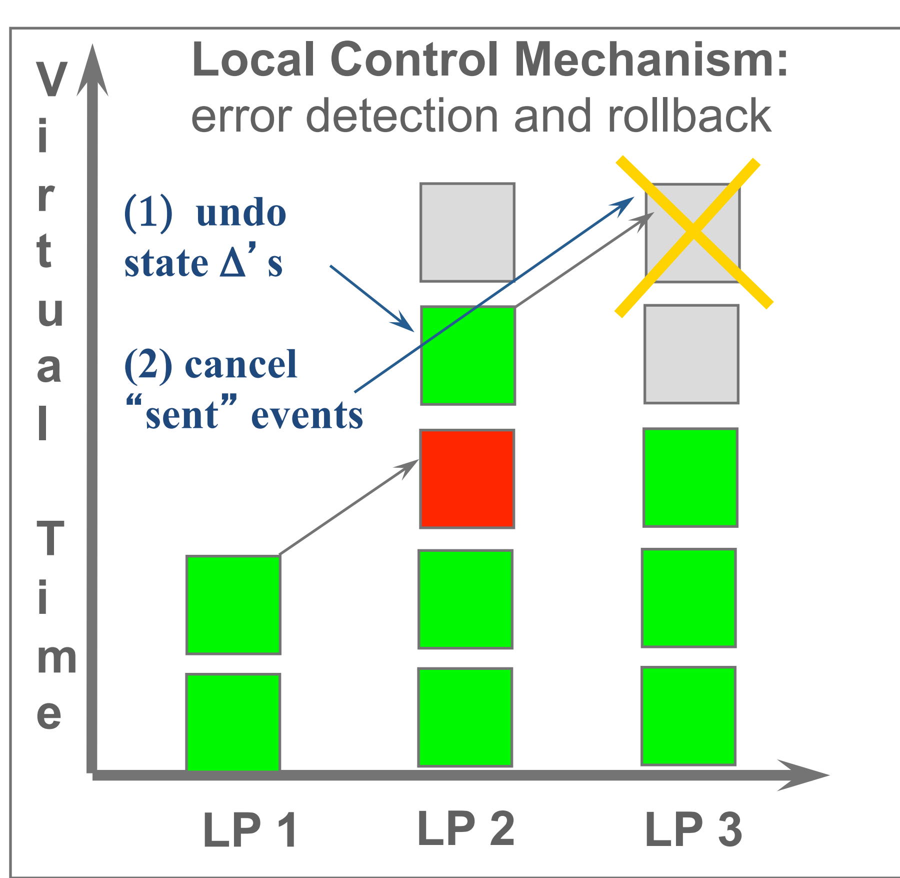
Ev Ev Ev Ev Ev

Network Free Event List (NEL)

MPI

ROSS: Local Control Implementation

- MPI_ISend/MPI_Irecv used to send/recv off core events
- Event & Network memory is managed directly.
  - Pool is allocated @ startup
- Event list keep sorted using a Splay Tree (logN)
- LP-2-Core mapping tables are computed and not stored to avoid the need for large global LP maps.



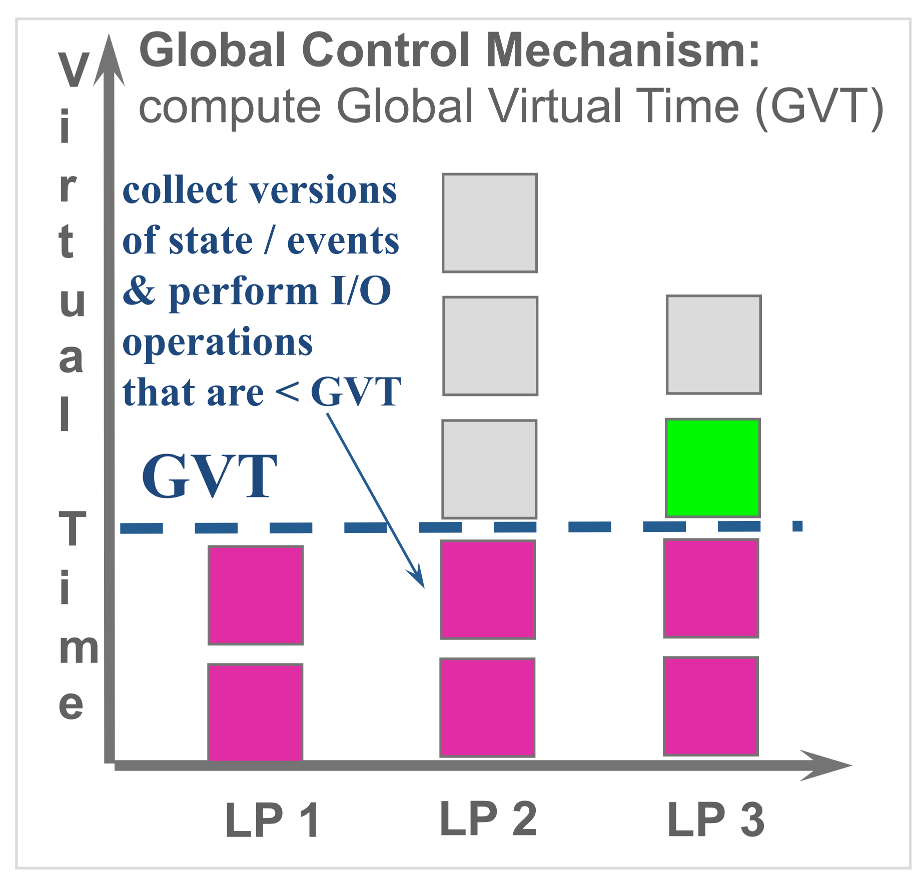
ROSS: Global Control Implementation

GVT (kicks off when memory is low):

1. Each core counts #sent, #recv
2. Recv all pending MPI msgs.
3. MPI_Allreduce Sum on (#sent - #recv)
4. If #sent - #recv != 0 goto 2
5. Compute local core's lower bound time-stamp (LVT).
6. GVT = MPI_Allreduce Min on LVTs

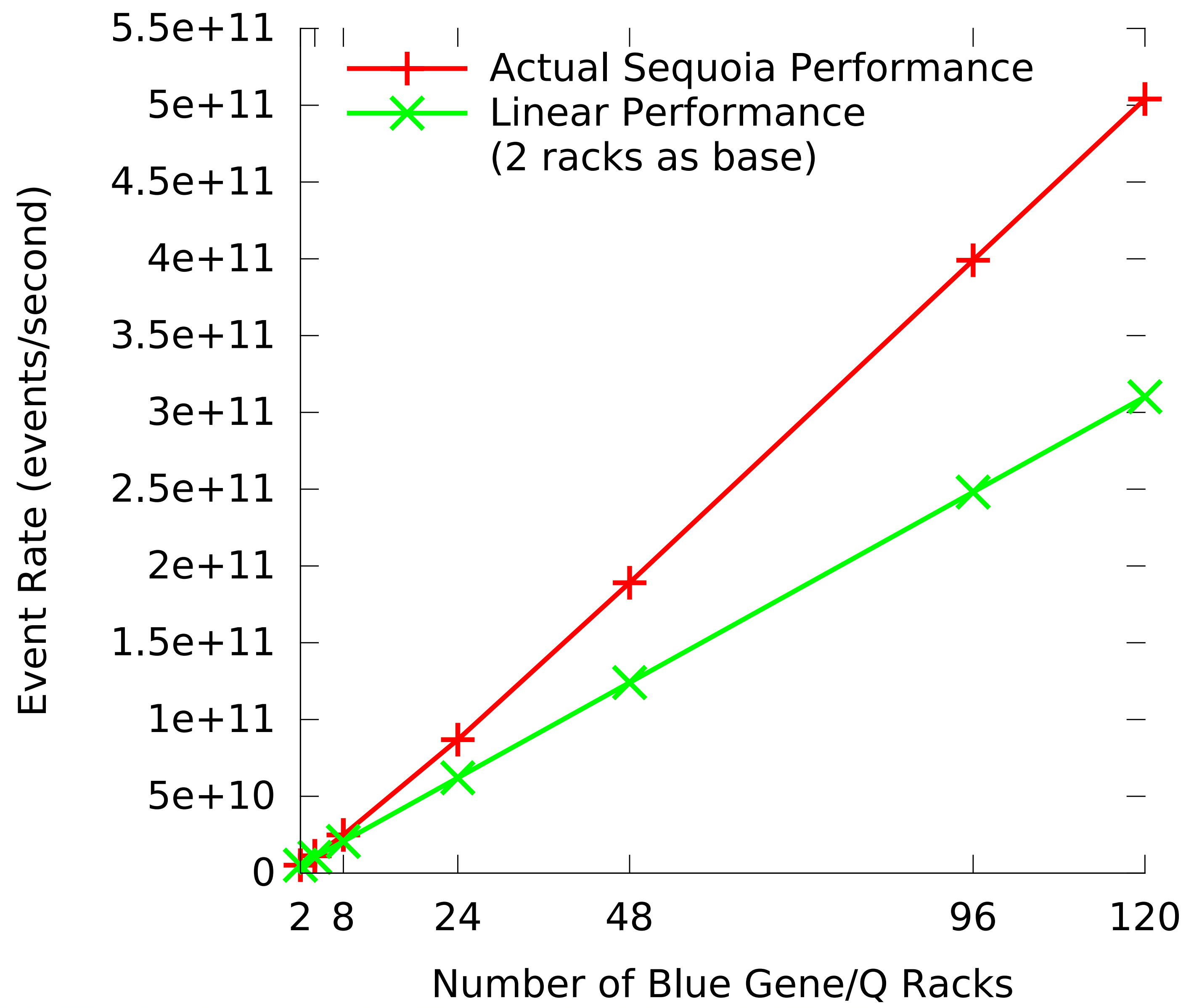
gvt-interval/batch parameters control how frequently GVT is done.

Note, repurposed GVT to implement conservative YAWNS algorithm!



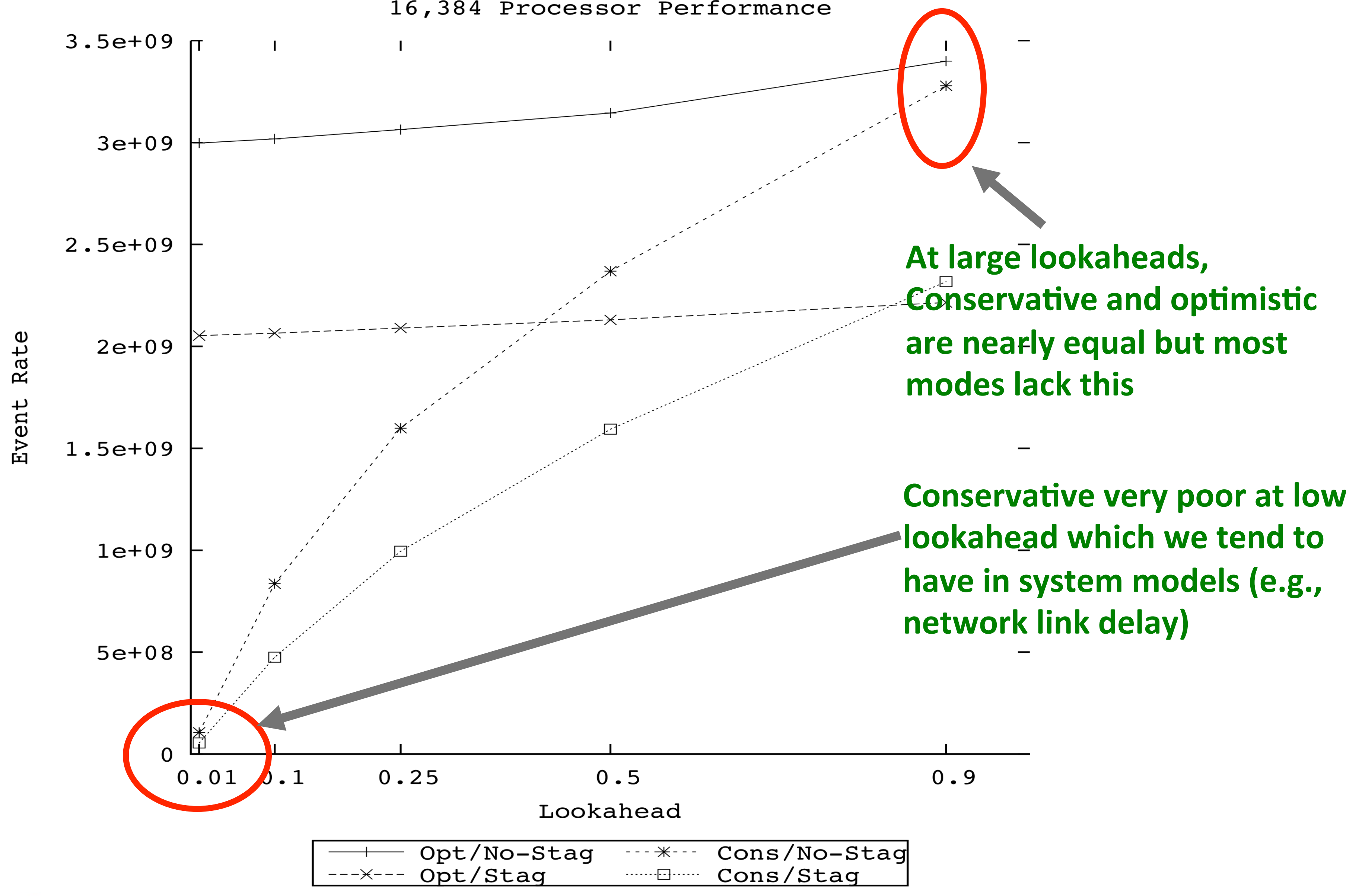
So, how does this translate into ROSS performance on BG/Q?

ROSS Strong Scaling Performance on Sequoia



Rensselaer

ROSS: Conservative/YAWNS vs. Optimistic on BG/L..



ROSS Model Building Steps

- Define LP and event/message data structures
- Define event handlers for initialize, forward, reverse and final processing for each LP type
- Define a custom mapping function for LPs to MPI ranks or use built-in “linear” or “round-robin”
- Bind LPs to KPs in model’s “main”
- Invoke “tw_run” in model’s “main”
- Collect stats directly using MPI collective calls
  - Lots of flexibility here, ROSS does not define an API here

ROSS Command Line Parameters

- **Model:**
  - --nlp=n number of LPs per processor (default 8)
  - --mean=ts exponential distribution mean for timestamps (default 1.00)
  - --mult=ts multiplier for event memory allocation (default 3.00)
  - --lookahead=ts lookahead for events (default 1.00)
  - --start-events=n number of initial messages per LP (default 1)
  - --memory=n additional memory buffers (default 100)
  - --run=str user supplied run name (default undefined)
- **Kernel:**
  - --synch=n Sychronization Protocol: SEQUENTIAL=1, CONSERVATIVE=2, OPTIMISTIC=3, **OPTIMISTIC DEBUG=4** (default 0)
  - --nkp=n number of kernel processes (KPs) per pe (default 1)
  - --end=ts simulation end timestamp (default 100000.00)
  - **--batch=n messages per scheduler block (default 16)**
- **GVT:**
  - **--gvt-interval=n GVT Interval (default 16)**
  - --report-interval=ts percent of runtime to print GVT (default 0.05)
- **Timing:**
  - --clock-rate=ts CPU Clock Rate (default 1000000000.00)
  - --help show this message

ROSS Model Developer Tips & Tricks

- Make sure you model's event population is stable (e.g., event handlers on average don't create/schedule more than 1 event).
- Don't access another LP's state directly → NO SHARED LP STATE!
- Message/event data is read-only, except when using for state-saving
- Use distinct RNG seeds for different actions within an LP to avoid correlations in time-stamps.
  - Note, you can control the number of seed sets per LP.
- Get you model working serial first
- Get your model working YAWNS/conservative next (--synch=2)
- Get your model working optimistically last (--synch=3)
  - Debug using –synch=4 scheduler
- Model is not valid until serial, conservative and optimistic all execute/ commit the same number of events.
- Avoid tie events by adding "random jitter" to event time stamps
- Reduce rollbacks by shrinking "batch" parameter

Acknowledgments

This work was supported by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-06CH11357.