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
Unprocessed events are stored in a pending list
Events are processed in time stamp order
Discrete Event Simulation System

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

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

model of the physical system

independent of the simulation application
Event-Oriented World View

state variables

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

Event handler procedures

<table>
<thead>
<tr>
<th>Event</th>
<th>Arrival Event</th>
<th>Landed Event</th>
<th>Departure Event</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Simulation application

<table>
<thead>
<tr>
<th>Event</th>
<th>Arrival Event</th>
<th>Landed Event</th>
<th>Departure Event</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Simulation executive

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

Pending Event List (PEL)

- 9:00
- 10:10
- 9:16

Now = 8:45
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.

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

Departure Event:
$\text{OnTheGround} := \text{OnTheGround} - 1;$. 
Execution Example

State Variables

<table>
<thead>
<tr>
<th>Time</th>
<th>RunwayFree</th>
<th>OnTheGround</th>
<th>InTheAir</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>true</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>false</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>false</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>true</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>true</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>true</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>true</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>true</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>true</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

R=3
G=4

Processing:

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Arrival F1</td>
</tr>
<tr>
<td>3</td>
<td>Arrival F2</td>
</tr>
<tr>
<td>4</td>
<td>Landed F1</td>
</tr>
<tr>
<td>7</td>
<td>Landed F2</td>
</tr>
<tr>
<td>8</td>
<td>Depart F1</td>
</tr>
<tr>
<td>11</td>
<td>Depart F2</td>
</tr>
</tbody>
</table>

Now=0
Now=1
Now=3
Now=4
Now=7
Now=8
Now=11
How to Synchronize Parallel Simulations?

**parallel time-stepped simulation:**
lock-step execution

**Problem:** events arriving in the past

**Approach:** Time Warp

**parallel discrete-event simulation:**
must allow for sparse, irregular event computations

processed event

“straggler” event
Massively Parallel Discrete-Event Simulation Via Time Warp

Local Control Mechanism: error detection and rollback

1. undo state $\Delta'$ s
2. cancel “sent” events

Virtual Time

Global Control Mechanism: compute Global Virtual Time (GVT)

collect versions of state / events & perform I/O operations that are $\text{< GVT}$

Virtual Time

processed event

unprocessed event

“straggler” 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

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>

- 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!

**deadlock state**

Assume minimum delay between airports is 3
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**

**Original**

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

**Forward**

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

**Reverse**

```c
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 $b_1$ < size of $q\text{len}$, $\text{sent}$, $\text{lost}$, 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

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Application Code</th>
<th>Bit Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Original</td>
<td>Translated</td>
</tr>
<tr>
<td>T0</td>
<td>simple choice</td>
<td>if() s1</td>
<td>if() {s1; b=1;}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else s2</td>
<td>else {s2; b=0;}</td>
</tr>
<tr>
<td>T1</td>
<td>compound choice</td>
<td>if () s1;</td>
<td>if() {s1; b=1;}</td>
</tr>
<tr>
<td></td>
<td>(n-way)</td>
<td>elseif() s2;</td>
<td>elseif() {s2; b=2;}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>elseif() s3;</td>
<td>elseif() {s3; b=3;}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else() sn;</td>
<td>else {sn; b=n;}</td>
</tr>
<tr>
<td>T2</td>
<td>fixed iterations (n)</td>
<td>for(n)s;</td>
<td>for(n) s;</td>
</tr>
<tr>
<td>T3</td>
<td>variable iterations</td>
<td>while() s;</td>
<td>b=0;</td>
</tr>
<tr>
<td></td>
<td>(maximum n)</td>
<td></td>
<td>while() {s; b++;}</td>
</tr>
<tr>
<td>T4</td>
<td>function call</td>
<td>foo();</td>
<td>foo();</td>
</tr>
<tr>
<td>T5</td>
<td>constructive</td>
<td>v@ = w;</td>
<td>v@ = w;</td>
</tr>
<tr>
<td></td>
<td>assignment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T6</td>
<td>k-byte destructive</td>
<td>v = w;</td>
<td>{b =v; v = w;}</td>
</tr>
<tr>
<td></td>
<td>assignment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T7</td>
<td>sequence</td>
<td>s1;</td>
<td>s1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>s2;</td>
<td>s2;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sn;</td>
<td>sn;</td>
</tr>
<tr>
<td>T8</td>
<td>Nesting of T0-T7</td>
<td>Recursively apply the above</td>
<td>Recursively apply the above</td>
</tr>
</tbody>
</table>

*Notes on bit requirements:
- Self: Bit requirements for the original code.
- Child: Bit requirements for translated code.
- Total: Combined bit requirements.*
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 [2][g];  k = s / 15499;
  s = 138556 * (s - k * 15499) - k * 3979;
  if (s < 0) s = s + 2147483423;
  Cg [2][g] = s;
  u = u + 4.6566136096842131e-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 - 4.65661336096842131e-10 * s;
  if (u < 0) u = u + 1.0;

  return (u);
}
ROSS RNGs: A Higher Level View

The previous RNG is based on the following recurrence....

\[ x_{i,n} = a_i x_{i,n-1} \mod m_i \]

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 \textit{primitive root of} \( m_i \).

Now, the above recurrence is in fact \textit{reversible}....

\textit{Inverse of} \( a_i \ \textit{modulo} \ m_i \) is defined,

\[ b_i = a_i^{m_i-2} \mod m_i \]

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

\[ x_{i,n-1} = b_i x_{i,n} \mod m_i \]
Reverse Code Efficiency...

• **Property**...
  – Non-reversibility of individual 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: 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

Event Rate (events/second)

Number of Blue Gene/Q Racks

Actual Sequoia Performance
Linear Performance
(2 racks as base)
ROSS: Conservative/YAWNS vs. Optimistic on BG/L.

At large lookaheads, Conservative and optimistic are nearly equal but most modes lack this.

Conservative very poor at low lookahead which we tend to have in system models (e.g., network link delay).
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=runname` 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.