Design of Shared Memory Pools for Improved Communications in ROSS

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

Rensselaer Polytechnic Institute

Center for Computational Innovations

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

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

Argonne National Laboratory

Mathematics and Computer Science

rross@mcs.anl.gov

Rensselaer CCI Center for Computational Innovations

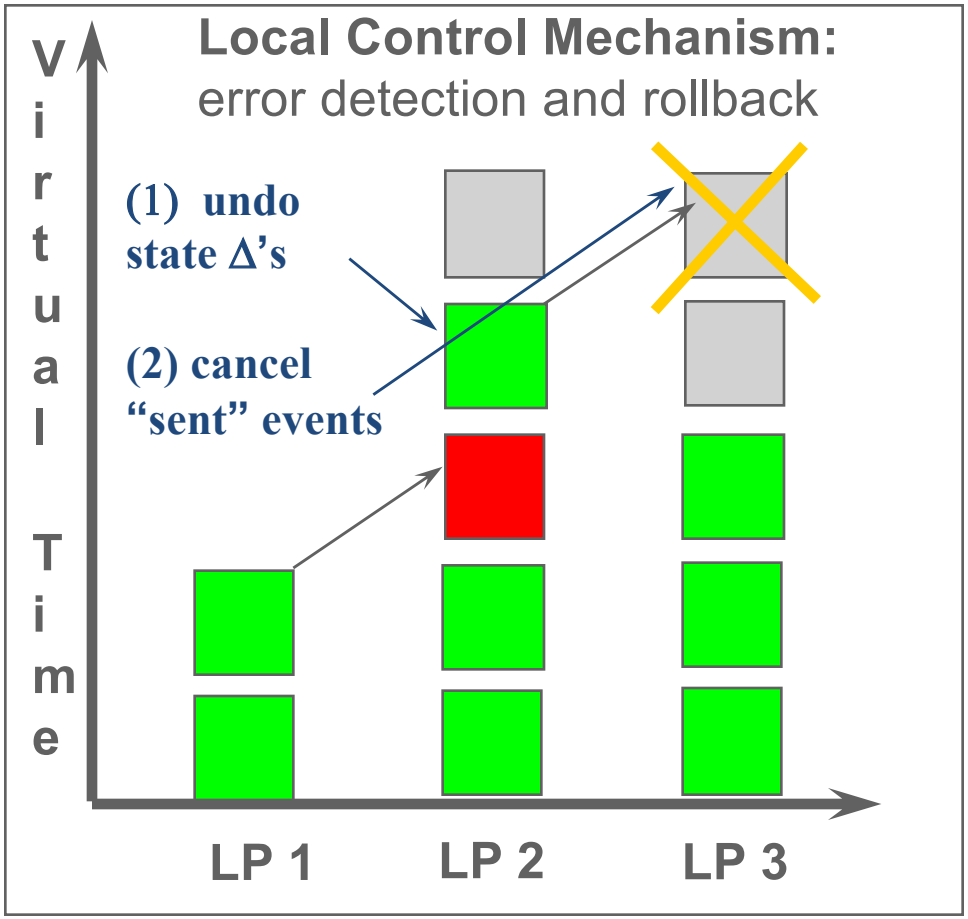
U.S. DEPARTMENT OF ENERGY

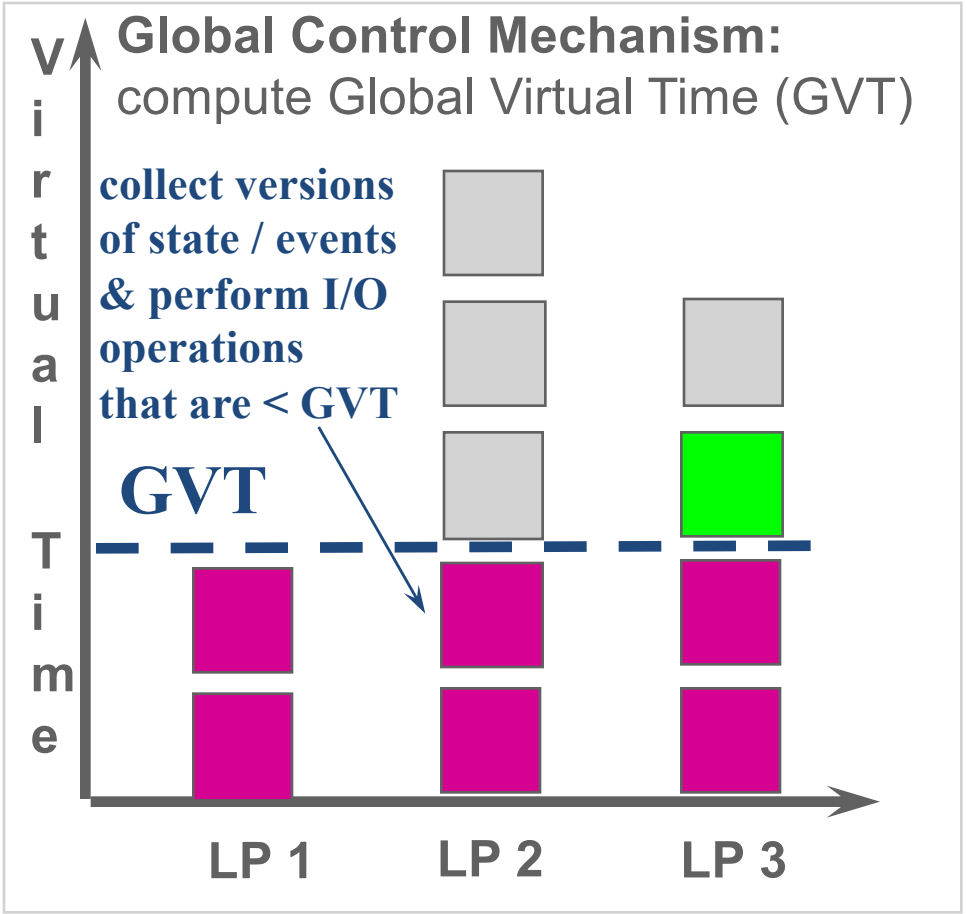
Outline

- ROSS Overview

- Shared Memory Pool Design

Massively Parallel Discrete-Event Simulation Via Time Warp





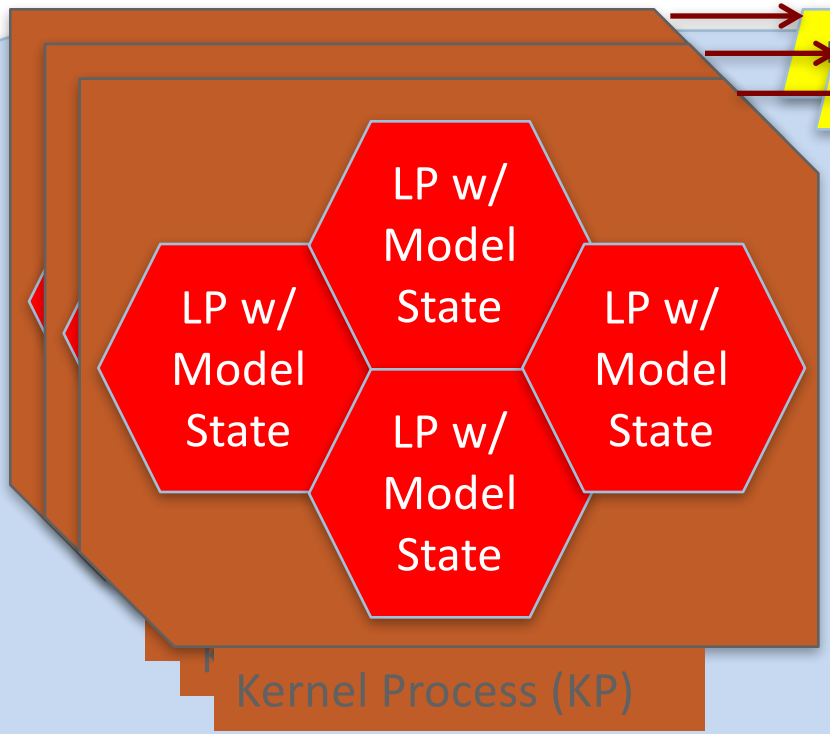
processed event

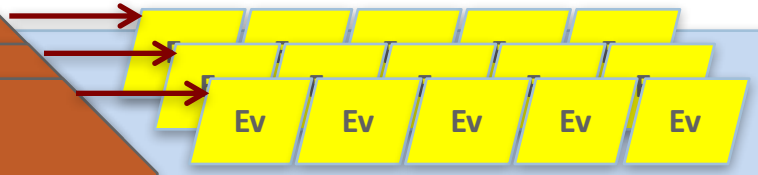
unprocessed event

"straggler" event

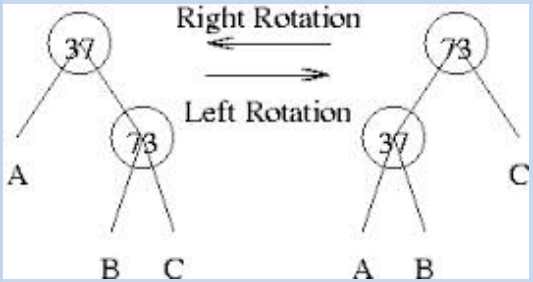
"committed" event

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

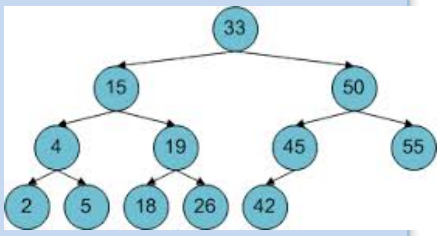




Processed Event Lists

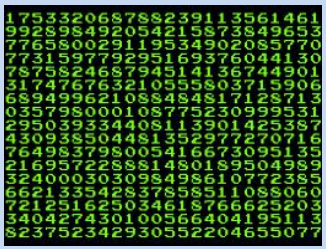


Pending events / Splay Tree



Remote event AVL Tree

Model Free Event List (FEL)



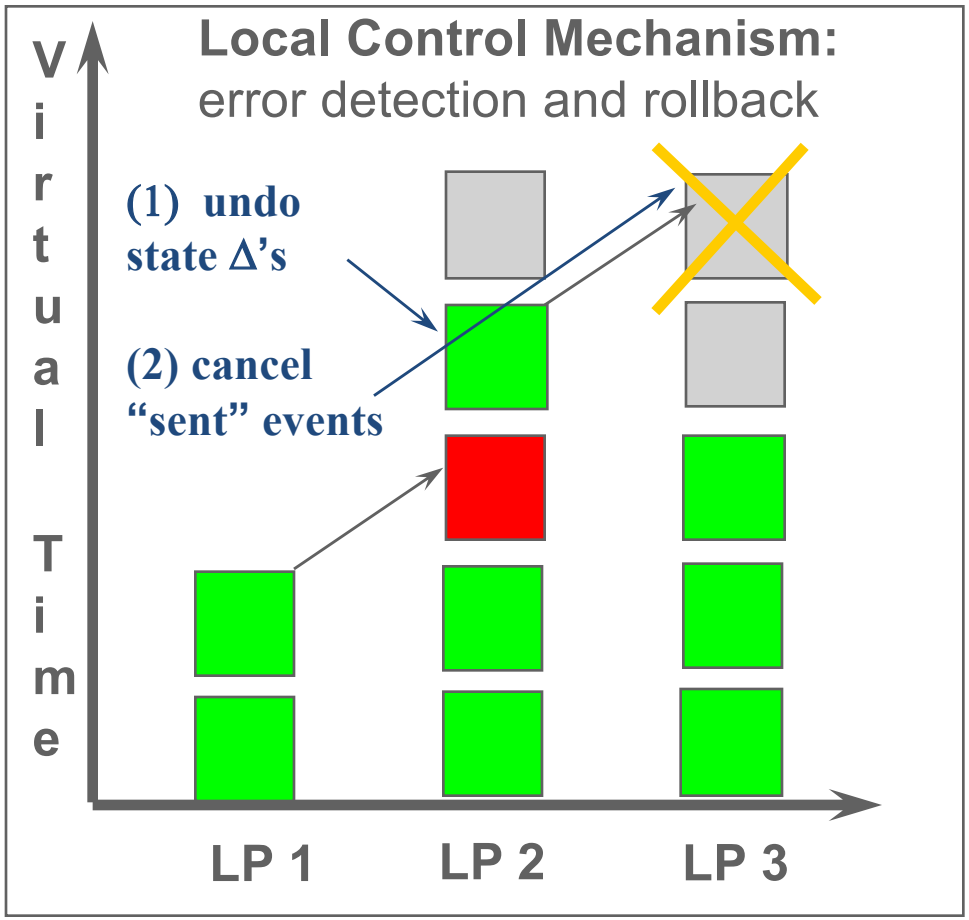
RNG lib

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.
- AVL Tree used to keep track of "remote" event sends to support cancel/rollback operations



ROSS: Global Control Implementation

GVT (kicks off when memory is low):

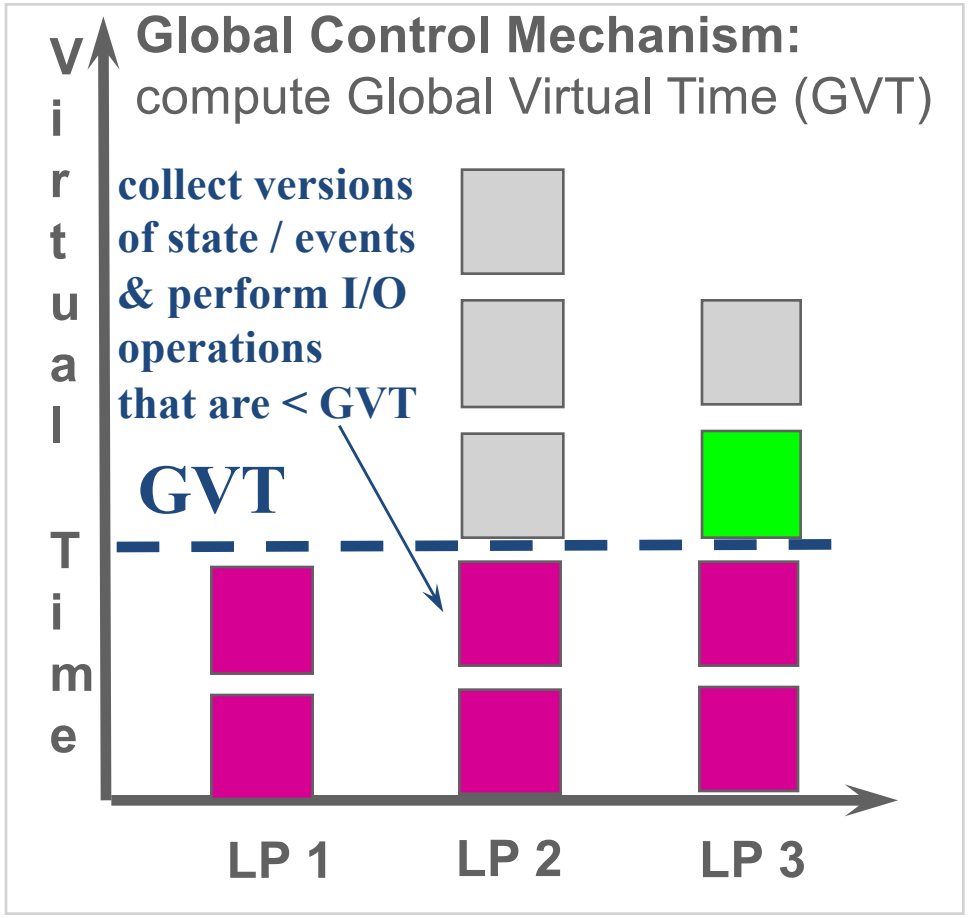
1. Each core counts #sent, #recv
2. Recv all pending MPI msgs.
3. MPI_Allreduce Sum on (#sent - #recv)
4. If #sent - #recv != 0 goto 2
5. Compute local core's lower bound time-stamp (LVT).
6. GVT = MPI_Allreduce Min on LVTs

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

Now have "optimistic realtime" GVT

--sync=5 option

Note, repurposed GVT to implement conservative YAWNS algorithm as well !



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

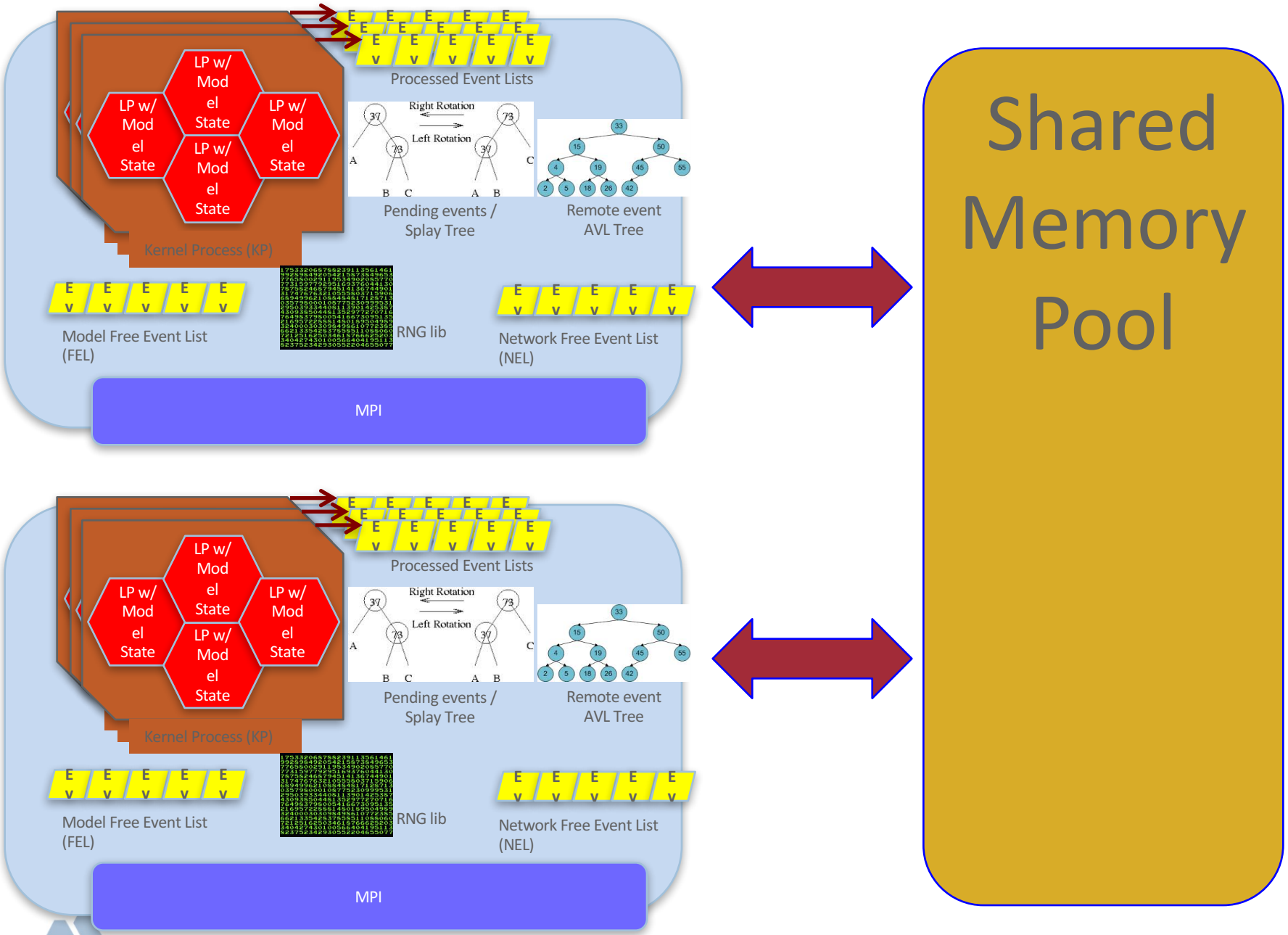
Outline

- ROSS Overview
- Shared Memory Pool Design

Motivation and Design Constraints

- We have observed that for larger remote communication rates, ROSS' performance degrades (potentially significantly) due to much greater MPI overheads.
  - **Know this because original ROSS was written for shared memory**
  - **Pthread ROSS is 2x faster than MPI ROSS on 1 node of BG/Q**
  - **Main optimization is it passes pointers to events as opposed to transferring a full copy of event data via MPI communications**
- So, MPI implementation is leaving a good bit of performance on the table
- A hybrid MPI + Pthreads is natural choice … but..
  - Would need to encapsulate the global ROSS state into per-thread state
  - Moving to a fully global shared memory space w/i a node will break all of CODES
  - Allowing pthreads to invoke MPI operations creates new overheads
  - Global shared address space introduces potential for "false sharing" among threads
- A better choice is to leverage MPI shared memory buffer API.
  - `MPI_Win_allocate_shared` provides a shared buffer to MPI rank w/I the same compute node
  - Built on SystemV shared memory using `shmget/shmat`

Design Overview



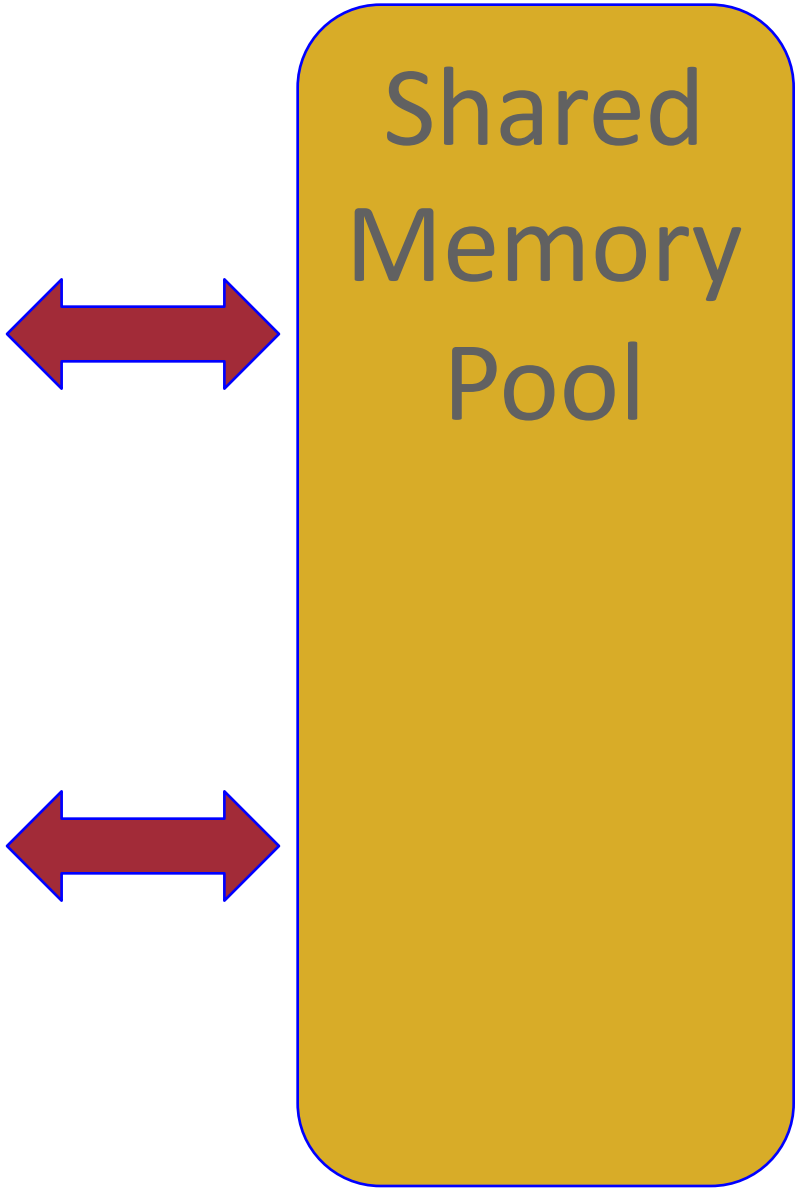
Problem with MPI SHM API

Shared memory buffer pool's address differs across MPI ranks

```
llnl-year-3-work.txt          slurm-54359.out          slurm-54914.out
chrisc@area52:~/PROJECTS/ROSS$ mpirun -np 2 ./mpi3shm_1Dring.v2.1
i'm rank 0 with 2 intranode partners, 1 (1), 1 (1)
i'm rank 1 with 2 intranode partners, 0 (0), 0 (0)
Rank 0: SHM pool starts at address 0x7efffe910000
Rank 0: partner rank 1 has partner base ptrs of 0x7f003e910000
Rank 0: partner rank 1 has partner base ptrs of 0x7f003e910000
Rank 0: write memory location 17 with 17 value
Rank 1: SHM pool starts at address 0x7fb904feb000
Rank 1: partner rank 0 has partner base ptrs of 0x7fb8c4feb000
Rank 1: partner rank 0 has partner base ptrs of 0x7fb8c4feb000
Rank 1: write memory location 17 with 0 value
load MPI/SHM values from neighbour: rank 1, numtasks 2 on area52
load MPI/SHM values from neighbour: rank 1, numtasks 2 on area52
Rank 1: write memory location 17 with 17 value
load MPI/SHM values from neighbour: rank 0, numtasks 2 on area52
load MPI/SHM values from neighbour: rank 0, numtasks 2 on area52
chrisc@area52:~/PROJECTS/ROSS$
```

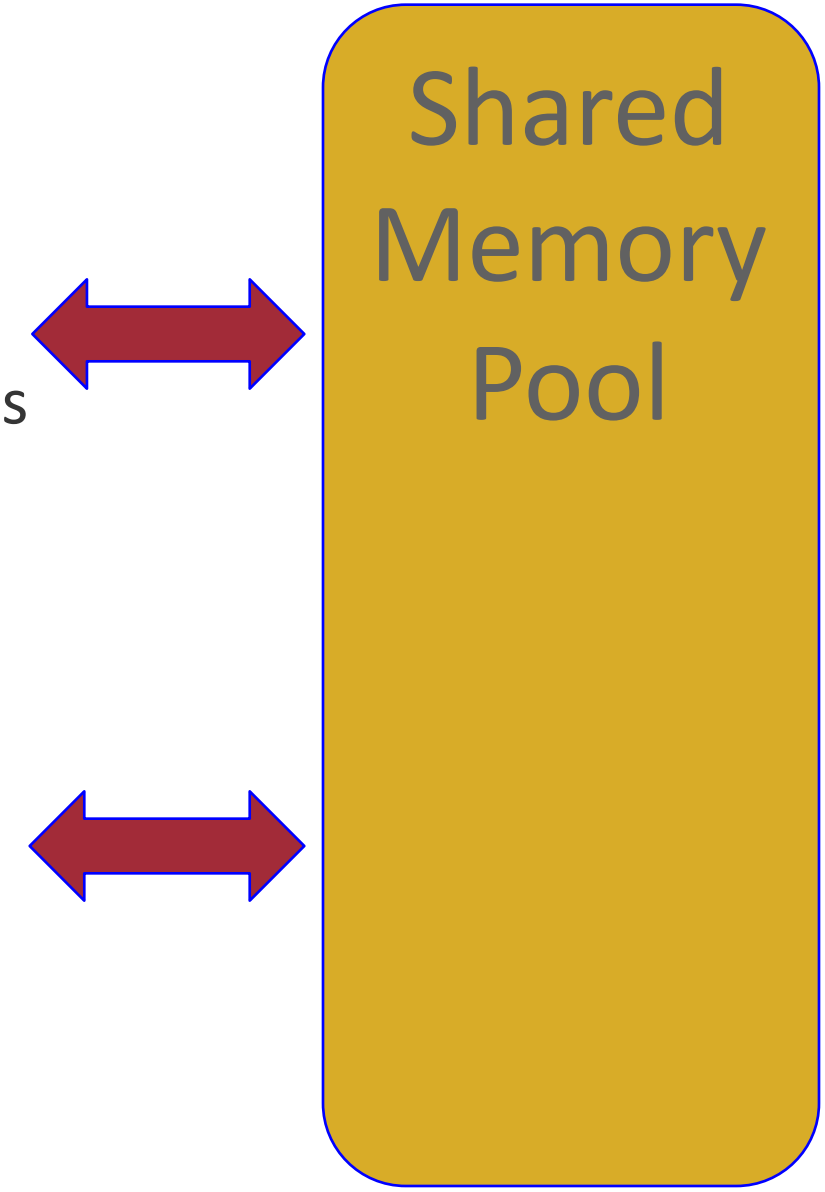
MPI SHM API - can't store address pointers (yet!)

- Need to store "shared" free event lists in buffer
- Need to perform operations like:
  - `event->next = ListHead`
  - `ListHead = event`
- Current API requires translating pointer code to "offset + base"
  - `Buffer[event].next = ListHead`
  - `ListHead = event`
- Greatly complicates pointer-based code
  - Performance loss of ~2x
- Greater potential for bugs with "offset + base" code
  - +++ lifetime employment! ☺
  - --- if grad student, you'll never finish your thesis



Solution: "Force" all MPI ranks to attach shared memory pool at same virtual address

- Current need to abandon the MPI SHM API at this point …
- Create shared memory pool using "`shmget`" system call.
- Find a common available virtual address space using "`mmap`" system call across ranks on same compute nodes
- Use "`shmat`" to attach to created memory pool at common address previous determined
- * Special thanks to Kamil Iskra @ ANL for this approach



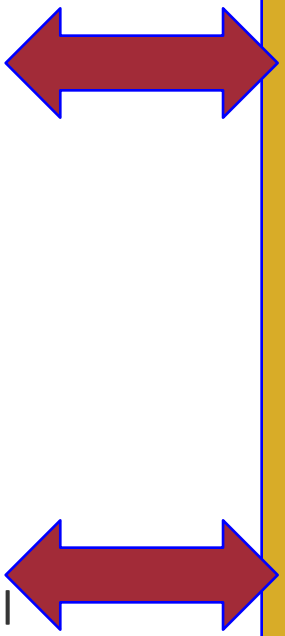
Structure & Function of Shared Memory Pool

- Now, that we have pointers, we can have a shared memory pool that resembles the shared memory approaches used previously.
- Free list contains events that can be shared by "sender" rank among N-1 other ranks on the same compute node
- Send event results sender allocating an event from their pool and inserting into desk ranks "`eventq`"
- Receiver rank directly uses shared events and places event into Splay Tree once any local rollback processing is complete
- Direct event cancellation supported on shared events. Here, pointer to original event is threaded into "`cancelq`" for rollback processing by receiver rank

```
struct tw_shared_pool
 {
   tw_eventq free_list;
   pthread_mutex fl_lck;
   tw_eventq eventq;
   pthread_mutex eq_lck;
   tw_eventq cancelq
   pthread_mutex
can_lck;
 };

struct tw_shared_pool
pool[0…N-1]

Rest of memory is used
to populate free lists
of each Rank's shared
pool
```

Buffer Return and Better GVT

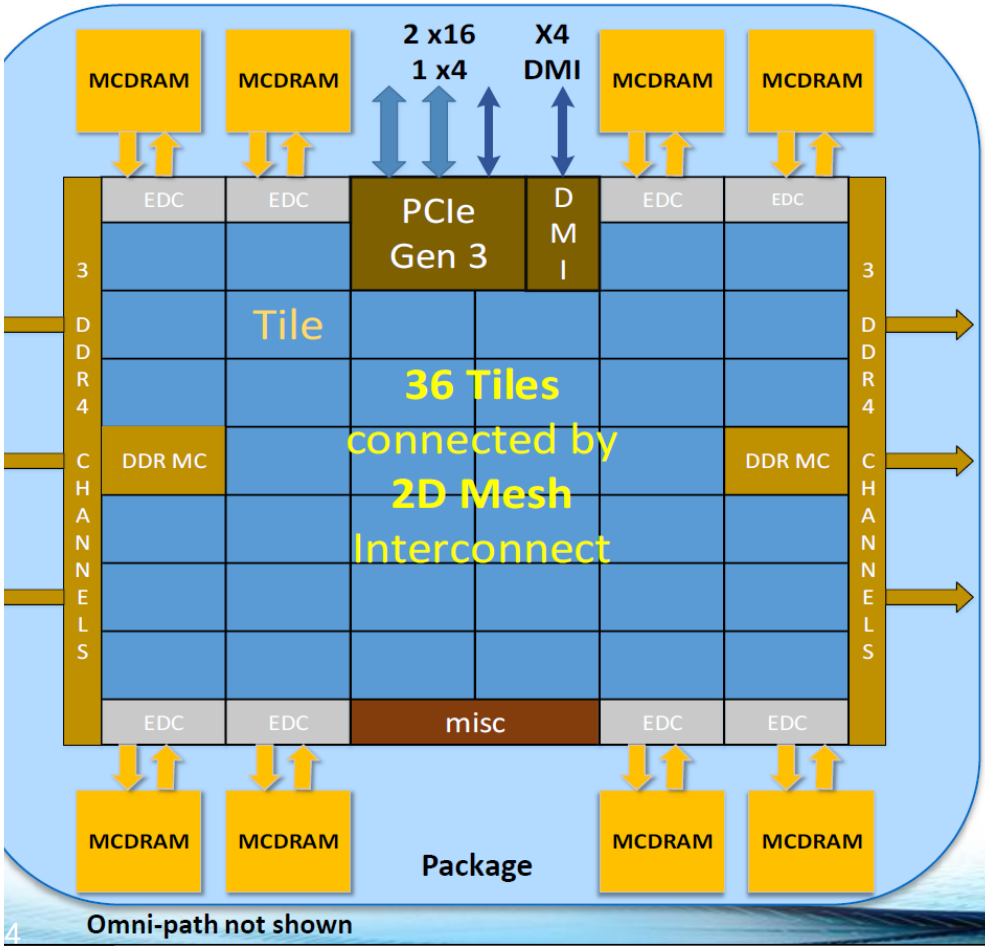
- **Buffer Return:** when GVT frees a shared event for reuse, it needs to be return to the sender.
  - Acquire “fl_lck”
  - Insert on sender’s “free_list”
  - Release “fl_lck”
  - Need to agument the event structure with sender information
- **Leverage Fujimoto’s Shared Memory GVT for LVT w/i a compute node**
  - Pull out all network MPI events
  - Sets a “flag” that all ranks w/i a compute node can “see”
  - No events lost in shared memory pool
  - Each ranks computes own LVT
  - Min of all LVTs is LVT for compute node.
  - Use Allreduce approach but only 1 rank from each compute need needs to particpate

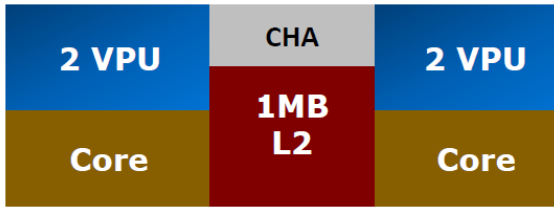


```
struct tw_shared_pool
 {
   tw_eventq free_list;
   pthread_mutex fl_lck;
   tw_eventq eventq;
   pthread_mutex eq_lck;
   tw_eventq cancelq
   pthread_mutex
can_lck;
 };

struct tw_shared_pool
pool[0…N-1]

Rest of memory is used
to populate free lists
of each Rank’s shared
pool
```

Preparing of Many-Core Architectures

Knights Landing Overview





Chip: 36 Tiles interconnected by **2D Mesh**
Tile: 2 Cores + 2 VPU/core + 1 MB L2

Memory: MCDRAM: 16 GB on-package; High BW
DDR4: 6 channels @ 2400 up to 384GB
IO: 36 lanes PCIe Gen3. 4 lanes of DMI for chipset
Node: 1-Socket only
Fabric: Omni-Path on-package (not shown)

Vector Peak Perf: 3+TF DP and 6+TF SP Flops
Scalar Perf: ~3x over Knights Corner
Streams Triad (GB/s): MCDRAM : 400+; DDR: 90+

Source Intel: All products, computer systems, dates and figures specified are preliminary based on current expectations, a are subject to change without notice. KNL data are preliminary based on current expectations and are subject to change without notice. 1Binary Compatible with Intel Xeon processors using Haswell Instruction Set (except TSX). 2Bandwidth numbers are based on STREAM-like memory access pattern when MCDRAM used as flat memory. Results have been estimated based on internal Intel analysis and are provided for informational purposes only. Any difference in system hardware or software design or configuration may affect actual performance.

How many sharing groups will be needed for optimal performance ?

Status

- Branch in GITHub is started
- Cmake build scripts have been modifed to select shared pool functionality
- Coding has begun:
  - Need to verify shget/mmap test and shmat path creates a common shared memory pool across all MPI ranks.
  - Determine which MPI ranks are co-located on a common compute node.
- Initial functionality by year end.

Thank You & 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.