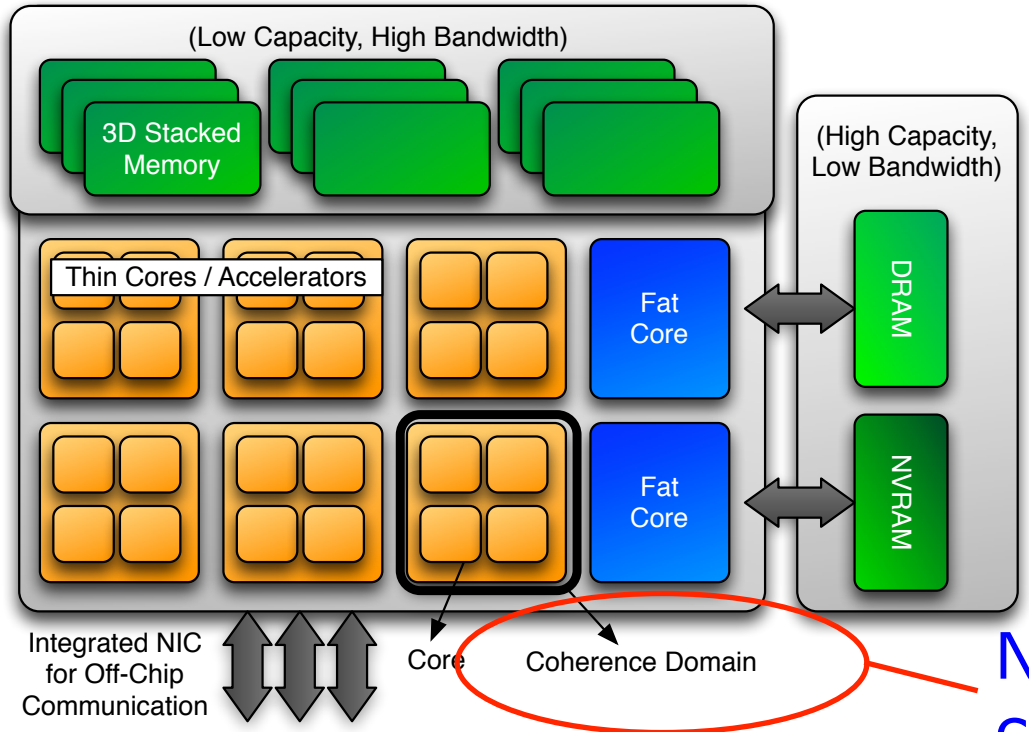
MPI + MPI: Using MPI-3 Shared Memory As a Multicore Programming System

William Gropp
www.cs.illinois.edu/~wgropp

Likely Exascale Architectures



Note: not fully cache coherent

Figure 2.1: Abstract Machine Model of an exascale Node Architecture

- From "Abstract Machine Models and Proxy Architectures for Exascale Computing Rev 1.1," J Ang et al

Applications Still MPI-Everywhere

- Benefit of programmer-managed locality
  - ♦ Memory performance nearly stagnant
  - ♦ Parallelism for performance implies locality must be managed effectively
- Benefit of a single programming system
  - ♦ Often stated as desirable but with little evidence
  - ♦ Common to mix Fortran, C, Python, etc.
  - ♦ But…Interface between systems must work well, and often don't
    - E.g., for MPI+OpenMP, who manages the cores and how is that negotiated?

Why Do Anything Else?

- Performance
  - ♦ May avoid memory (though probably not cache) copies
- Easier load balance
  - ♦ Shift work among cores with shared memory
- More efficient fine-grain algorithms
  - ♦ Load/store rather than routine calls
  - ♦ Option for algorithms that include races (asynchronous iteration, ILU approximations)
- Adapt to modern node architeture…

Performance Bottlenecks with MPI Everywhere

- Classic Performance Model
  - T = s + rn
  - Model combines overhead and network latency (s) and a single communication rate 1/r
  - Good fit to machines when it was introduced (esp. if adapted to eager and rendezvous regimes)
  - But does it match modern SMP-based machines?

SMP Nodes: One Model

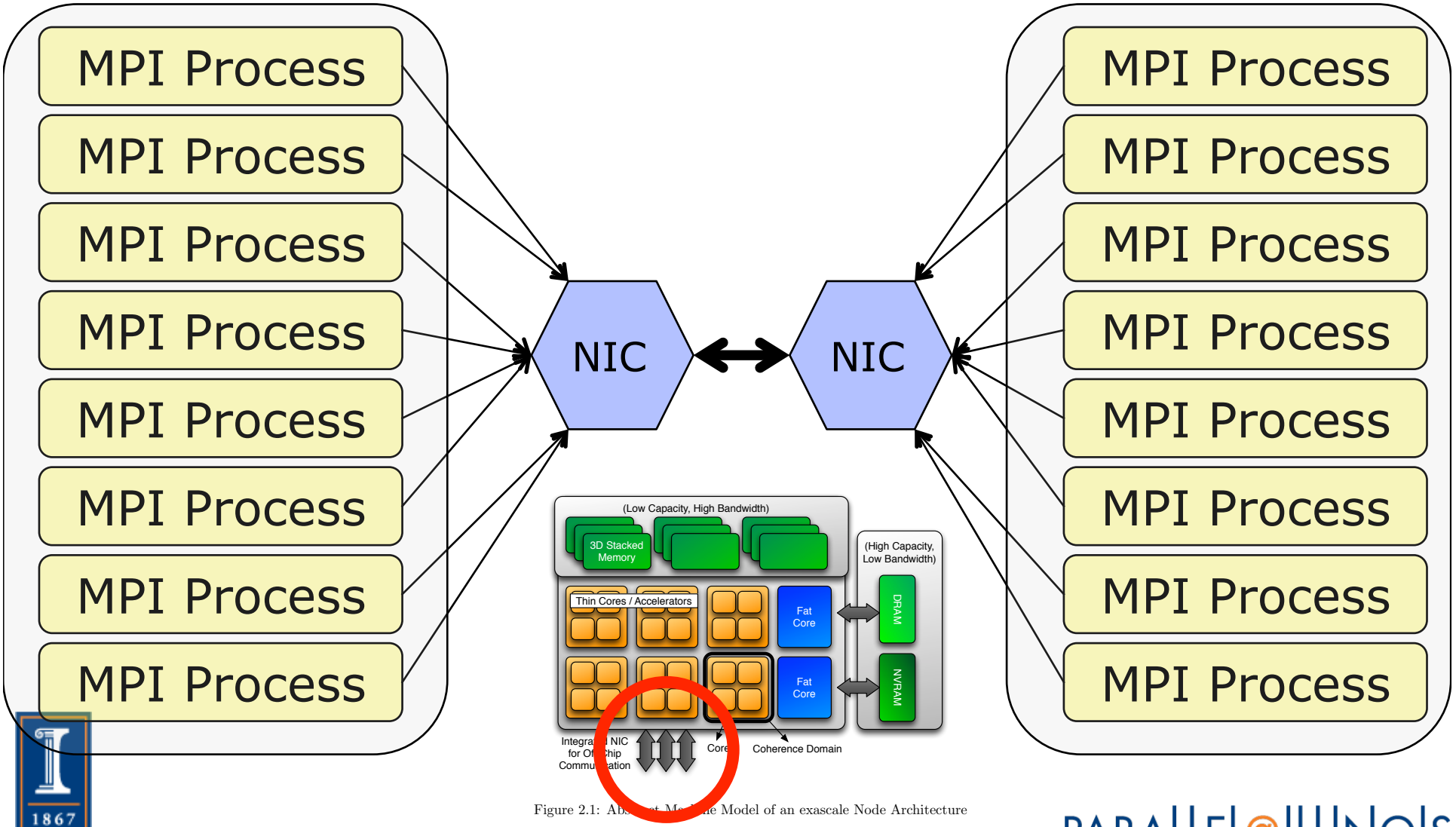


Figure 2.1: Abstract Machine Model of an exascale Node Architecture

PARALLEL@ILLINOIS

Modeling the Communication

- Each link can support a rate r_L of data
- Data is pipelined (Logp model)
  - ♦ Store and forward analysis is different
- Overhead is completely parallel
  - ♦ k processes sending one short message each takes the same time as one process sending one short message

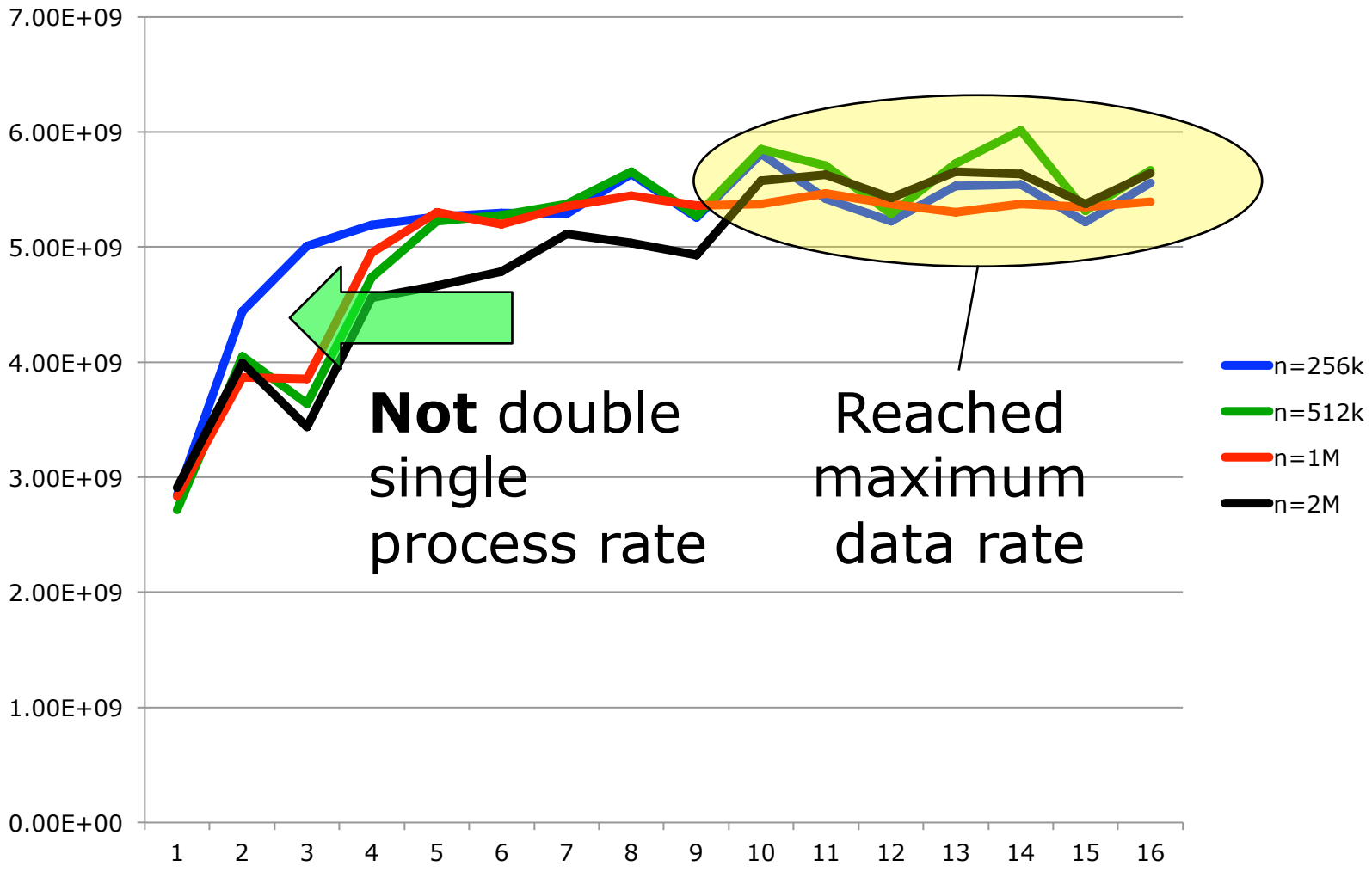
A Slightly Better Model

- Assume that the sustained communication rate is limited by
  - The maximum rate along any shared link
    - The link between NICs
  - The aggregate rate along parallel links
    - Each of the "links" from an MPI process to/from the NIC

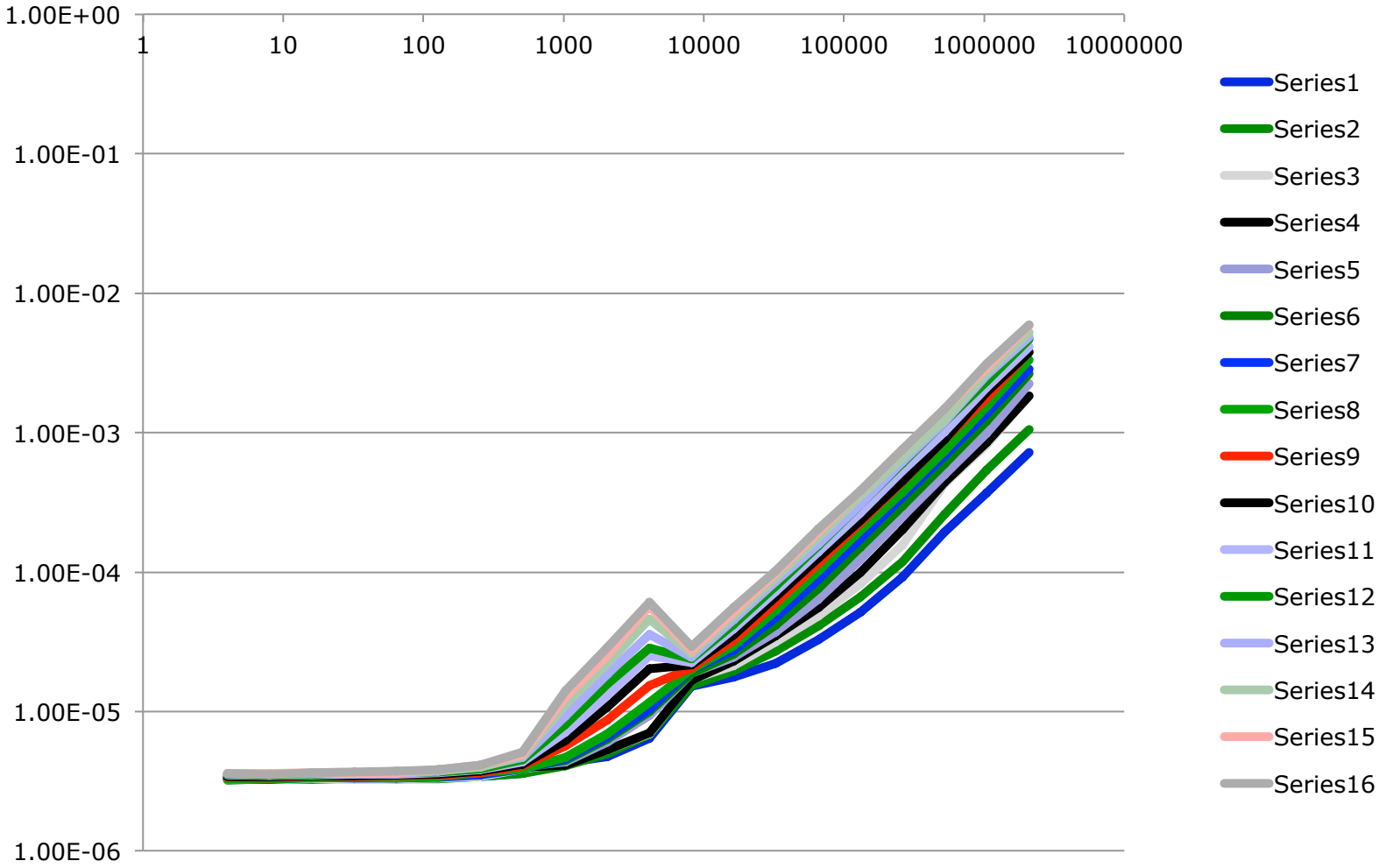
A Slightly Better Model

- For k processes sending messages, the sustained rate is
  - $\min(R_{NIC\text{-}NIC}, kR_{CORE\text{-}NIC})$
- Thus
  - $T = s + kn/\text{Min}(R_{NIC\text{-}NIC}, kR_{CORE\text{-}NIC})$
- Note if $R_{NIC\text{-}NIC}$ is very large (very fast network), this reduces to
  - $T = s + kn/(kR_{CORE\text{-}NIC}) = s + n/R_{CORE\text{-}NIC}$

Observed Rates for Large Messages



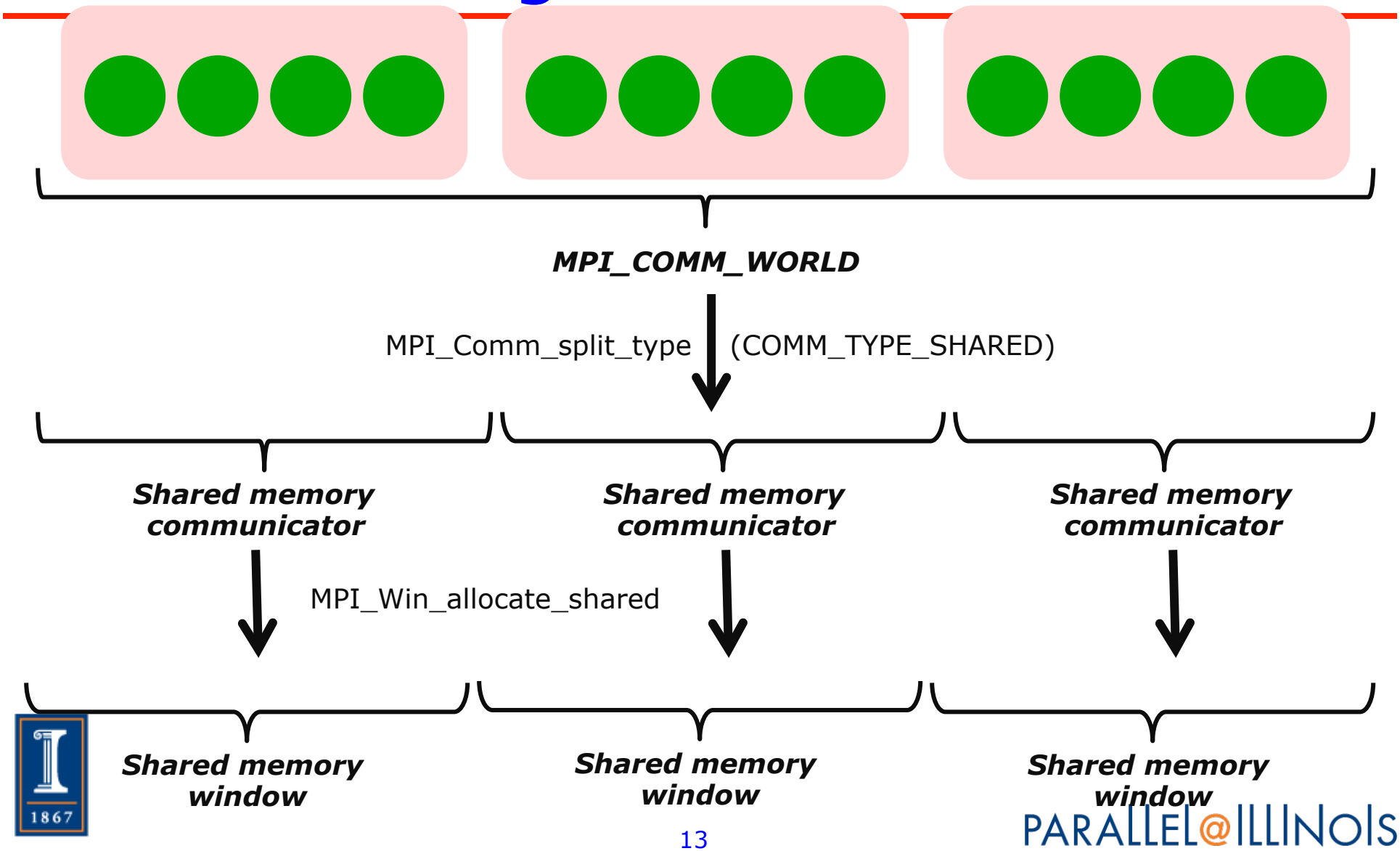
Time for PingPong with k Processes



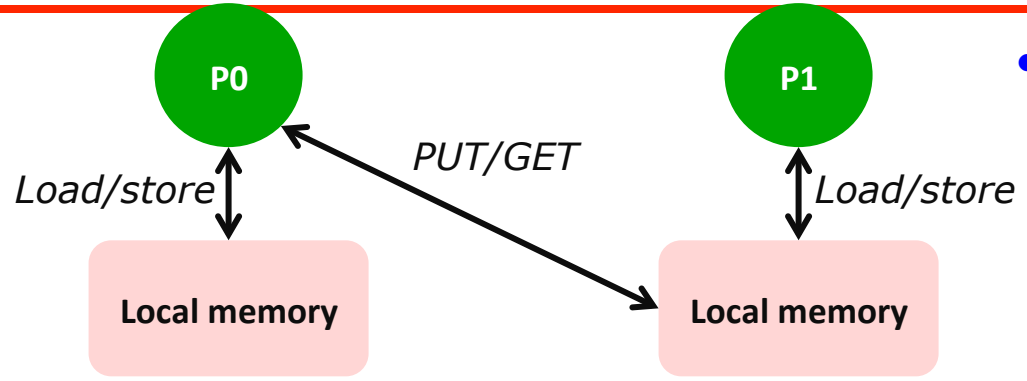
Hybrid Programming with Shared Memory

- MPI-3 allows different processes to allocate shared memory through MPI
  - MPI_Win_allocate_shared
- Uses many of the concepts of one-sided communication
- Applications can do hybrid programming using MPI or load/store accesses on the shared memory window
- Other MPI functions can be used to synchronize access to shared memory regions
- Can be simpler to program than threads

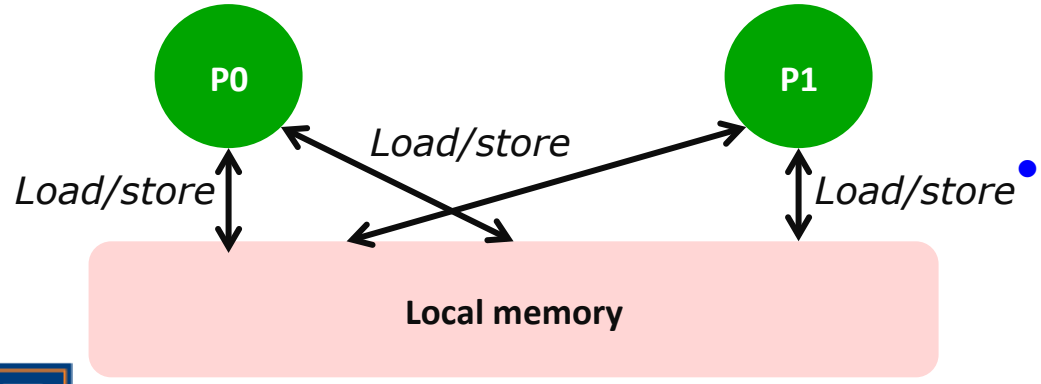
Creating Shared Memory Regions in MPI



Regular RMA windows vs. Shared memory windows



Traditional RMA windows



Shared memory windows

- Shared memory windows allow application processes to directly perform load/store accesses on all of the window memory
  - E.g., x[100] = 10
- All of the existing RMA functions can also be used on such memory for more advanced semantics such as atomic operations
- Can be very useful when processes want to use threads only to get access to all of the memory on the node
  - You can create a shared memory window and put your shared data

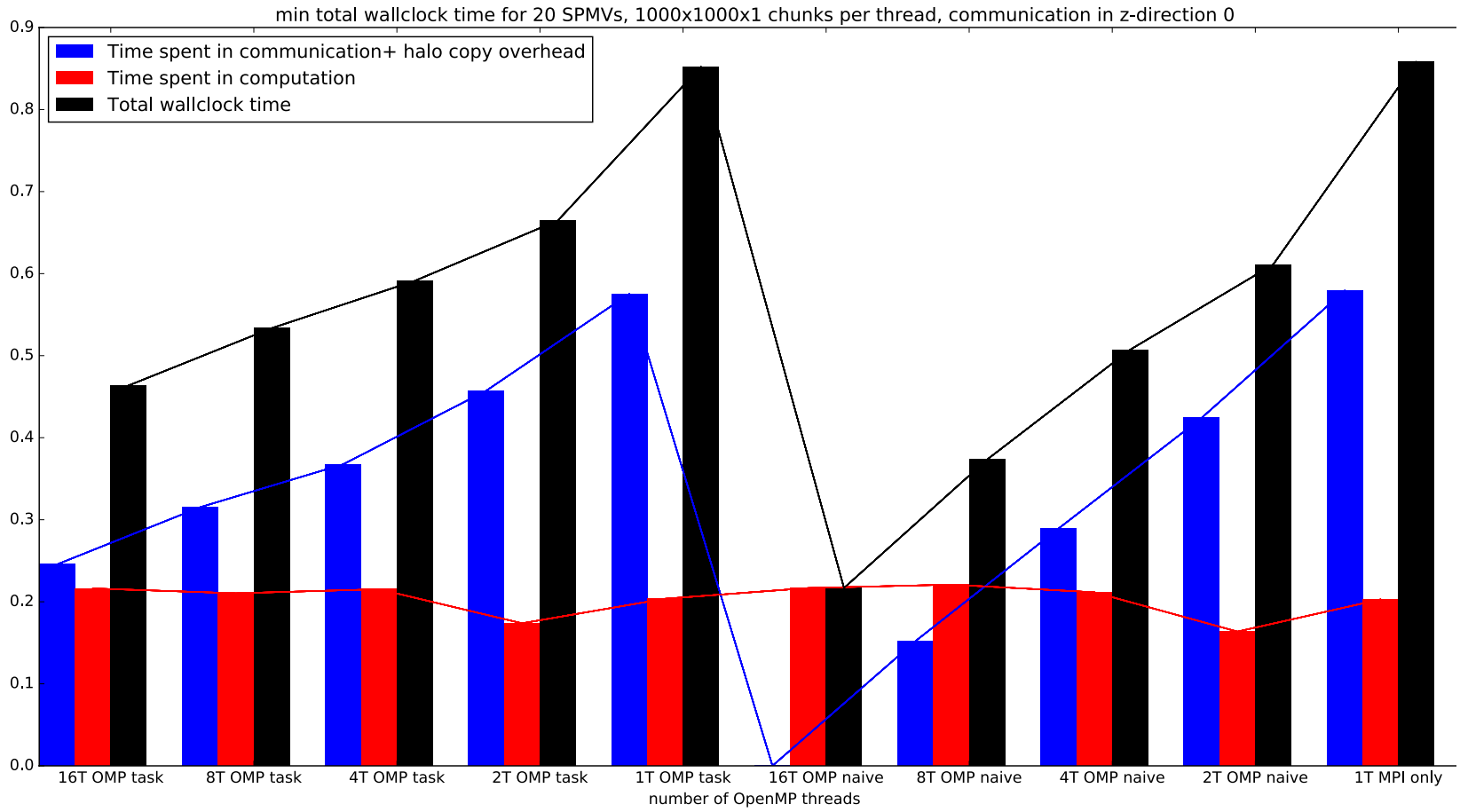
Shared Arrays With Shared Memory Windows

```
int main(int argc, char ** argv)
{
    int buf[100];

    MPI_Init(&argc, &argv);
    MPI_Comm_split_type(..., MPI_COMM_TYPE_SHARED, .., &comm);
    MPI_Win_allocate_shared(comm, ..., &win);

    MPI_Win_lockall(win);

    /* copy data to local part of shared memory */
    MPI_Win_sync(win);

    /* use shared memory */

    MPI_Win_unlock_all(win);

    MPI_Win_free(&win);
    MPI_Finalize();
    return 0;
}
```

Example: Using Shared Memory with Threads

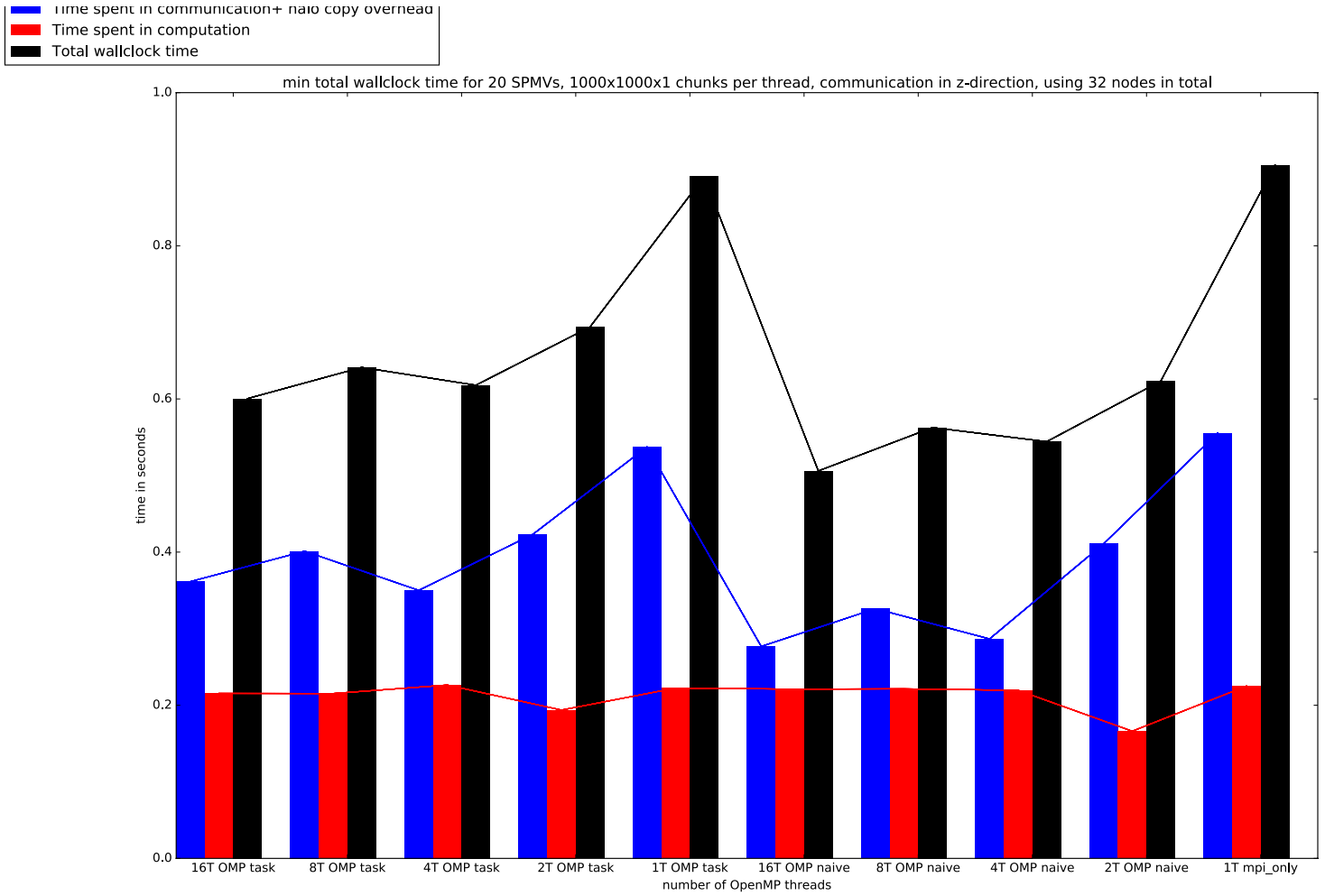
- Regular grid exchange test case
  - 3D regular grid is divided into subcubes along the xy-plane, 1D partitioning
  - Halo exchange of xy-planes: P0 -> \P1 -> P2 -> P3…
  - Three versions:
    - MPI only
    - Hybrid OpenMP/MPI model with loop parallelism, no explicit communication: "hybrid naïve”
    - Coarse grain hybrid OpenMP/MPI model, explicit halo exchange within shared memory: "hybrid task", threads essentially treated as MPI processes, similar to MPI SM
- A simple 7-point stencil operation is used as a test SPMV

Intranode Halo Performance



PARALLEL@ILLINOIS

Internode Halo Performance



PARALLEL@ILLINOIS

Summary

- Unbalanced interconnect resources require new thinking about performance
- Shared memory, used *directly* either by threads or MPI processes, can improve performance by reducing memory motion and footprint
- MPI-3 shared memory provides an option for MPI-everywhere codes
- Shared memory programming is *hard*
  - There are good reasons to use data parallel abstractions and let the compiler handle shared memory synchronization

Thanks!

- Philipp Samfass
- Luke Olson
- Pavan Balaji, Rajeev Thakur, Torsten Hoefler
- ExxonMobile
- Blue Waters Sustained Petascale Project