# MPI at Exascale: Challenges for Data Structures and Algorithms

William Gropp

# Challenges at Exascale

- Exascale is not just "more" Petascale
  - Concurrency
  - Fault Resilience
  - Memory Capacity
  - Power and Energy
- These are just what's needed to get something to *run* at Exascale
  - For an Exascale system to work effectively on applications, it must scale well
    - This implies communication/ computation overlap


**ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems**

**Peter Kogge, Editor & Study Lead**
**Keren Bergman**
**Shekhar Borkar**
**Dan Campbell**
**William Carlson**
**William Dally**
**Monty Denneau**
**Paul Franzon**
**William Harrod**
**Kerry Hill**
**Jon Hiller**
**Sherman Karp**
**Stephen Keckler**
**Dean Klein**
**Robert Lucas**
**Mark Richards**
**Al Scarpelli**
**Steven Scott**
**Allan Snavely**
**Thomas Sterling**
**R. Stanley Williams**
**Katherine Yelick**




**September 28, 2008**

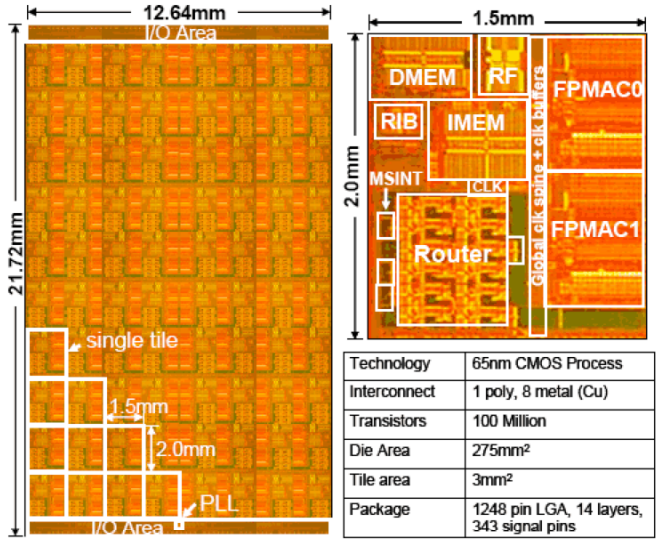
This work was sponsored by DARPA IPTO in the ExaScale Computing Study with Dr. William Harrod as Program Manager; AFRL contract number **FA8650-07-C-7724.** This report is published in the interest of scientific and technical information exchange and its publication does not constitute the Government's approval or disapproval of its ideas or findings

# What Will Exascale Systems Look Like?

- Constraints
  - ♦ Power: < 100MW total system (including cooling and storage)
    - Clock rate 4-10 GHz
    - Implies about $10^8$ functional units per peak ExaOp
    - 100-1000 times as power efficient as current Petascale systems
    - 4-40 pJ/operation
  - ♦ Cost and Size
    - 100-1000 Racks
      - (DARPA UHPC RFI envisions a 1PF Rack/half rack)
  - ♦ Two obvious approaches

# Very light-weight cores

- Very simple, low-power processors
  - ♦ in-order execution
  - ♦ small caches
  - ♦ no hardware cache coherence
- Intel TFLOPS processor an early example




| | |
|---|---|
| Technology | 65nm CMOS Process |
| Interconnect | 1 poly, 8 metal (Cu) |
| Transistors | 100 Million |
| Die Area | 275mm² |
| Tile area | 3mm² |
| Package | 1248 pin LGA, 14 layers, 343 signal pins |

# Heterogeneous

- Several different cores, optimized for different operations
  - GPGPU + Commodity processor a *very* small step in this direction
  - Exascale likely to be more tightly integrated (for power, fault) and more specialized (for power, density)

# Energy and Power

- Energy is “wasted” in data motion
- Need to compute closer to memory
  - ♦ Argues for embedded memory processors (EMP)
- Network Costs
  - ♦ Specialize here as well
  - ♦ Simple operations, offloaded from processor to make effective use of space and power
  - ♦ Not offloaded to increase performance at the cost of additional power

# How many MPI processes are there?

- Case 1: MPI Everywhere
  - Memory needs if all data local
  - Distributed data
    - Architectural support
- Case 2: nested parallelism
  - Hierarchical models and support; progress
    - Natural decomposition
    - Cross-node decomposition

# The Homogenous Approach

- MPI Everywhere
  - One MPI "process" per core
  - Note that an MPI process may be lighter-weight than an OS process
    - E.g., tmpi (thread MPI)
      - Used combination of compiler techniques and runtime to let each MPI "process" be an OS thread, with global variables handled correctly
- System parameters include
  - $10^8$ cores
  - $10^{18}$-$10^{22}$ bytes mass storage

# Lets look at possible issues

- Basic API Issues
- Memory use implied by distributed Memory model
- Scalability of algorithms used to implement MPI features

# Basic API Issues

- $10^8$ cores requires 27 bit integer to enumerate
  - Ranks will just fit into 4 byte integers
    - Possible problem: virtualization of processes to provide latency hiding could push this to more than 31 bits (ranks are signed integers)
  - Good news – with that many cores, memory / core is likely to be under 4GB
    - MPI_Aint can be 4 bytes
- $10^{18}$-$10^{22}$ bytes mass storage requires 60-74 bit integers
  - MPI_Offset must be large than an 8-byte integer
- Datatypes
  - Do these use MPI_Aint (4 bytes) or MPI_Offset (12-16 bytes)?
  - Using Datatypes for Messages and IO is elegant
    - But offsets in messages are relative to one MPI process whereas offsets in IO may be relative to p-processes

# Replicated Data Structures

- MPI objects are often used by many/every process
  - Simple and natural implementation is to have a representation in each MPI process
- Even simple objects, such as datatypes, represent significant storage when there are $10^8$ copies

# MPI_Group

- Consider an MPI_Group, consisting of g members
  - ♦ Either enumerated or a collection of strides
    - No MPI_Group_split
    - Thus no implicit description of groups
  - ♦ An enumerated list is good (even efficient) for <1K processes
  - ♦ Storage is O(pg) total
- Some groups are collective (results from MPI_Comm_group), some are individual (groups used in RMA PSCW)

# RMA Windows

- MPI_Win_create allows each process to specify a different local offset
  - This provides flexibility for applications with dynamic object creation (no need to require “symmetric” allocation of memory
- However, to perform direct remote memory access, the origin process needs the offset of the target process’s memory window
  - Simple and obvious implementation is a table of offsets
  - $O(p^2)$ memory required
  - For $10^8$ processes, this is $4*10^{16}$ bytes (40 Petabyes)

# MPI Communicators

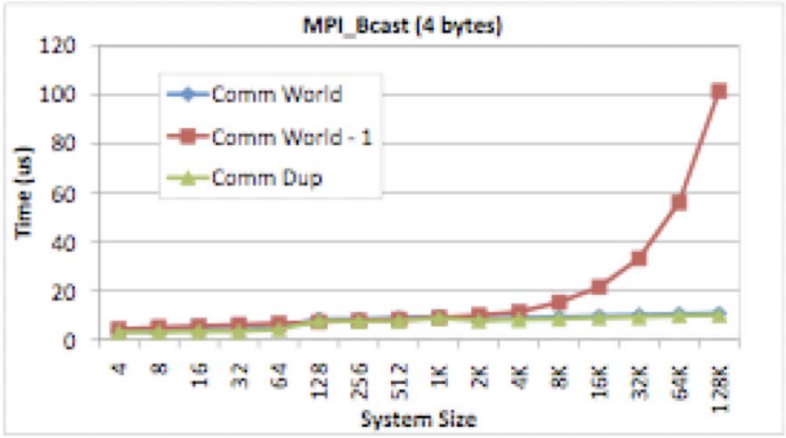
- Communicators have a similar problem as RMA Windows
- Each communicator must map each rank to a specific MPI process
  - Some one needs to be able to map this to a specific physical location in the parallel computer
  - Easiest: maintain a table
    - $O(p^2)$ – 40 Petabytes again
    - May have additional storage, such as state of connection/communication with each remote process

# Using Special Information

- Some special communicators can use implicit representations
  - E.g., MPI_COMM_WORLD (and its dups) on hardware where rank can be mapped to specific hardware, such as on a mesh
  - MPI_Comm_split could also store implicit representation in some cases
    - Row or column in a mesh
    - Immediate neighbors
  - But the general case no practical implicit representation exists
  - Should it be possible to manipulate implicit representations directly?
    - What if physical network is not simple?

# Cost of non-Canonical Communicators

- MPI_Bcast on MPI_COMM_WORLD and "MPI_COMM_WORLD – 1 process"




  - ♦ See "Toward message passing for a million processes: characterizing MPI on a massive scale BlueGene/P", Balaji, Chan, Thakur, Gropp, and Lusk

# Communication Queues

- Simple implementations of message-passing queues can lead to problems
  - Simple implementation uses a single queue
  - In alltoall communication, each search-and-remove takes O(p) time, for $O(p^2)$ total work
  - See “Non-Data-Communicatin Overheads in MPI: Analysis on Blue Gene/P, “ Balaji, Chan, Thakur, Gropp, Lusk

# Some Queue Options

- Single Queue for all communications
  - Excellent performance on pingpong benchmark
  - Search time O(q), q=queue length
- Separate Queue for each partner
  - Search time likely to be O(1)
  - Requires O(p) storage on each process, or $O(p^2)$ total
    - Optimziation: queue only for active partner
    - Adds overhead to deal with new partners, agin old partners
  - Has “MPI_ANY_SOURCE” problem
- Single Queue, with hash for search
  - Adds some extra overhead (and thus costs power and time)

# Aside on the "ANY_SOURCE" problem

- Many data-parallel applications are (or should be) deterministic and do not *require* ANY_SOURCE
- Why not get rid of it?
  - More dynamic computations may be weakly deterministic – enforcing a specific completion order may impact scalability
  - May also not have predictable source
- Should we look at alternatives?
  - Most applications *either* want messages from a specific sender *or* the next message from any sender in the group of senders-whose-destinations-I-don't-know-in-advance
  - Why mix these in the same interface?
  - Use a separate interface for each; optimize for each type separately
  - Same idea as heterogeneous nodes – optimize for separate function
    - Something for MPI-3?

# Buffer Management

- Common to use eager buffering to reduce overhead of short messages
- Simple (and most efficient for p < 1k) approach is to preallocate a buffer for each process
  - ♦ $O(p^2)$ data required
    - For a mere 16k bytes/buffer, requires $2*10^{20}$ bytes (200 Exabytes)
- Alternative – provide buffering to preselected partners
  - ♦ Matches many simulations, particular PDE-based ones
  - ♦ Ancient approach (available on Intel Paragon)
- More sophisticated alternative, adaptive buffer allocation
  - ♦ Current project of Dooley, Kale, Gropp
    - Reduces necessity for rendezvous or other control messages (good)
    - But adds overhead to decision (bad)
    - Recall control messages use energy with performing useful computation

# Common Issues

- Partitioned, local address space
  - Gives good locality, but
  - Encourages “early, deep copies”
- Alternate approach
  - Late, shallow copies
  - Also known as caching
- But
  - Adds a level of indirection
  - MPI Implementation must have low-latency access to remote data
  - Programmer-assisted prefetch will (probably) be needed
    - As with current, high-performance caches

# Replicated Data Structures

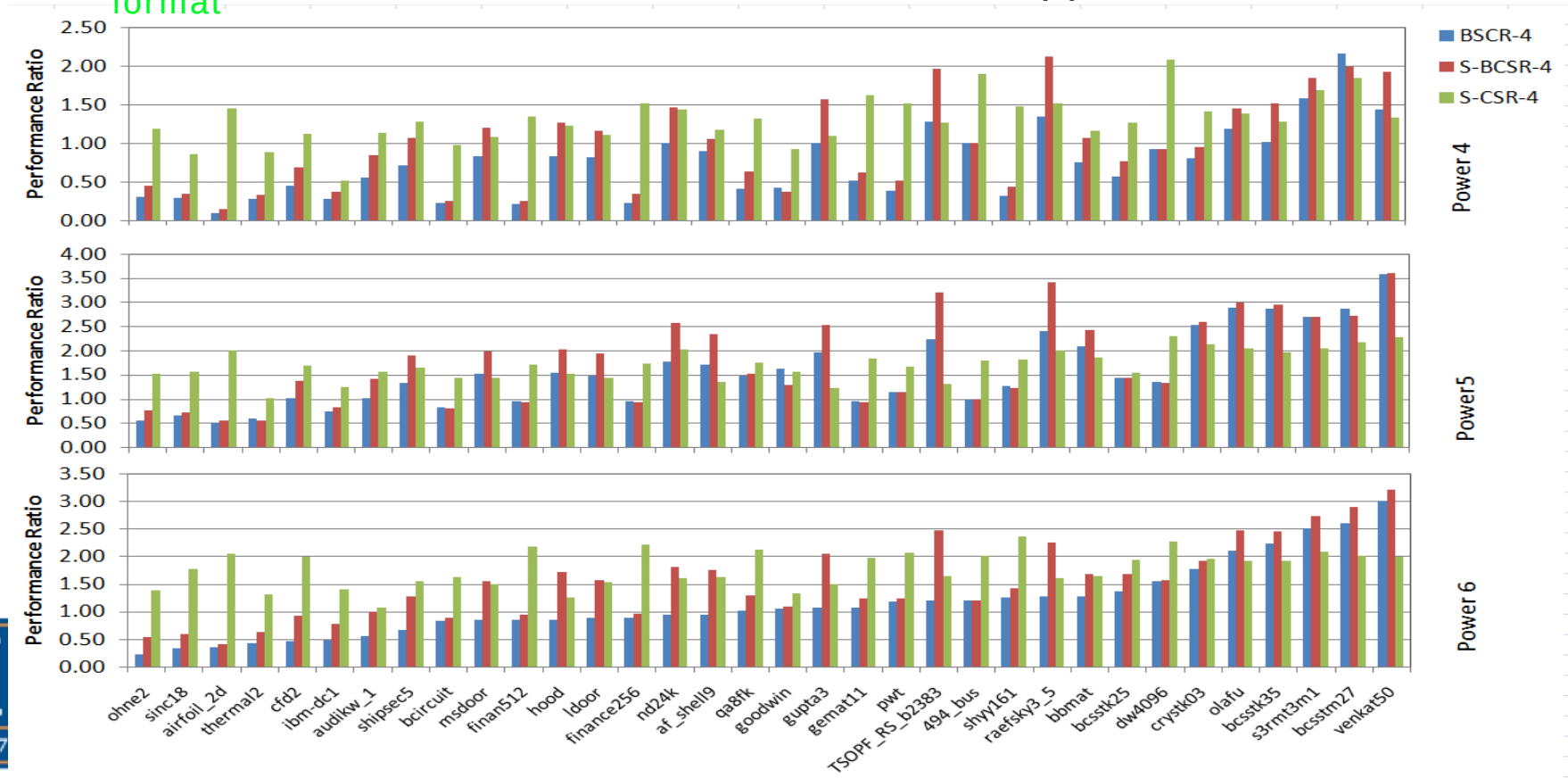
- MPI Communicators are an example of using replication of identical or similar data structures
  - User applications (too) often do the same thing
  - Cannot afford this approach at Exascale
- One solution
  - Don't replicate - distribute
  - Cache values actually needed locally
    - Adds overhead – looking up in cache must be very fast
  - Use remote load/get on a cache miss
    - Mapping must be simple to compute (no tables!)

# Example of the Benefit of Programming for Prefetch

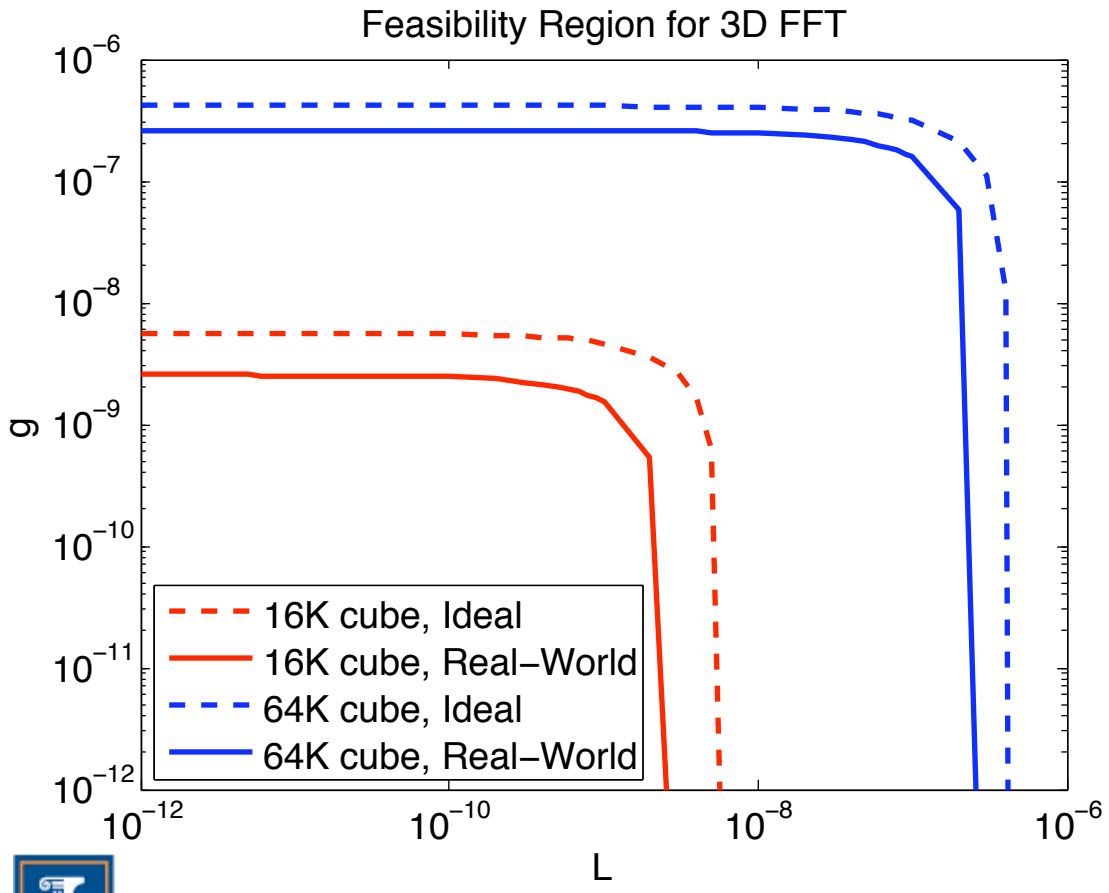
- S-CSR format is better than CSR format for all (on Power 5 and 6) or Most ( on Power 4) matrices
- S-BCSR format is better than BCSR format for all (on Power 6) or Most ( on Power 4 and 5) matrices
- Blocked format performance from ½ to 3x CSR.

Work of Dahai Guo (NCSA) and Gropp

Performance Ratio Compared to CSR format



# Performance Requirements for an AlltoAll Algorithm




Work of Gahvari and Gropp

- Global 3D FFT a very demanding application
- Best case assumption: only communication cost, using LogP model (with overhead = zero)
- Current latency +overhead times are .25-10 usec
  - Roughly the right edges

# Implementing MPI Operations

- Some MPI operations are non-local
  - For these, scalability must be evaluated
- MPI_Comm_split is a powerful, elegant method for creating new communicators
  - No explicit enumeration of processes
- The other option is to create a group, then use MPI_Comm_create
  - But using a group (with enumerated members) requires $O(p^2)$ total space
- But can MPI_Comm_split be implemented scalably?

# Implementing COMM_SPLIT

- MPI_COMM_SPLIT(inComm, color, key, &outComm)
  - Processes with same color are in the same output communicator
  - Key used to order ranks
- Simple Algorithm
  - Alltoall (color,rank)
  - Each process locally finds those in the same color, ordered according to rank
  - Create new communicator from that information

# Scalability Analysis

- Allgather
  - ♦ $O(p^2)$ space to store tables
    - Communication time depends on interconnect, but includes O(p) term for amount of data and probably at least log p latency – $O(p^2)$ total communication work
- P logp time to sort to find processes in the same communicator and order rank by key.
- The Simple algorithm for MPI_Comm_split is not scalable

# A (sketch of a more) Scalable Algorithm

- Work of Paul Sack
- Solution: distributed tree structure for communicators
  - No process ever has the entire data structure
  - Use parallel sort by color/key
  - Then distribute results to the processes sharing the same color
    - Space is $O(p^{4/3})$
      - For $10^8$, this is only about $4 * 10^{10}$
- Further tune by hybridizing
  - Some local copies
    - Faster, but redundant – watch that energy use
  - Design for smooth performance from small to enormous
  - Also optimize for special cases, such as
    - Few colors
- Key=rank in oldComm

# Summary for MPI Everywhere Case

- Exascale stretches the 64-bit integer space
  - Should MPI skip from external32 directly to external128?
- Flexibility of independent operations at every rank add cost in time/space
  - But other extreme, rigid, COMM_WORLD – only, limits applications (e.g., AMR, dynamic algorithms)
- Replicated data structures are not viable at (homogeneous) Exascale
  - Both software and hardware support required for late/shallow copies
  - Such features are also of general value to the programmer
- Algorithms must be revisited for scaling
  - COMM_SPLIT is not as bad as you might think
  - Solution leverages support for distributed data structures
  - But all-to-all algorithms are in trouble

# Is a Homogeneous System the Answer?

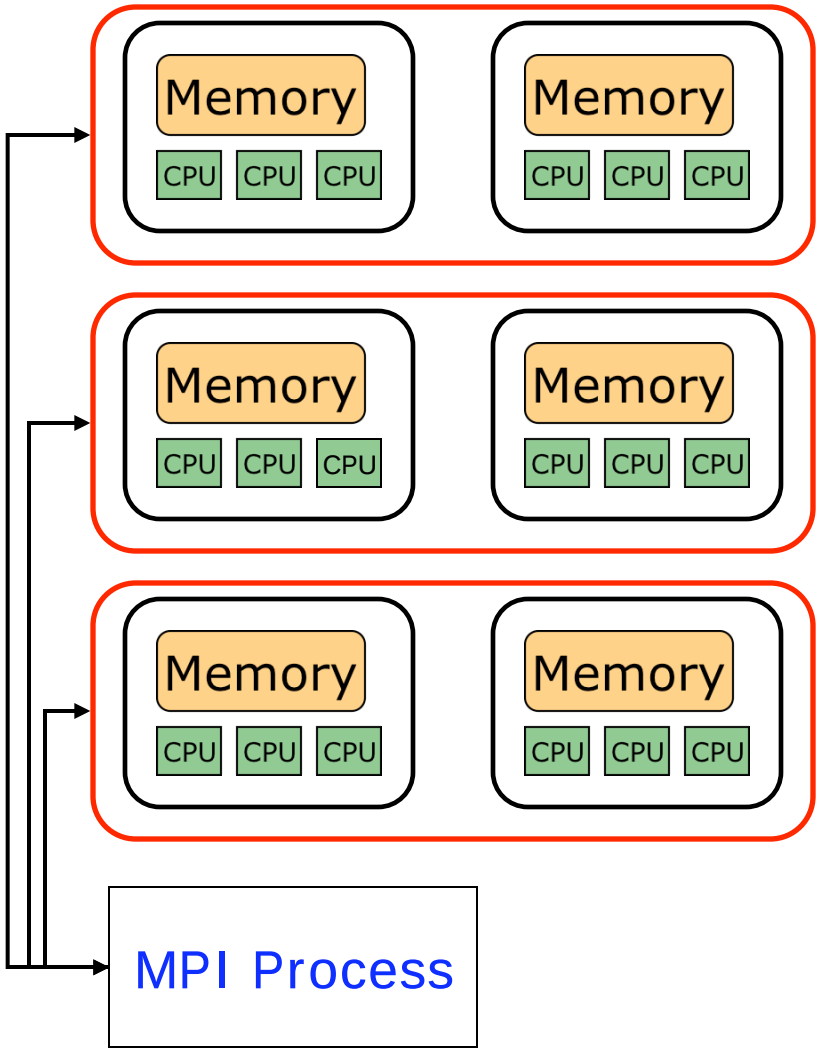
- What if instead we have 100k "processes", each with 1000-fold parallelism?
  - ♦ MPI already runs (with some struggle) with this number of processes
  - ♦ Implementations do need improvement
  - ♦ Enumeration of ranks etc. is still a concern
    - Note memory capacity is an issue for exascale systems – total memory per core may be low relative to current systems
  - ♦ This still adds complexity to the programming model

# Hybrid Models for MPI

- MPI on SMP
  - Typical implementation
    - Use regular processes for each MPI process, use special services to share memory between the processes
  - An alternative
    - Use each MPI process on the SMP is a thread that is part of a single operating system process
    - See “Optimizing threaded MPI execution on SMP clusters”, H. Tang and T. Yang
    - Must use a special compiler
    - Global variables in the user program must be thread-private by default
- Question: should we rethink the identification of MPI processes with OS Processes?
  - Particularly with respect to memory requirements

# Generalized MPI Processes

- Let an MPI "Process" span multiple nodes
  - Solves the memory problem
  - Provides a way to address the local/global problem
- Issues
  - What does a send with a remote pointer mean?
  - What is the address space for an MPI process?
  - Initializing - who is in charge?
  - Programming model support for the MPI "Process"
    - Distributed OpenMP?
    - UPC? CAF?

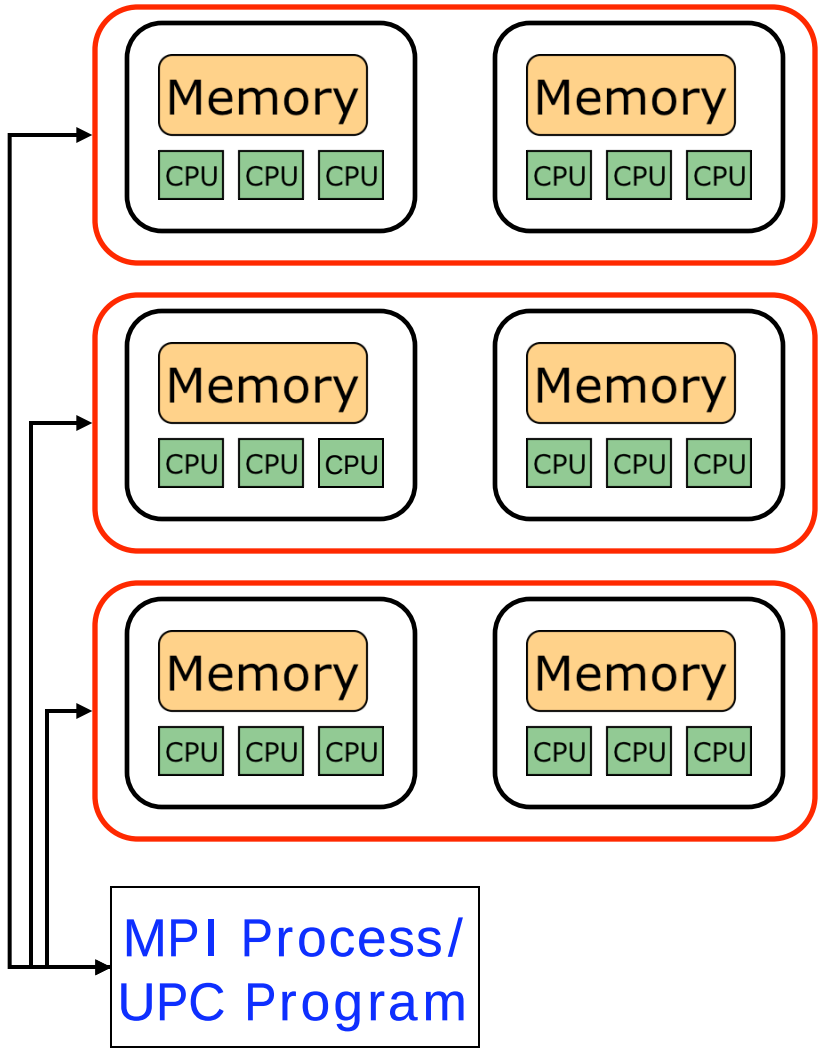



# More General MPI Hybrid Programming Models

- Why consider the Hybrid Model with PGAS or other programming models?
  - Load balancing
  - Shared data (reduce memory pressure, particularly for processor-rich (and hence memory poor) nodes)
  - Component software (use the best programming model to implement a component)
  - OpenMP and MPI understood
  - What about others: MPI/UPC (or PGAS) interoperability
- The following is based on discussions at a "Workshop on collective communication primitives in PGAS and SPMD languages", May 2008, IBM Hawthorne
- Possible combinations for MPI and UPC (or other PGAS) languages include:
  - MPI processes are UPC programs
  - MPI processes are UPC threads
  - UPC Programs are combined into MPI programs

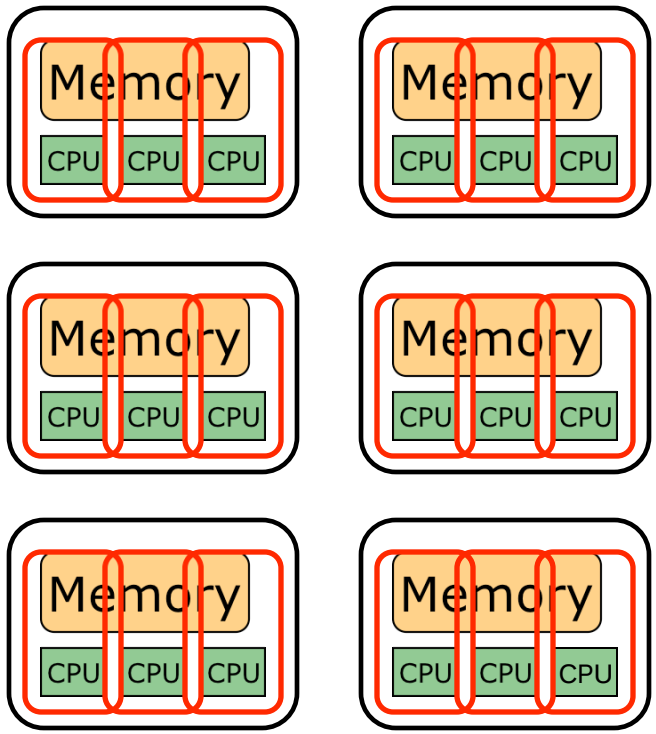
# MPI Processes are UPC Programs

- MPI Processes are UPC programs (not threads), spanning multiple nodes. This model is the closest counterpart to the MPI+OpenMP model, using PGAS to extend the "process" beyond a single node. (An MPI process need not be an OS process).



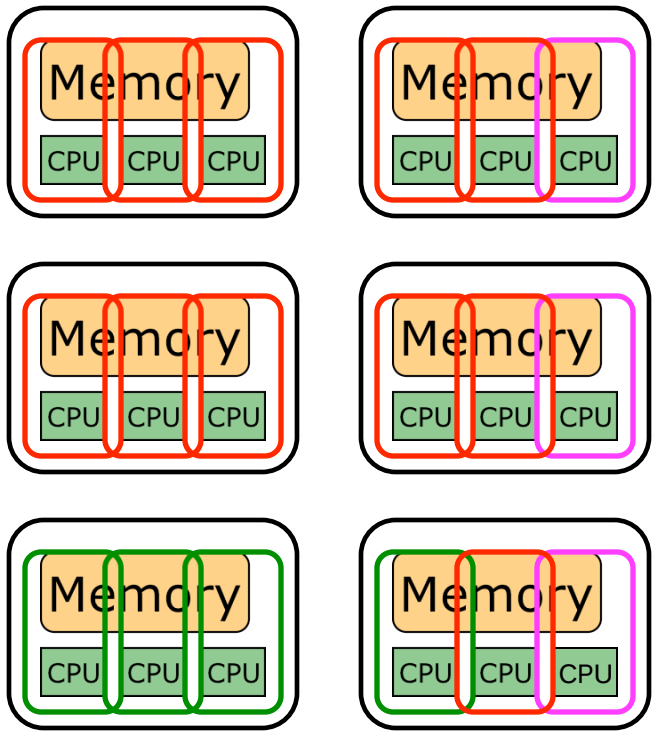

# MPI Processes are UPC Threads

- The program starts as a single UPC program. Each UPC thread calls MPI_Init (or MPI_Init_thread). The process management system must permit UPC programs to use MPI_Init to also become MPI programs.
- The program starts as a single MPI program (started with mpiexec). UPC is initialized somehow
  - UPC initialized explicitly with a routine call
  - UPC initialized implicitly because UPC compiler knew this was an MPI + UPC program




# MPI Processes are UPC Threads (con't)

- The MPI program is tiled with separate UPC programs. That is, every MPI process is also a UPC thread, but not all MPI processes belong to the same UPC program.
  a) The UPC programs are created from MPI subcommunicators with an explicit call, e.g., add a upc_init( MPI_Comm ) (proposed by Marc Snir)
  b) The UPC programs are defined at startup through interaction with the process management system; e.g., an extension to mpiexec defines how the MPI processes are tiled with UPC programs.
  c) Like (b), but not all MPI processes correspond to a UPC thread. This is like (a) if not all MPI processes were to call upc_init.




# The Program is a Collection of UPC Programs

- The program starts as a collection of separate UPC programs.
  - Use MPI_Comm_connect/accept to become an MPI program on all threads
  - Use MPI_Comm_connect/accept to become an MPI program on a subset of UPC threads
- Both require efficient support for updating routing information

# Handling Faults in Memory

- A major source of faults in current, large systems are transient memory faults (e.g., two-bit upsets)
- Some data must be robust (cannot be recomputed, updated in random way)
  - Control data, often on stack. Failures not recoverable.
- Some data may be recoverable
  - Large arrays. If updated en masse, recovery possible with in-memory ECC
- Should we have different kinds of memory for these different cases?

# MPI Fault Issues: Memory

- MPI application data is in several categories
  - Shared, updated frequently. Failures not recoverable
  - Static (const object), such as communicator. Rebuild from software; ECC possible
  - Cached. Recover from other copies possible

# Aside: Thoughts on Reducing Impact of Faults

- User application data is in similar categories to MPI data
  - ♦ May want to generalize this:
    - Stack data is robust; use hardware to provide additional reliability
    - Large object (e.g., array) data is protected in collaboration with programming model, algorithm
  - ♦ Need not guarantee all faults handled (impossible anyway). (Just make the rate of unrecoverable faults small enough)
    - Beginning to explore these ideas, particularly wrt numerical algorithms, with Elizabeth Jessup of U Colorado at Boulder

# Conclusions

- An MPI Everywhere model will be challenging for an Exascale system
  - Not impossible, however, particular with more distributed implementation strategies
  - Such strategies will be needed by applications as well
- A Hybrid Model reduces demands on MPI
  - But increases demands on an efficient interface between MPI and other programming models
- Faults, scaling may require new algorithmic approaches, both in applications and in MPI implementations