MPI at Exascale

Rajeev Thakur

Mathematics and Computer Science Division

Argonne National Laboratory

MPI on the Largest Machines Today

- Systems with the largest core counts in June 2010 Top500 list

Juelich BG/P	294,912 cores
Oak Ridge Cray XT5	224,162 cores
LLNL BG/L	212,992 cores
Argonne BG/P	163,840 cores
LLNL BG/P (Dawn)	147,456 cores

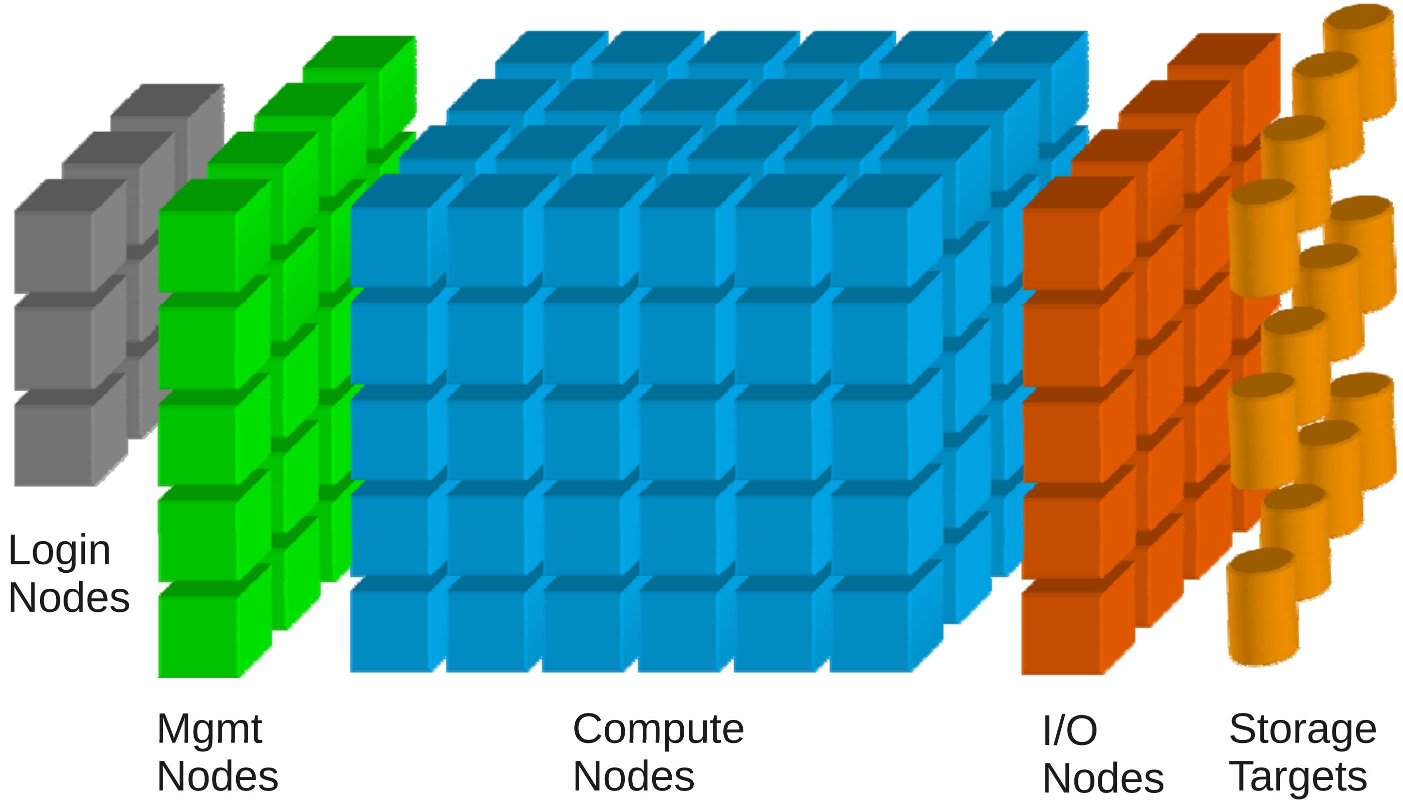
(All these systems run MPICH2-based MPI implementations)

- In a couple of years, we will have systems with more than a million cores

- For example, in 2012, the Sequoia machine at Livermore will be an IBM Blue Gene/Q with 1,572,864 cores (~1.6 million cores)

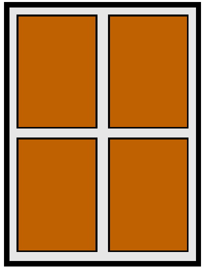
Future Extreme Scale Platforms

- Hundreds of thousands of “nodes”
- Each node has large numbers of cores, including
  - Regular CPUs and accelerators (e.g., GPUs)

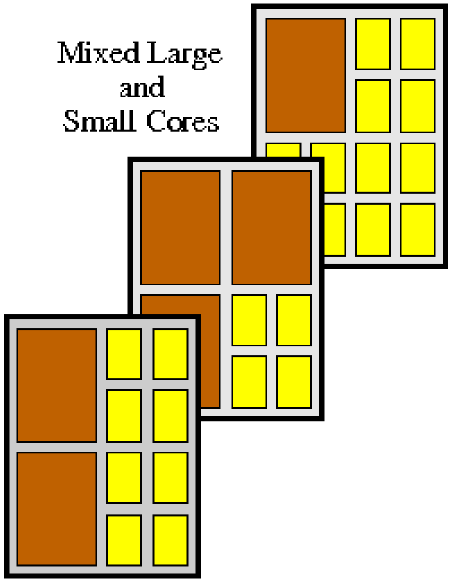


Multiple Cores Per Node

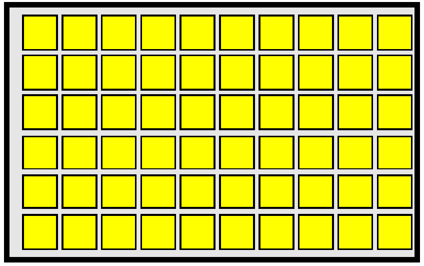
All Large Cores



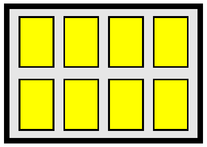
Mixed Large and Small Cores



Many small cores

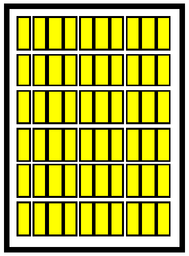


All small cores

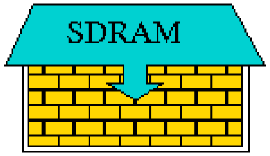


Different Classes of Chips
- Home
- Games/Graphics
- Business
- Scientific

Memory Floating Point Cores



+ 3D Stacked Memory



Scaling MPI to Exascale

- MPI already runs on the largest systems today at ~300,000 cores

- What would it take to scale MPI to exascale systems with millions of cores?

- On exascale, MPI is likely to be used as part of a “hybrid programming” model (MPI+X), much more so than it is today
  - MPI being used to communicate between “address spaces”
  - With some other “shared-memory” programming model (OpenMP, UPC, CUDA, OpenCL) for programming within an address space

- How can MPI support efficient “hybrid” programming on exascale systems?

Scaling MPI to Exascale

- Although the original designers of MPI were not thinking of exascale, MPI was always intended and designed with scalability in mind. For example:
  - A design goal was to enable implementations that maintain very little global state per process
  - Another design goal was to require very little memory management within MPI (all memory for communication can be in user space)
  - MPI defines many operations as *collective* (called by a group of processes), which enables them to be implemented scalably and efficiently

- Nonetheless, some parts of the MPI specification may need to be fixed for exascale
  - Being addressed by the MPI Forum in MPI-3

Factors Affecting MPI Scalability

- Performance and memory consumption
- A nonscalable MPI function is one whose time or memory consumption per process increase linearly (or worse) with the total number of processes (all else being equal)
- For example
  - If memory consumption of MPI_Comm_dup increases linearly with the no. of processes, it is not scalable
  - If time taken by MPI_Comm_spawn increases linearly or more with the no. of processes being spawned, it indicates a nonscalable implementation of the function
- Such examples need to be identified and fixed (in the specification and in implementations)
- The goal should be to use constructs that require only constant space per process

Requirements of a message-passing library at extreme scale

- No O(*nprocs*) consumption of resources (memory, network connections) per process
- Resilient and fault tolerant
- Efficient support for hybrid programming (multithreaded communication)
- Good performance over the entire range of message sizes and all functions, not just latency and bandwidth benchmarks
- Fewer performance surprises (in implementations)

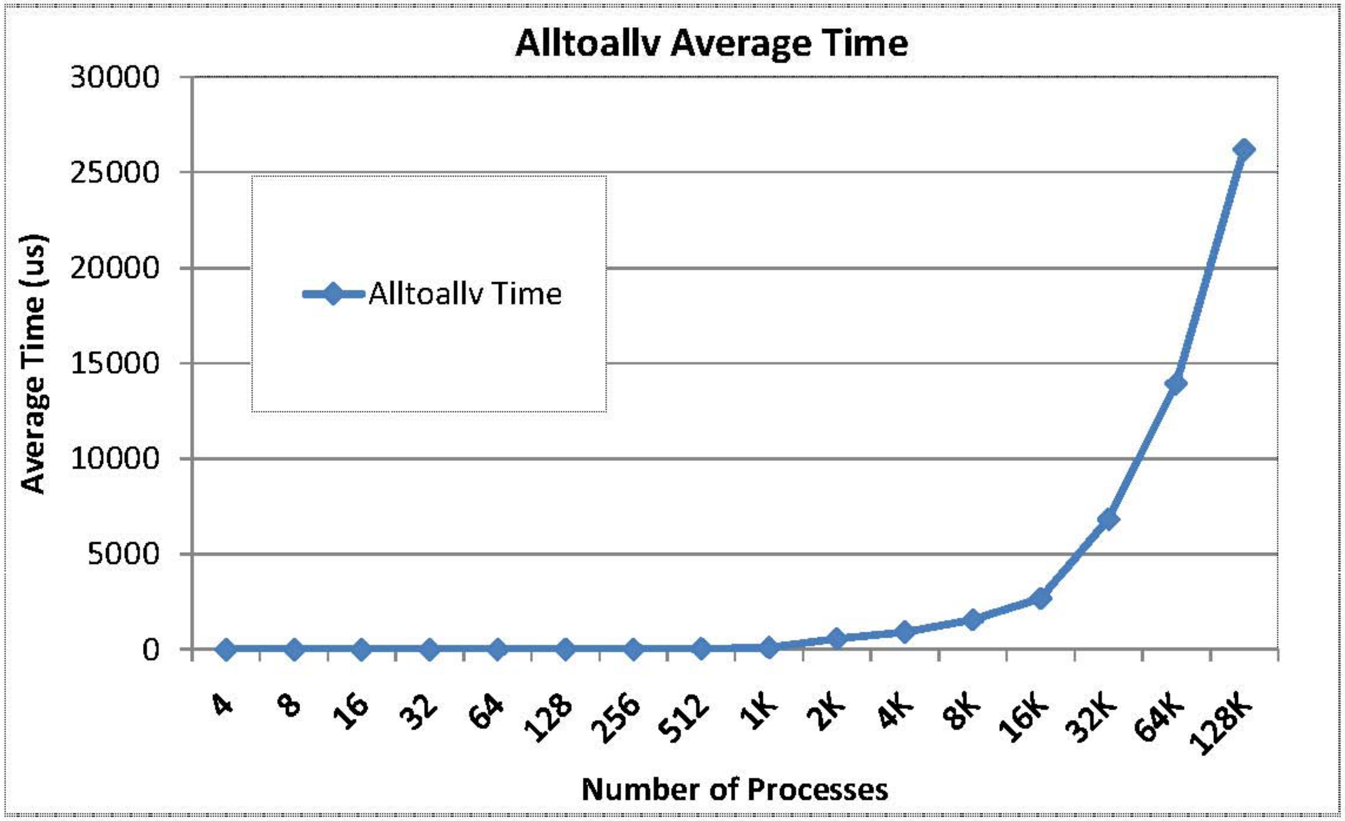
- These issues are being addressed by the MPI Forum for MPI-3 and by MPI implementations

Scalability Issues in the MPI Specification

- Some functions take parameters that grow linearly with number of processes
- E.g., irregular (or “v”) version of collectives such as MPI_Gatherv
- Extreme case: MPI_Alltoallw takes six such arrays
    - On a million processes, that requires 24 MB on each process
- On low-frequency cores, even scanning through large arrays takes time (see next slide)

- Solution: The MPI Forum is considering a proposal to define sparse, neighborhood collectives that could be used instead of irregular collectives

Zero-byte MPI_Alltoallv time on BG/P



- This is just the time to scan the parameter array to determine it is all 0 bytes. No communication performed.

Scalability Issues in the MPI Specification

- Graph Topology
  - In MPI 2.1 and earlier, requires the entire graph to be specified on each process
  - Already fixed in MPI 2.2 – new distributed graph topology functions

- One-sided communication
  - Synchronization functions turn out to be expensive
  - Being addressed by RMA working group of MPI-3

- Representation of process ranks
  - Explicit representation of process ranks in some functions, such as MPI_Group_incl and MPI_Group_excl
  - Concise representations should be considered

Scalability Issues in the MPI Specification

- All-to-all communication
  - Not a scalable communication pattern
  - Applications may need to consider newer algorithms that do not require all-to-all

- Fault tolerance
  - Large component counts will result in frequent failures
  - Greater resilience needed from all components of the software stack
  - MPI can return error codes, but need more support than that
  - Being addressed in the fault tolerance group of MPI-3

MPI Implementation Scalability

- MPI implementations must pay attention to two aspects as the number of processes is increased:
  - memory consumption of any function, and
  - performance of *all* collective functions
    - Not just collective communication functions that are commonly optimized
    - Also functions such as MPI_Init and MPI_Comm_split

Process Mappings

- MPI communicators maintain mapping from ranks to processor ids
- This mapping is often a table of O(nprocs) size in the communicator
- Need to explore more memory-efficient mappings, at least for common cases
- More systematic approaches to compact representations of permutations (research problem)
  - See recent paper at HPDC 2010 by Alan Wagner et al. from the University of British Columbia

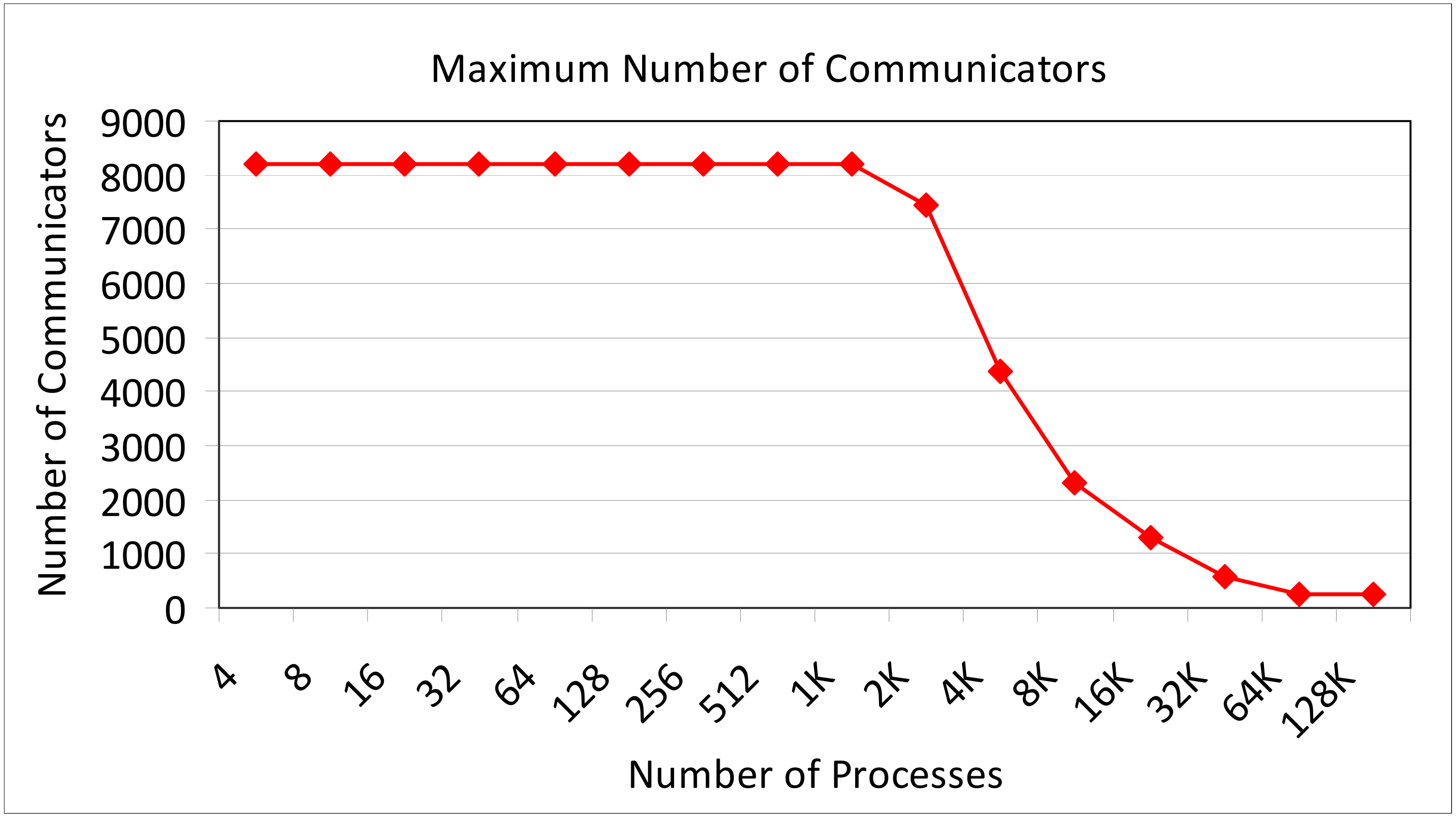
Communicator Memory Consumption

- NEK5000 is a well-known fluid dynamics code developed by Paul Fischer and colleagues at Argonne
- When they first tried to scale this code on the BG/P, it failed on as little as 8K processes because the MPI library ran out of communicator memory
- NEK5000 calls MPI_Comm_dup about 64 times (because it makes calls to libraries)
- 64 is not a large number, and, in any case, MPI_Comm_dup should not consume O(nprocs) memory (it doesn't in MPICH2)

- We ran an experiment to see what was going on…

Communicator Memory Consumption with original MPI on BG/P

- Run MPI_Comm_dup in a loop until it fails. Vary the no. of processes



What was going on --- and the fix

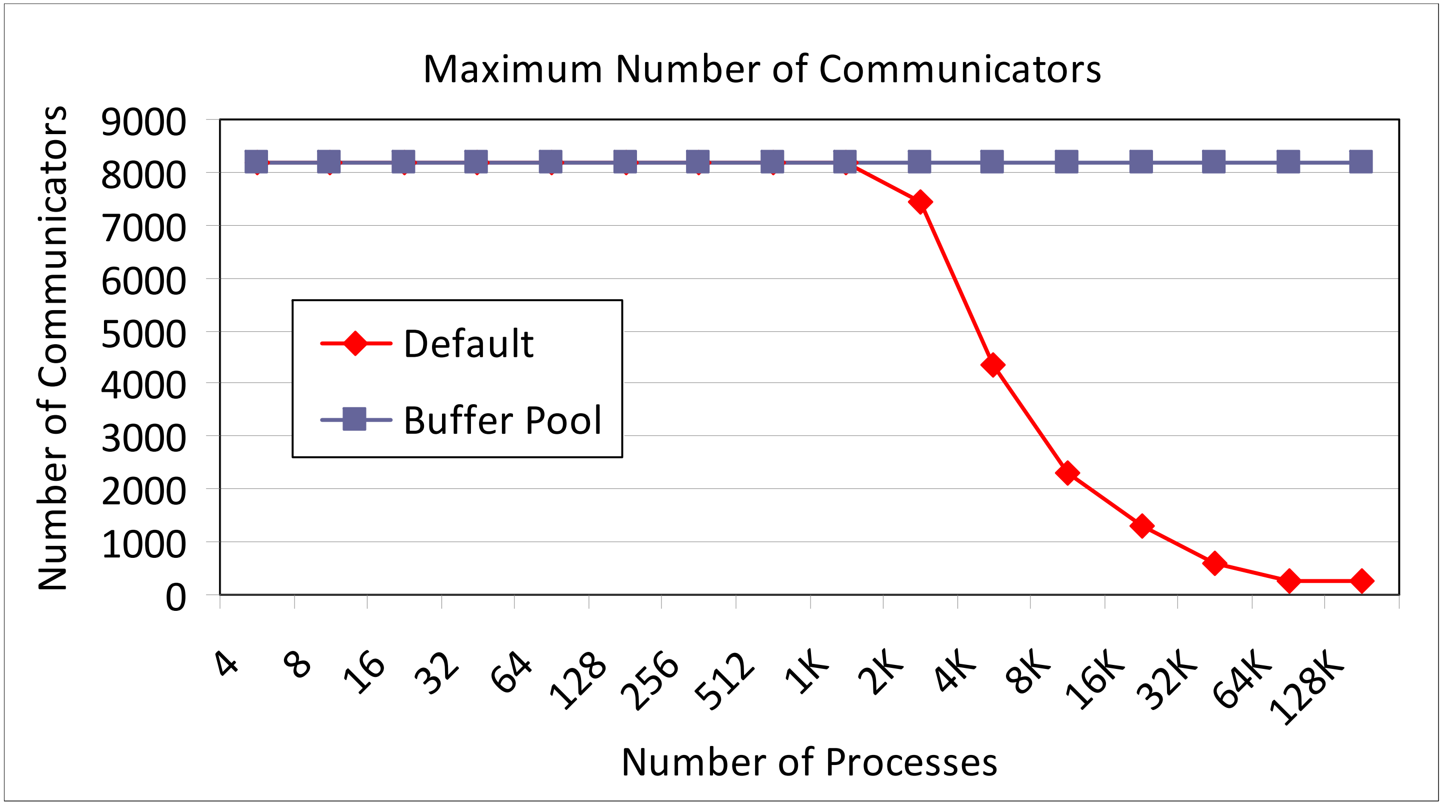
- The default MPI_Comm_dup in IBM's MPI was allocating memory to store process mapping info for optimizing future calls to collective communication (Alltoall)
- Allocated memory was growing linearly with system size
- One could disable the memory allocation with an environment variable, but that would also disable the collective optimizations

- On further investigation we found that they really only needed one buffer per thread instead of one buffer per new communicator

- Since there are only four threads on the BG/P, we fixed the problem by allocating a fixed buffer pool within MPI

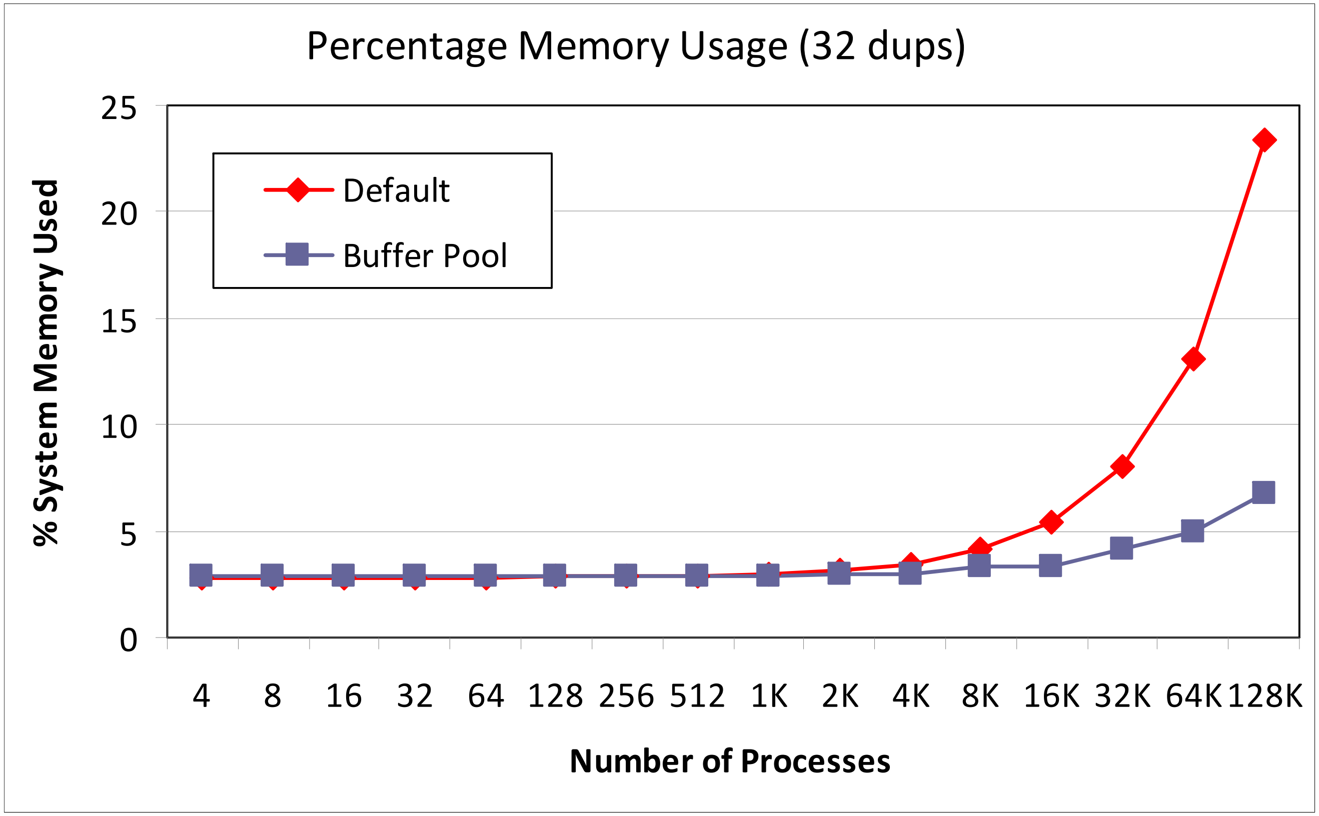
- We provided IBM with a patch that fixed the problem

Communicator Memory Consumption Fixed



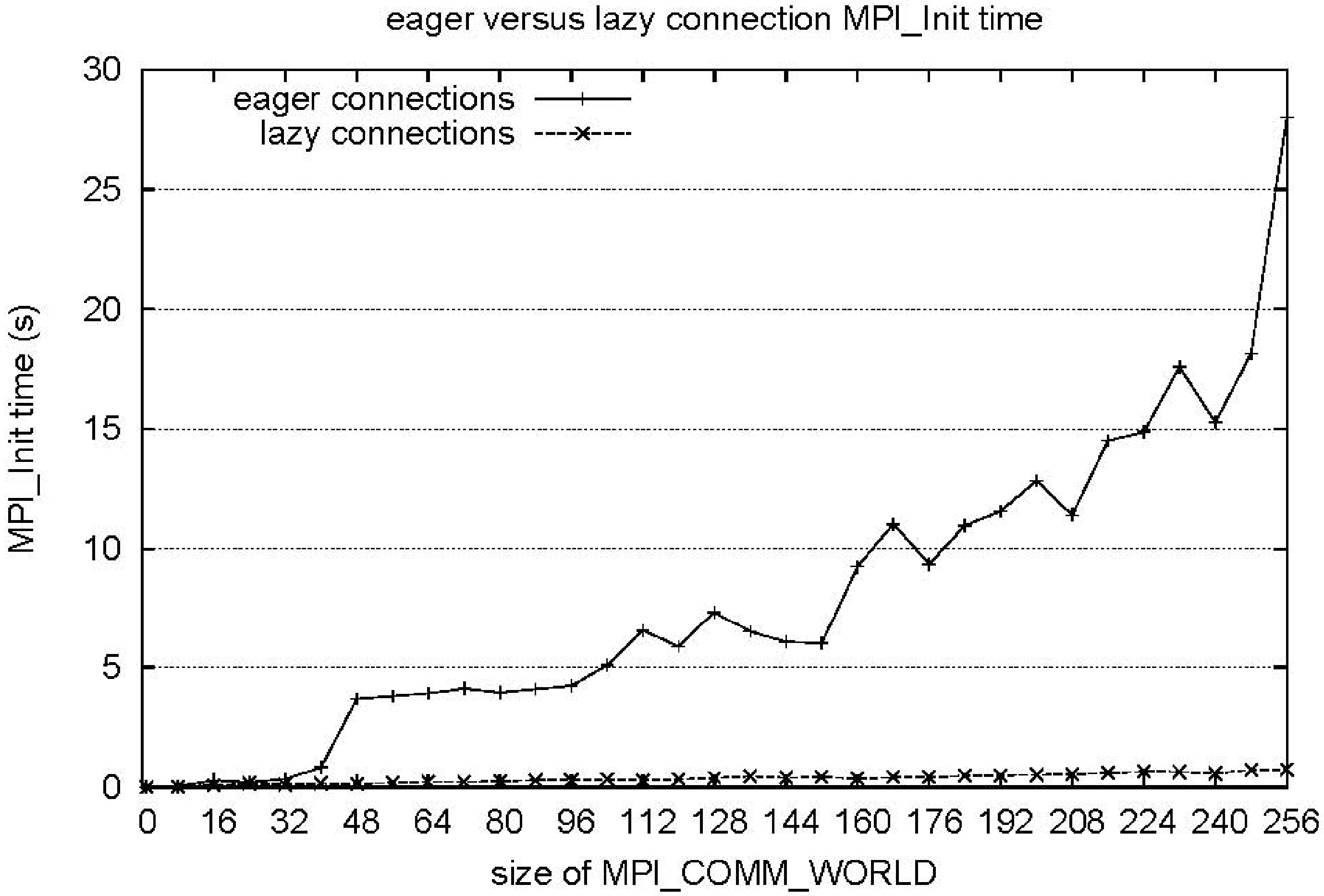
- NEK5000 code failed on BG/P at large scale because MPI ran out of communicator memory. We fixed the problem by using a fixed buffer pool within MPI and provided a patch to IBM.

MPI Memory Usage on BG/P after 32 calls to MPI_Comm_dup



- Using a buffer pool enables all collective optimizations and takes up only a small amount of memory

Scalability of MPI_Init



- Cluster with 8 cores per node. TCP/IP across nodes
- Setting up all connections at Init time is too expensive at large scale; must be done on demand as needed

Scalable Algorithms for Collective Communication

- MPI implementations typically use
  - O(lg p) algorithms for short messages (binomial tree)
  - O(m) algorithms, where m=message size, for large messages
    - E.g., bcast implemented as scatter + allgather
- O(lg p) algorithms can still be used on a million processors for short messages
- However, O(m) algorithms for large messages may not scale, as the message size in the allgather phase can get very small
  - E.g., for a 1 MB bcast on a million processes, the allgather phase involves 1 byte messages
- Hybrid algorithms that do logarithmic bcast to a subset of nodes, followed by scatter/allgather may be needed
- Topology-aware pipelined algorithms may be needed
- Use network hardware for broadcast/combine

Enabling Hybrid Programming

- MPI is good at moving data between address spaces
- Within an address space, MPI can interoperate with other "shared memory" programming models
- Useful on future machines that will have limited memory per core
- (MPI + X) Model: MPI across address spaces, X within an address space
- Examples:
  - MPI + OpenMP
  - MPI + UPC/CAF (here UPC/CAF address space could span multiple nodes)
  - MPI + CUDA/OpenCL on GPU-accelerated systems
- Precise thread-safety semantics of MPI enable such hybrid models

- MPI Forum is exploring further enhancements to MPI to support efficient hybrid programming

MPI-3 Hybrid Proposal on Endpoints

- In MPI today, each process has one communication endpoint (rank in MPI_COMM_WORLD)
- Multiple threads communicate through that one endpoint, requiring the implementation to do use locks etc., which are expensive
- This proposal (originally by Marc Snir) allows a process to have multiple endpoints
- Threads within a process attach to different endpoints and communicate through those endpoints as if they are separate ranks
- The MPI implementation can avoid using locks if each thread communicates on a separate endpoint

Fewer Performance Surprises

- Sometimes we hear…

“I replaced

MPI_Allreduce

by

MPI_Reduce + MPI_Bcast

And got better results…”

Should not happen…

Or…

"I replaced

MPI_Send(n)

by

MPI_Send(n/k) + MPI_Send(n/k) + … + MPI_Send(n/k)

And got better results…"

Well, should probably not happen…

Or…

“I replaced

MPI_Bcast(n)

by

<this homemade algorithm with MPI_Send(n) and MPI_Recv(n)>

And got better results…”

Should not happen…

Self-Consistent MPI Performance Guidelines

- Although MPI is portable, there is a lot of performance variability among MPI implementations
  - Lots of performance surprises

- We (Traff, Gropp, Thakur) have defined some common-sense performance guidelines for MPI
  - *“Self-Consistent MPI Performance Guidelines”, IEEE TPDS, 2010*

- Tools could be written to check for these requirements

General Principles

If there is an obvious way – intended by the MPI standard – of improving communication time,

⇨

a sound MPI implementation should do so!

- And not the user!

Sample Requirements

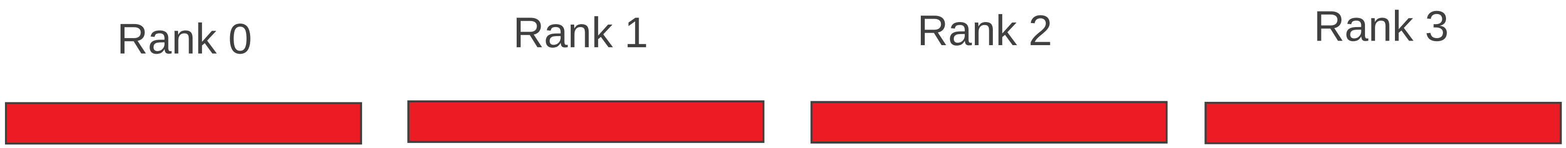
- Subdividing messages into multiple messages should not reduce the communication time
  - MPI_Send(1500 bytes) <= MPI_Send(750 bytes) + MPI_Send(750 bytes)

- Replacing an MPI function with a similar function that provides additional semantic guarantees should not reduce the communication time
  - MPI_Send <= MPI_Ssend

- Replacing a specific MPI operation by a more general operation by which the same functionality can be expressed should not reduce communication time
  - MPI_Scatter <= MPI_Bcast

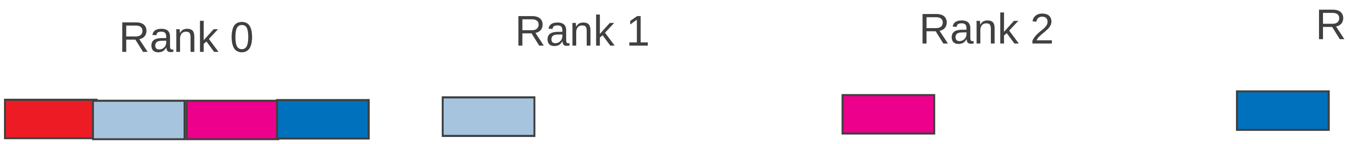
Example: Broadcast vs Scatter

Broadcast



Rank 0 Rank 1 Rank 2 Rank 3

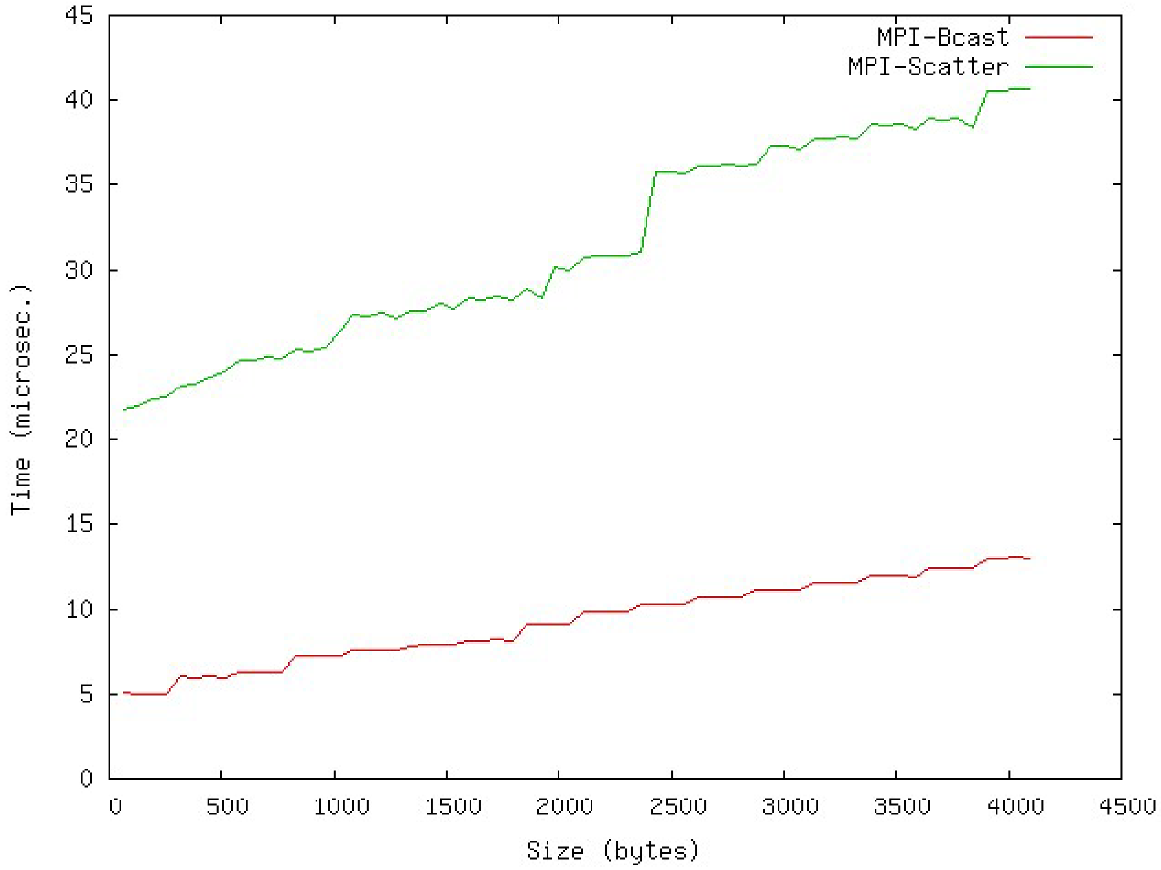
Scatter



Rank 0 Rank 1 Rank 2 Rank 3

- Scatter should be faster (or at least no slower) than broadcast

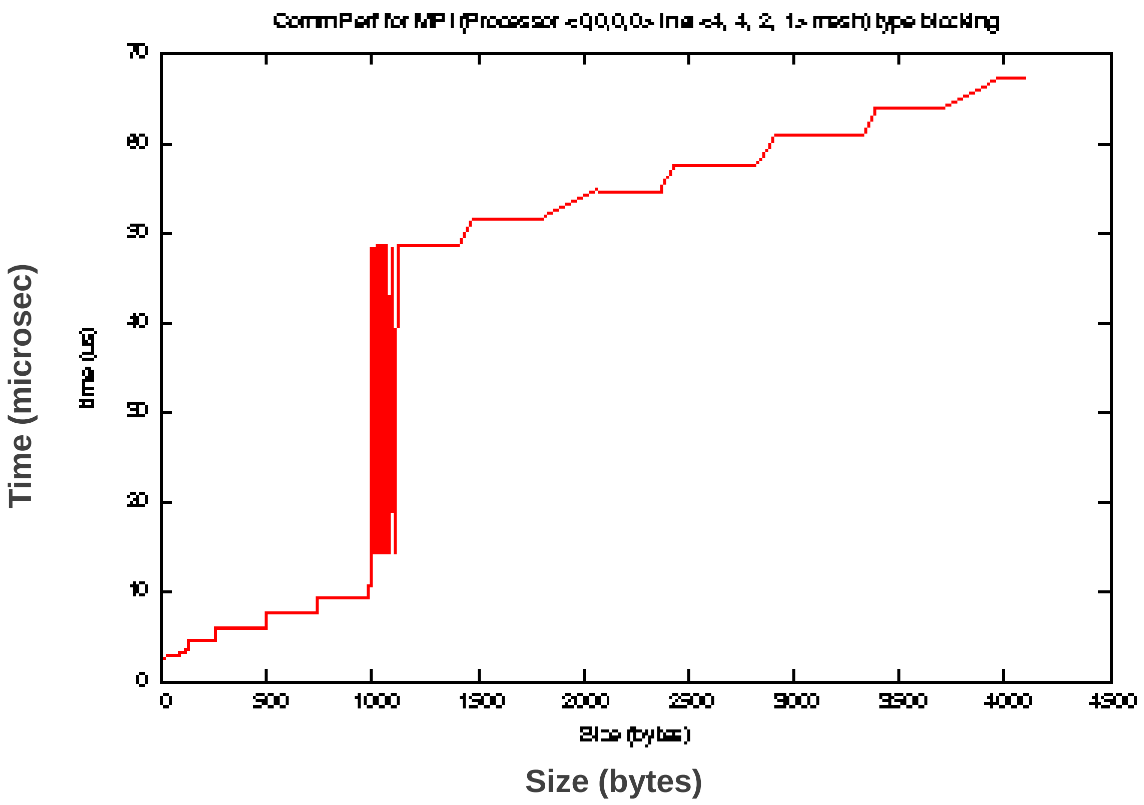
MPI_Bcast vs MPI_Scatter



64 processes

- On BG/P, scatter is 3-4 times slower than broadcast
- Broadcast has been optimized using hardware, scatter hasn't

Eager vs Rendezvous Messages



- Large jump in time when message delivery switches from eager to rendezvous
- Sending 2 750-byte messages is faster than 1 1500-byte message

Recent Efforts of the MPI Forum

MPI Standard Timeline

- MPI-1 (1994)
  - Basic point-to-point communication, collectives, datatypes, etc
- MPI-2 (1997)
  - Added parallel I/O, RMA, dynamic processes, C++ bindings, etc
- ---- Stable for 10 years ----
- MPI-2.1 (2008)
  - Minor clarifications and bug fixes to MPI-2
- MPI-2.2 (2009)
  - Today’s official standard
  - Small updates and additions to MPI 2.1. Backward compatible
- MPI-3 (in progress, expected late 2011)
  - Major new features and additions to extend MPI to exascale
  - Organized into several working groups

New Features being considered in MPI-3

- **Note: All these are still under discussion in the Forum and not final**

- Support for hybrid programming (Lead: Pavan Balaji, Argonne)
  - Extend MPI to allow multiple communication endpoints per process
  - Helper threads: application sharing threads with the implementation

- Improved RMA (Leads: Bill Gropp, UIUC, and Rajeev Thakur, Argonne)
  - Fix the limitations of MPI-2 RMA
  - New compare-and-swap, fetch-and-add functions
  - Collective window memory allocation
  - Test for completion of individual operations
  - Others…

New Features being considered in MPI-3

- New collectives (Lead: Torsten Hoefler, UIUC)
  - Nonblocking collectives already voted in (MPI_Ibcast, MPI_Ireduce, etc)
  - Sparse, neighborhood collectives being considered as alternatives to irregular collectives that take vector arguments

- Fault tolerance (Lead: Rich Graham, Oak Ridge)
  - Detecting when a process has failed; agreeing that a process has failed
  - Rebuilding communicator when a process fails or allowing it to continue in a degraded state
  - Timeouts for dynamic processes (connect-accept)
  - Piggybacking messages to enable application-level fault tolerance

New Features being considered in MPI-3

- Fortran 2008 bindings (Lead: Craig Rasmussen, LANL)
  - Full and better quality argument checking with individual handles
  - Support for choice arguments, similar to (void *) in C
  - Passing array subsections to nonblocking functions
  - Many other issues
- Better support for Tools (Lead: Martin Schulz, LLNL)
  - MPIT performance interface to query performance information internal to an implementation
  - Standardizing an interface for parallel debuggers

What are we doing in MPICH2

Goals of the MPICH2 project

- Be the MPI implementation of choice for the highest-end parallel machines
  - 7 of the top 10 machines in the June 2010 Top500 list use MPICH2-based implementations
- Carry out the research and development needed to scale MPI to exascale
  - Optimizations to reduce memory consumption
  - Fault tolerance
  - Efficient multithreaded support for hybrid programming
  - Performance scalability
- Work with the MPI Forum on standardization and early prototyping of new features

MPICH2 collaboration with vendors

- Enable vendors to provide high-performance MPI implementations on the leading machines of the future

- Collaboration with IBM on MPI for the Blue Gene/Q
  - Aggressive multithreaded optimizations for high concurrent message rates
  - Recent publications in Cluster 2010 and EuroMPI 2010

- Collaboration with Cray for MPI on their next-generation interconnect (Gemini)

- Collaboration with UIUC on MPICH2 over LAPI for Blue Waters

- Continued collaboration with Intel, Microsoft, and Ohio State (MVAPICH)

Conclusions

- MPI has succeeded because
  - features are orthogonal (complexity is the product of the number of *features*, not routines)
  - complex programs are no harder than easy ones
  - open process for defining MPI led to a solid design
  - programmer can control memory motion and program for locality (critical in high-performance computing)
  - precise thread-safety specification has enabled hybrid programming

- MPI is ready for scaling to extreme scale systems with millions of cores barring a few issues that can be (and are being) fixed by the MPI Forum and by MPI implementations