Future Directions in MPI

Rajeev Thakur

Mathematics and Computer Science Division

Argonne National Laboratory

MPI on the Largest Machines Today

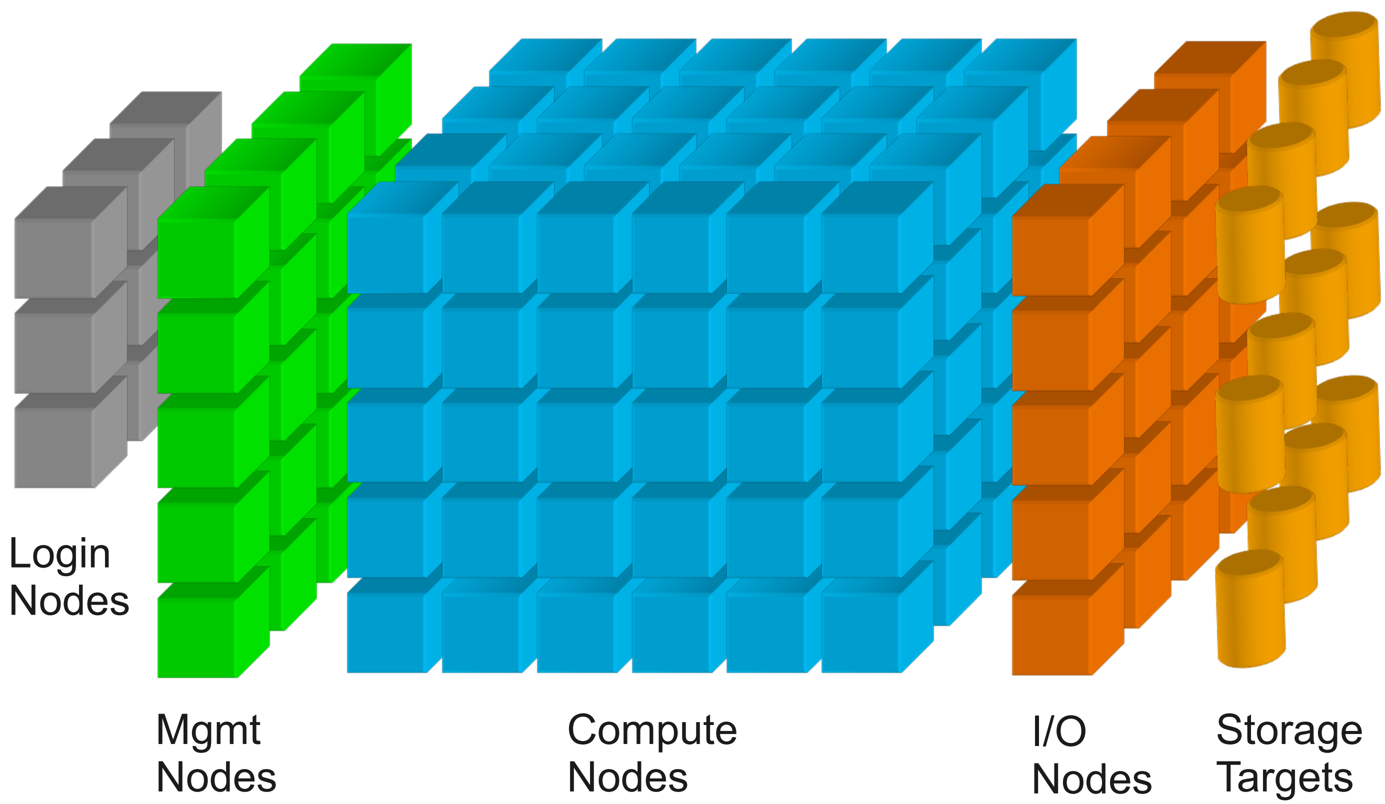
- Systems with the largest core counts in June 2010 Top500 list

Jülich 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

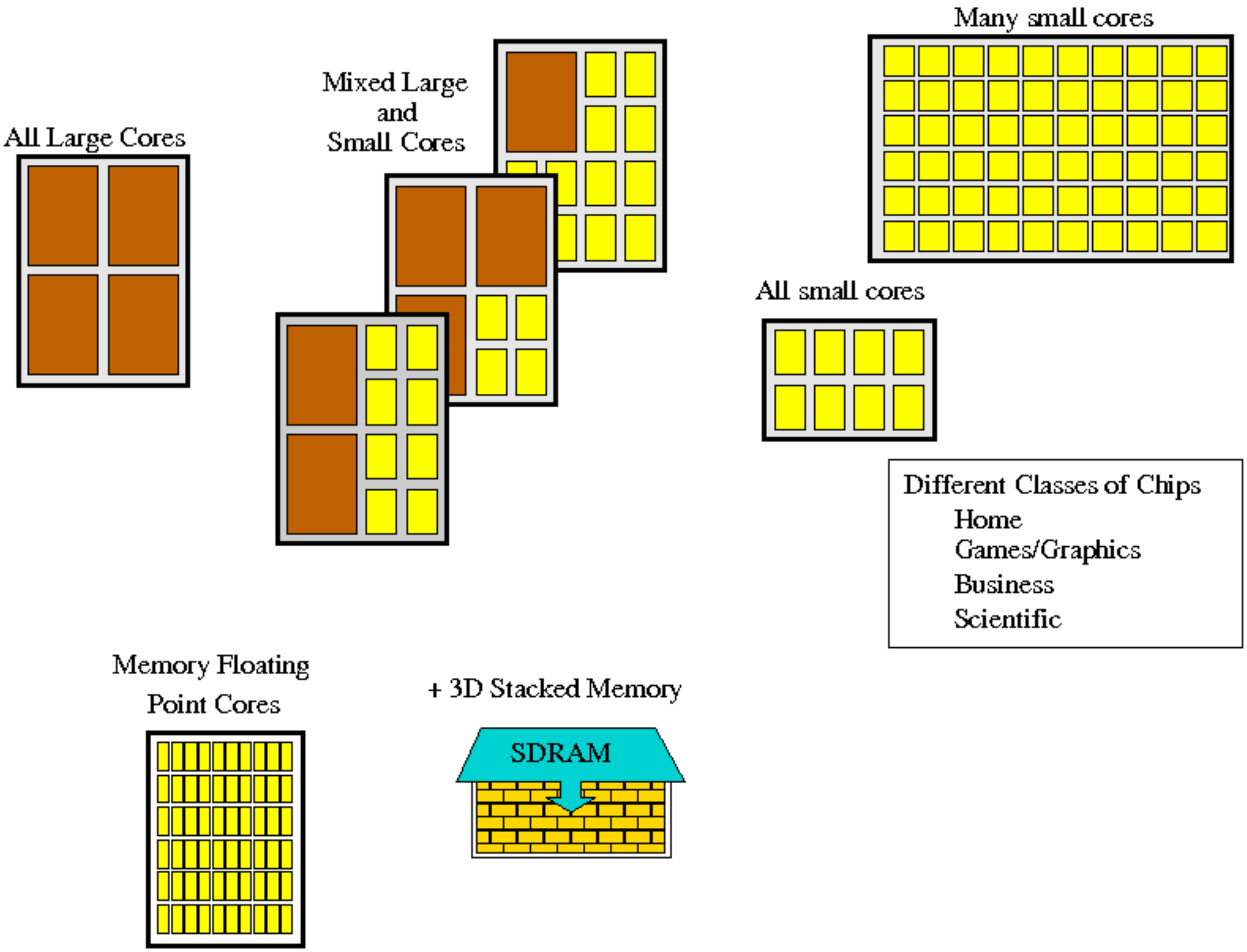
- MPI already runs successfully on these systems
- In a couple of years, we will have systems with more than a million cores
- For example, in 2012, the Sequoia machine at Livermore will be an IBM Blue Gene/Q with ~1.6 million cores
  - More than 5 times the size of today's largest machine

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



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

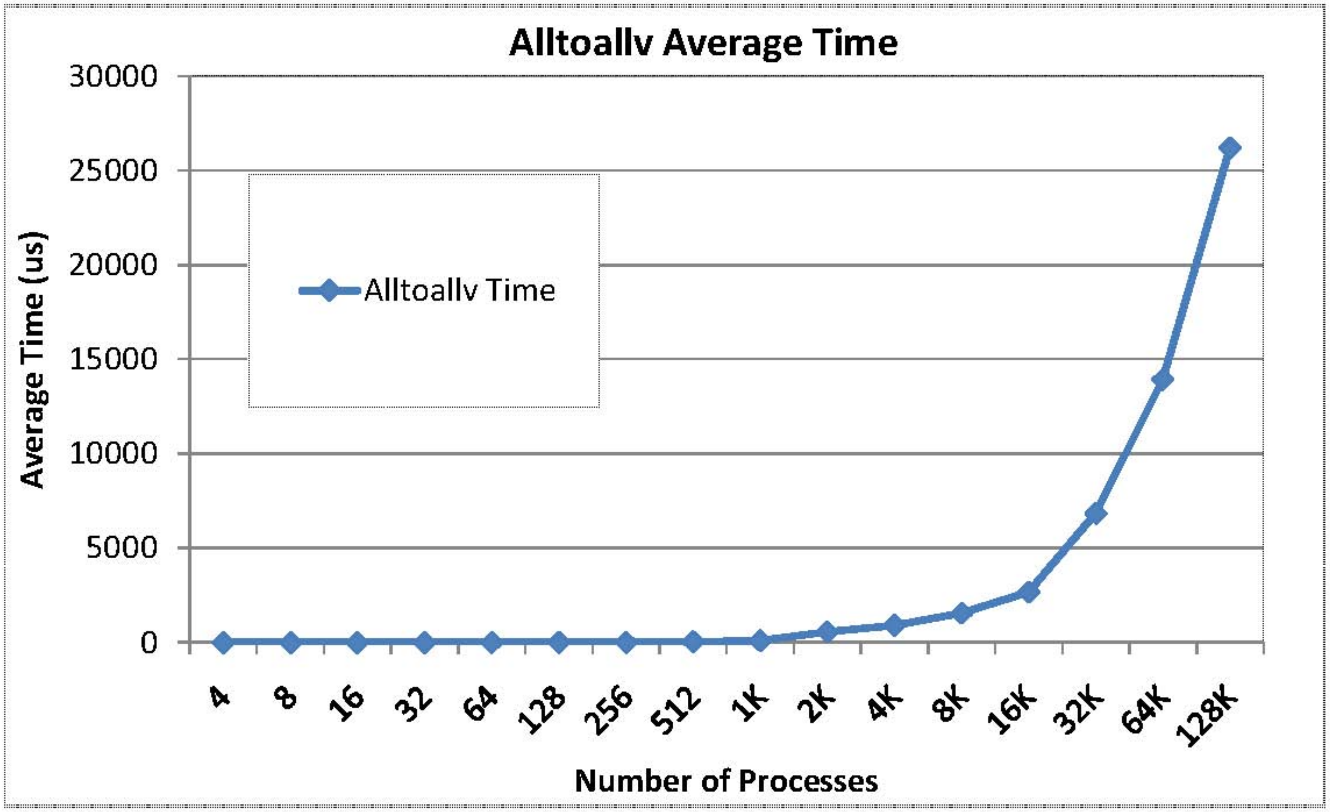
- Performance, memory consumption, fault tolerance
- A nonscalable MPI function is one whose time or memory consumption per process increase linearly (or worse) with the total number of processes
- For example
  - If memory consumption of MPI_Comm_dup increases linearly with the no. of processes, it 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

Examples of 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.

Other 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
  - But existing applications must switch to the new interface

- 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

Fault Tolerance

- Large component counts will result in frequent failures
- Greater resilience needed from all components of the stack
  - Hardware, system software, MPI library, applications
- MPI already allows implementations to return an error code and remain alive, but more support is needed
- Various research projects have explored fault tolerance in MPI
  - MPICH-V, FT-MPI, HARNESS, ABARIS, and others
- Supported to various degrees in Open MPI and MPICH2
- CiFTS project aims to coordinate fault tolerance among various system software components, including MPI

- Fault tolerance working group in the MPI Forum is exploring additional fault tolerance features for MPI-3 (more later)

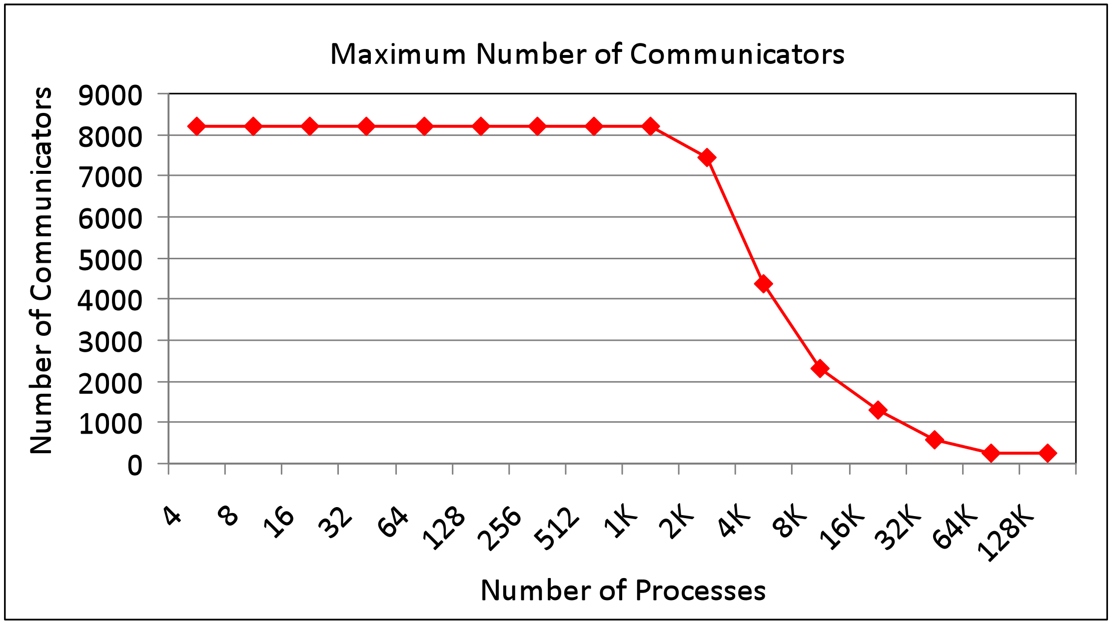
Requirements of a message-passing library at extreme scale

- 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

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

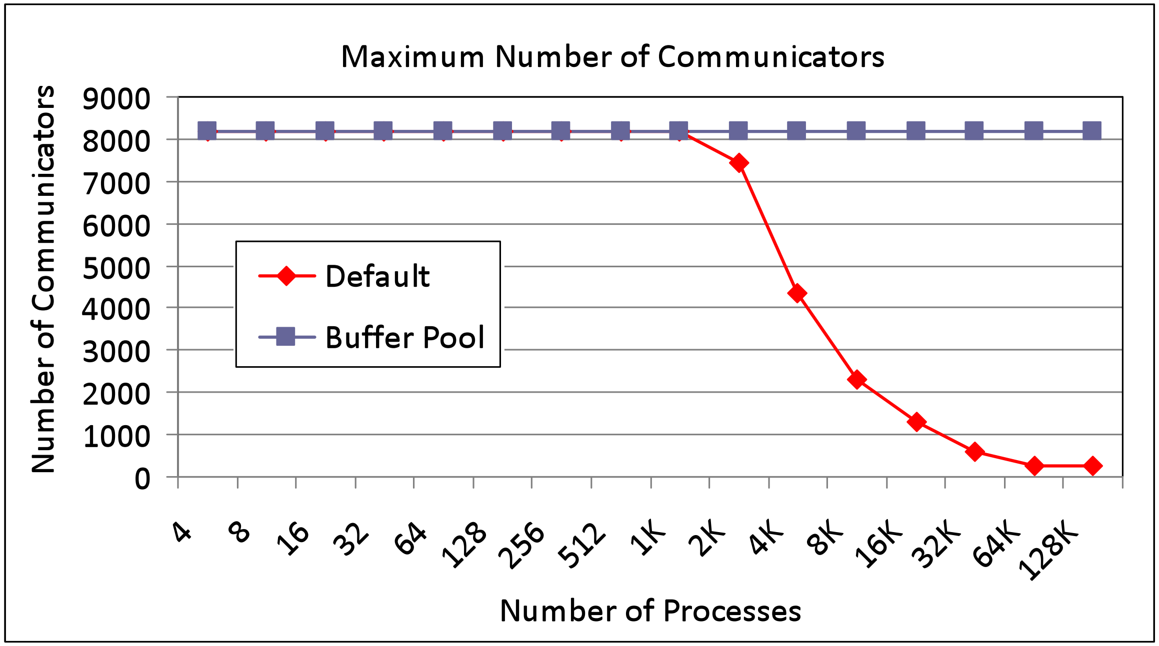
Example of a Memory Consumption Problem

- NEK5000 code initially failed on 8K processes on IBM BG/P because the MPI implementation ran out of memory in MPI_Comm_dup
- IBM's MPI was allocating O(nprocs) memory in each call to MPI_Comm_dup to store some process mapping info for optimizing collectives
- After some 40-50 calls to MPI_Comm_dup, NEK5000 failed



Communicator Memory Consumption Fixed

- Looking at the source code, we found that IBM's MPI 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 and enabled NEK5000 to run at full scale



Example of a Performance Scalability Problem

- A user (Nick Romero) on our BG/P complained that MPI_Comm_split was scaling poorly
- As he *doubled* the number of processes, the time taken by MPI_Comm_split *quadrupled*

16,384 procs	1.5 sec
32,768 procs	6.3 sec
65,536 procs	25.3 sec
131,072 procs	101.2 sec

- Clearly something O(p^2) going on

The Problem and the Fix

- MPI_Comm_split does an allgather of the colors and keys from all processes, followed by a local sort of the keys for the same color
- In the case where all ranks pass the same color, the data set to be sorted is of size *p*
- The local sort used a simple bubble sort algorithm, which is O(p^2)
  - The code did have a FIXME comment acknowledging this
- Simply switching the local sort to use quicksort, which is O(*plgp*), fixed the problem

	OLD	NEW
16,384 procs	1.5 sec	0.105 sec
32,768 procs	6.3 sec	0.126 sec
65,536 procs	25.3 sec	0.168 sec
131,072 procs	101.2 sec	0.255 sec

- At this scale, there is a big difference between p^2 and *plgp*!

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!

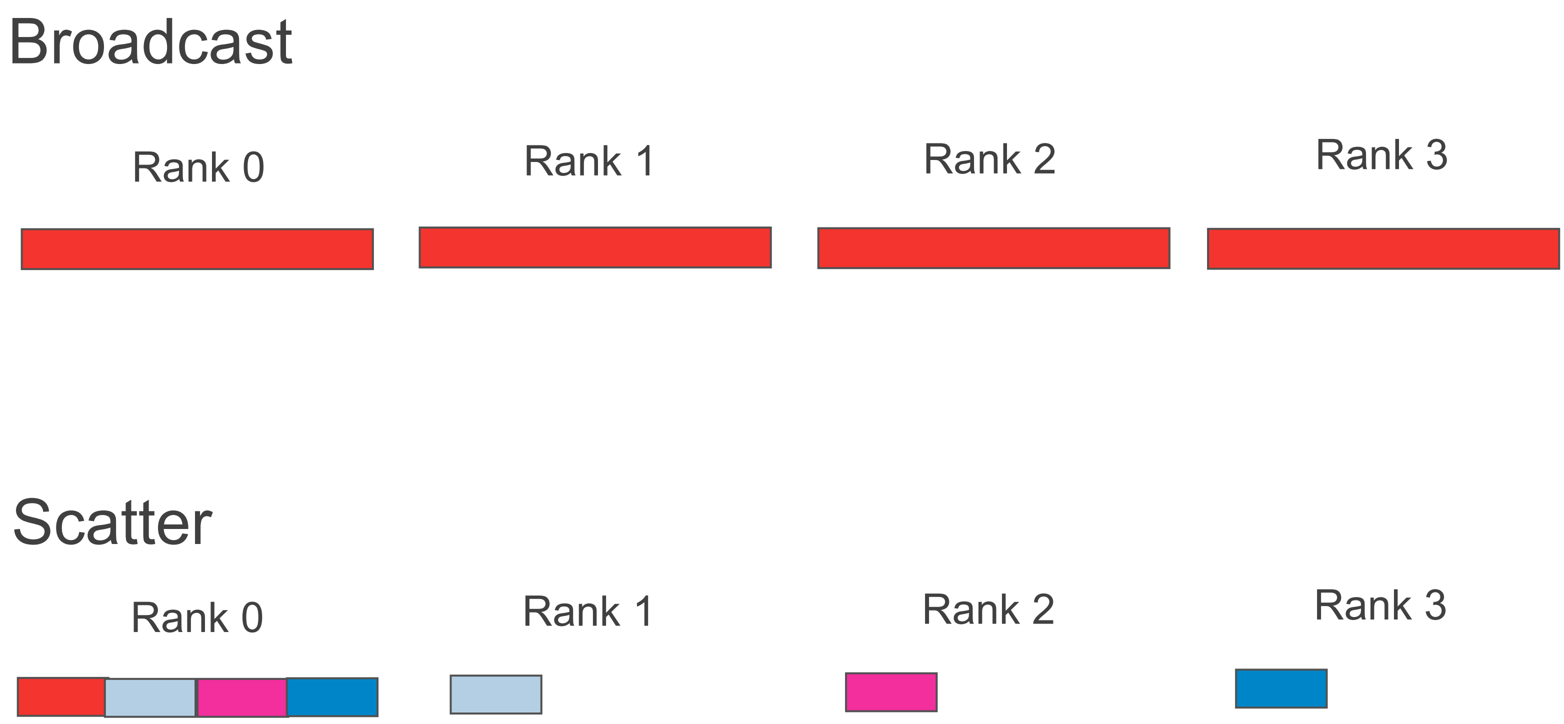
Sample Requirements

- 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

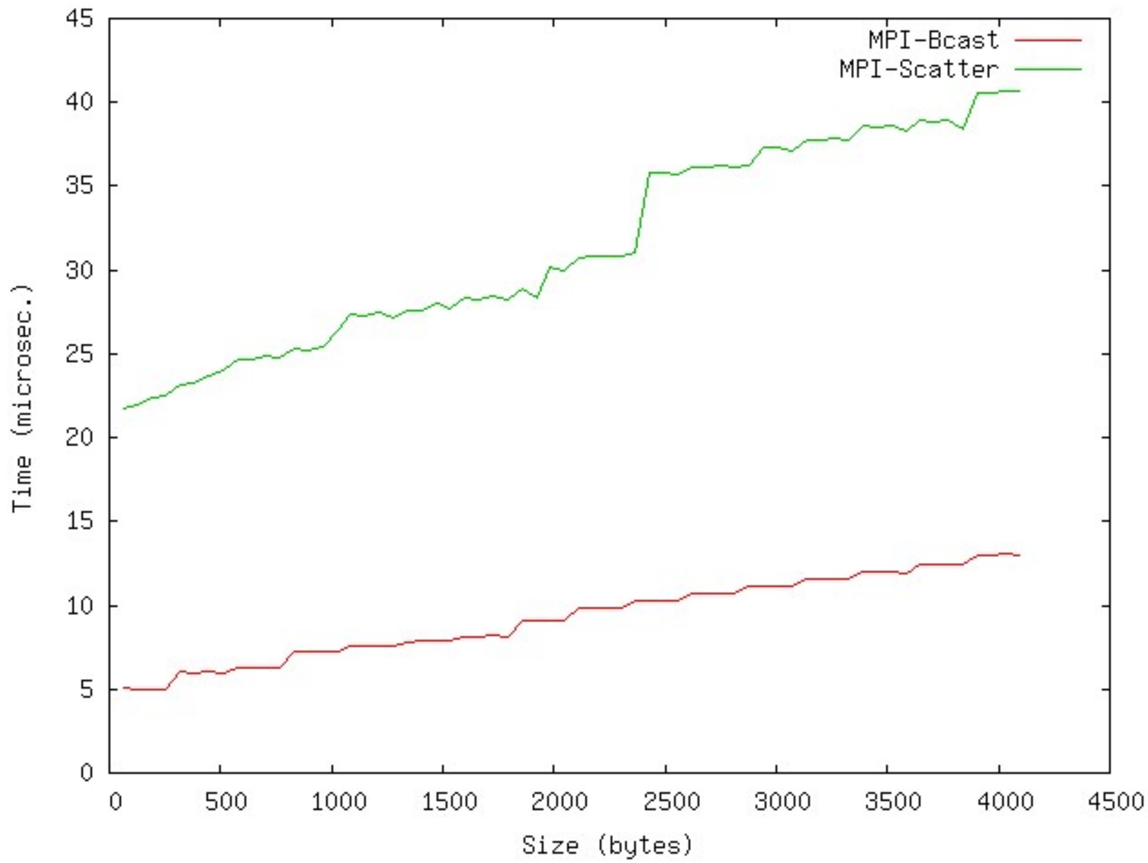
- Replacing a specific MPI operation by a more general operation by which the same functionality can be expressed should not reduce communication time
  - MPI_Scatter <= MPI_Bcast

Example: Broadcast vs Scatter



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

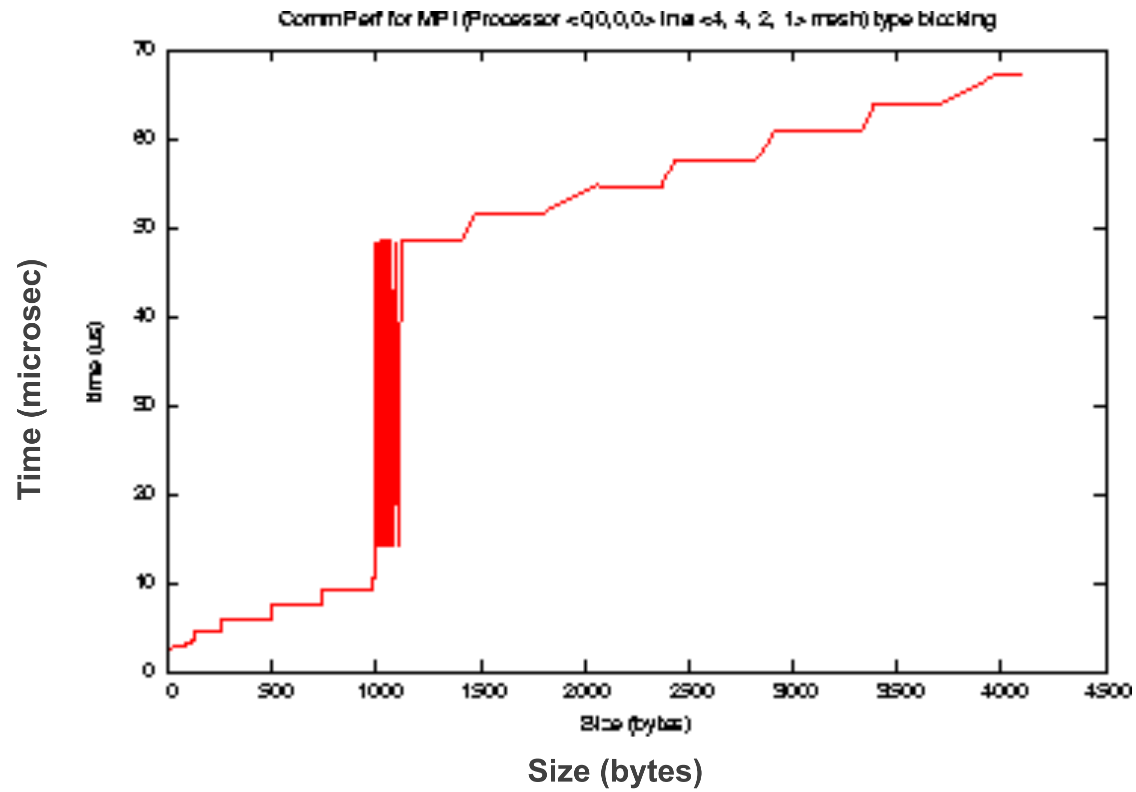
MPI_Bcast vs MPI_Scatter



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
  - Window representing entire memory
  - Query function to determine whether system is cache coherent (for reduced synchronization requirement)
  - 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
  - Others

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

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