Application-Bypass Reduction for Large-Scale Clusters

Adam Wagner, Darius Buntinas, D.K. Panda
Network-Based Computing Lab
The Ohio State University

Ron Brightwell
Scalable Computing Systems Group
Sandia National Laboratories

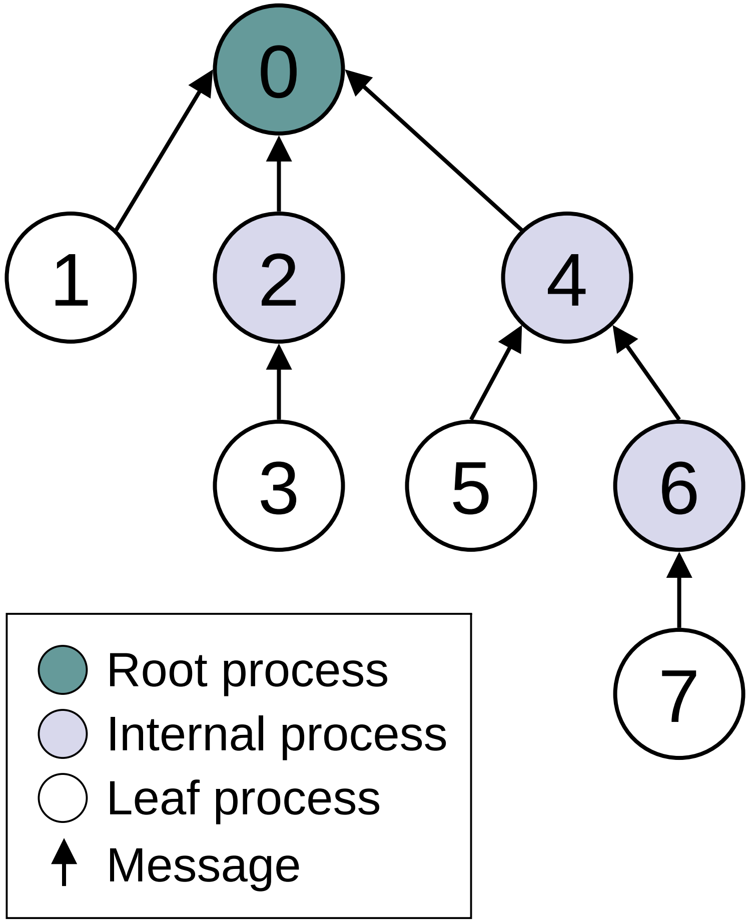
Overview

- Background & Motivation
- Design Challenges
- Our Implementation
- Experimental Results
- Conclusions & Future Work

Process Skew

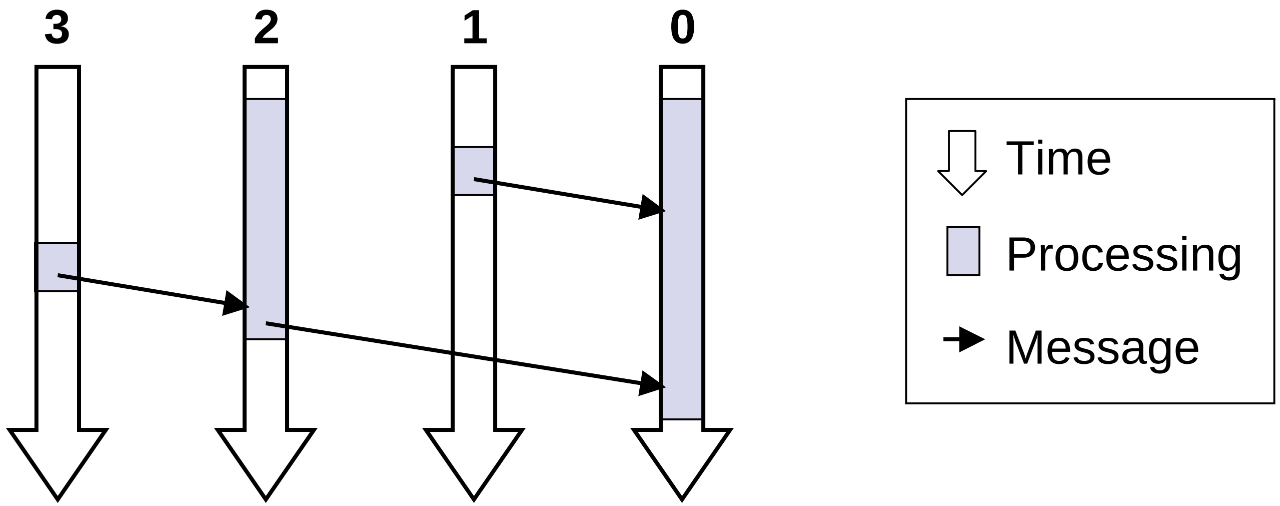
- Processes in a parallel application may become unsynchronized or *skewed* over time
- Contributing factors include:
  - Heterogeneous systems
  - Varying communication latencies
  - Interrupts and contention at individual nodes
- Potential for skew increases with size of cluster
- Collective operations especially vulnerable due to communication dependencies between processes

Example Reduction Operation in MPICH



- Each process calls MPI_Reduce function
- Processes are organized into logical binomial tree
- Process must complete receive from each child before initiating send to parent
- Under conditions of skew, parent processes may wait idly on late children

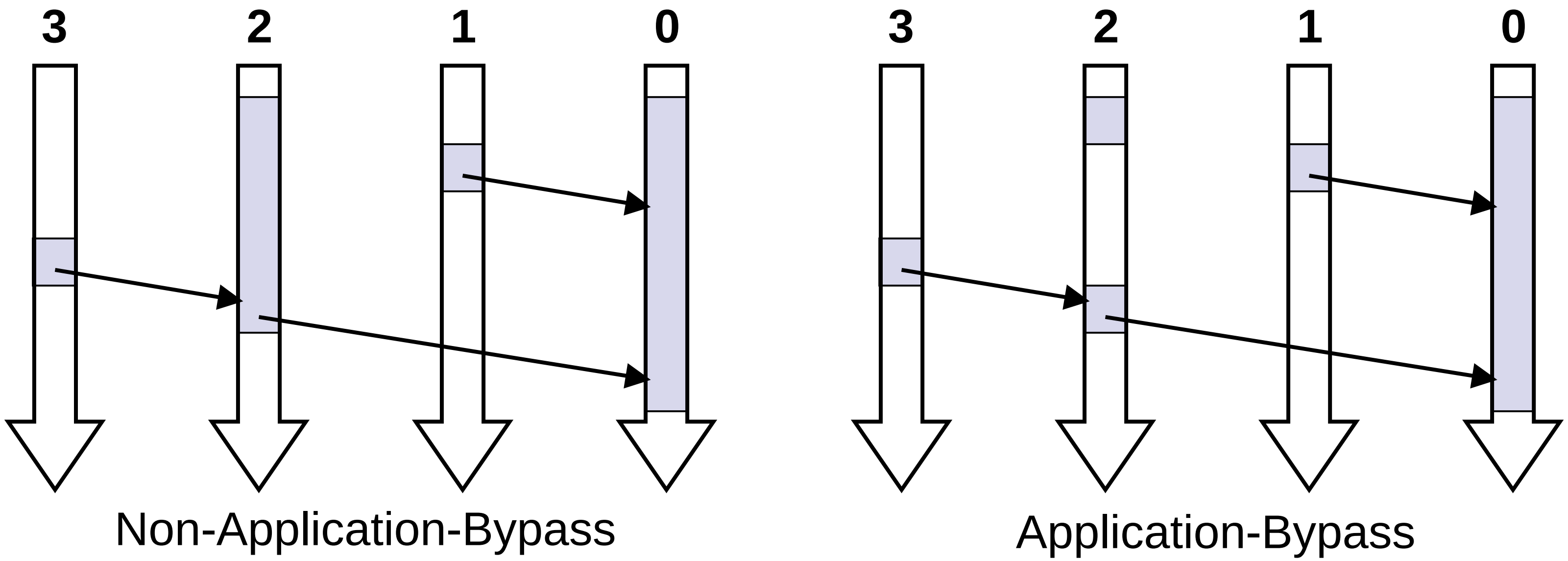
Example Reduction Timeline



Non-Application-Bypass

- Processes are skewed with process three starting last
- Process two must wait idly for message from process three
- All processing is synchronous within MPI_Reduce

Example Reduction Timeline



- Only initial processing is associated with call to MPI_Reduce
- Message from process three is handled later asynchronously
- Time in between can be used for other processing

Application Bypass

- Split operation into synchronous and asynchronous components
- Allow communication to make progress independently of application
- Under conditions of process skew, application-bypass:
  - Reduces implicit synchronization between processes
  - Reduces the amount of time processes spent waiting unnecessarily on each other
  - Reduces CPU utilization associated with communication
  - Improves potential for overlap of communication and computation

Overview

- Background & Motivation
- **Design Challenges**
- Our Implementation
- Experimental Results
- Conclusions & Future Work

Design Challenges

- Communication progress mechanism
- Maintenance of intermediate state
- Message queuing infrastructure
- Reducing frequency of late messages

Communication Progress Mechanism

- Need mechanism to trigger asynchronous processing when late messages arrive
- Want to isolate associated overhead to avoid impacting unrelated communication
- Want to maximize resources available for computation

Communication Progress Mechanism – Options

- Polling at MPICH level
  - Check for new messages at library calls
  - Relies on application to make progress
- Dedicated communication thread
  - Avoids overhead of interrupts
  - Constant CPU overhead associated with polling
  - Requires sharing of data structures
- Single-threaded interrupt-driven approach
  - Interrupt overhead only incurred when necessary
  - Maximizes available computation resources
  - Avoids complications associated with multithreading

Maintenance of Intermediate State

- Must maintain state to bridge gap between synchronous and asynchronous components
  - Need to keep track of running result of operation
  - Need a way to know when all children have been processed and result may be sent to parent
  - Need to maintain identity of parent (varies based on root of operation)
  - Need a way to differentiate early and late messages
  - Need to be able to match late messages to appropriate reduction instance

Message Queuing Infrastructure

- MPICH maintains two basic queues:
  - Receive – Receives waiting for incoming messages
  - Unexpected – Early messages buffered for later use
- MPICH also offers non-blocking send and receive primitives (asynchronous semantics)
- Can we re-use existing infrastructure?
  - Want to avoid impacting non-application-bypass communication
  - Still need a way to accommodate maintenance of application-bypass state

Message Queuing Infrastructure - Options

- Re-use MPICH infrastructure
  - Avoids re-inventing the wheel
  - Risks impacting non-application-bypass communication
  - Need to post buffers for all asynchronous receives
- Design separate infrastructure
  - Can isolate impact to unrelated communication
  - Can process receives without requiring extra buffers
  - Can integrate queuing with state maintenance
  - Actually less complex than abusing existing infrastructure

Reducing Frequency of Late Messages

- Late messages may incur interrupt overhead
- Can we reduce the number of late messages?
  - Process as many messages as possible synchronously
  - Wait for brief time and then re-check for outstanding messages before exiting MPI_Reduce
- Issue is how long to wait:
  - Too short – Introduce extra synchronous latency and still miss delayed messages
  - Too long – May wait longer than actually necessary depending on implementation
- Tried simple scheme based on log of system size
- Planning to investigate further

Overview

- Background & Motivation
- Design Challenges
- **Our Implementation**
- Experimental Results
- Conclusions & Future Work

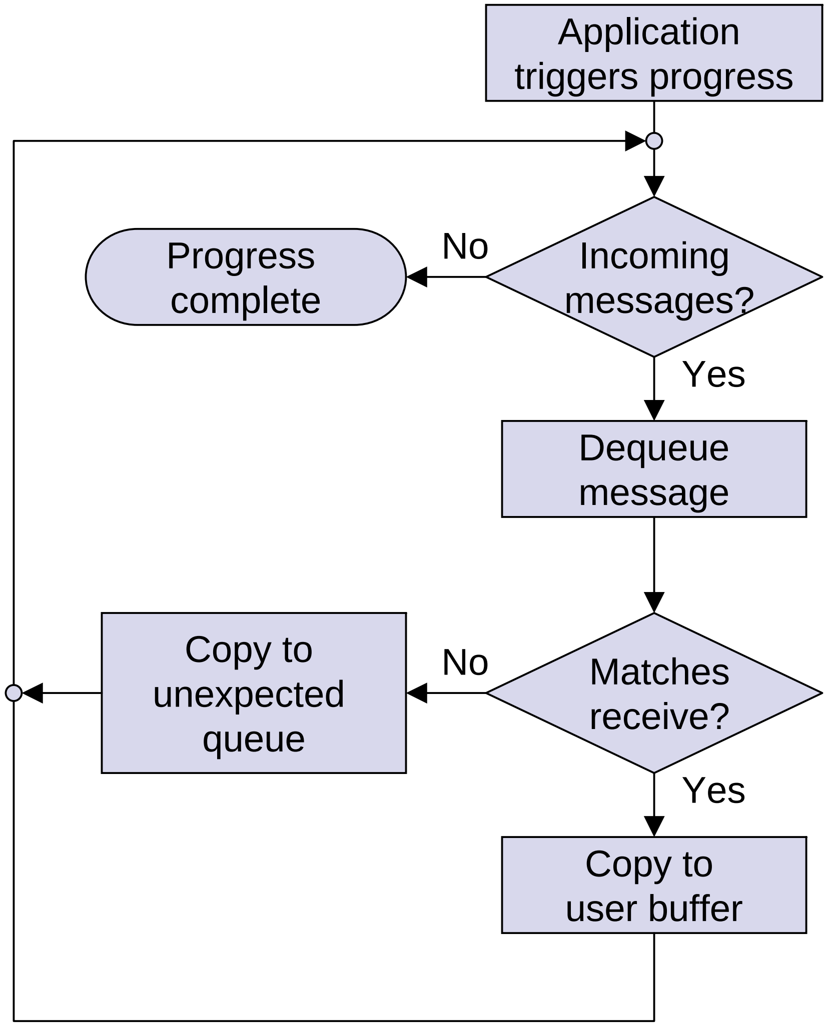
MPICH over GM

- GM
  - User-level message-passing subsystem for Myrinet networks
  - Myrinet network network interface cards (NICs) offer customizable firmware
- MPICH
  - Reference implementation of Message Passing Interface (MPI) standard
  - Port over GM developed by Myricom

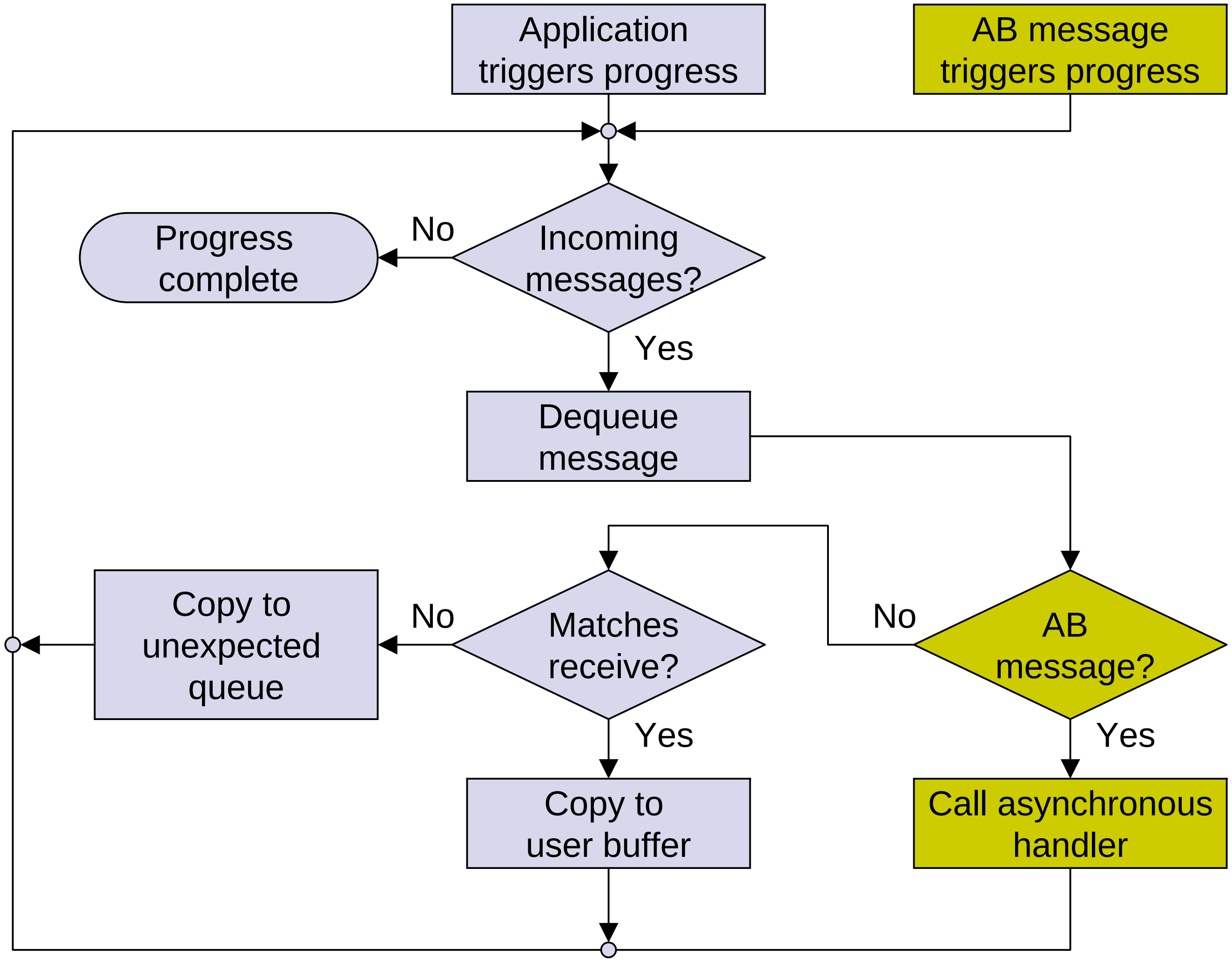
Mechanism to Trigger Asynchronous Processing

- Added new packet type for use when sending messages related to application-bypass reduction
- Modified NIC-based control program to generate signal on arrival of such messages
- Added signal handler in MPICH layer to trigger communication progress
- Modified GM layer to provide capability to enable and disable signals at runtime:
  - Disabled during synchronous processing
  - Enabled only when asynchronous receives pending
  - Signals ignored if already making progress

Progress in MPICH



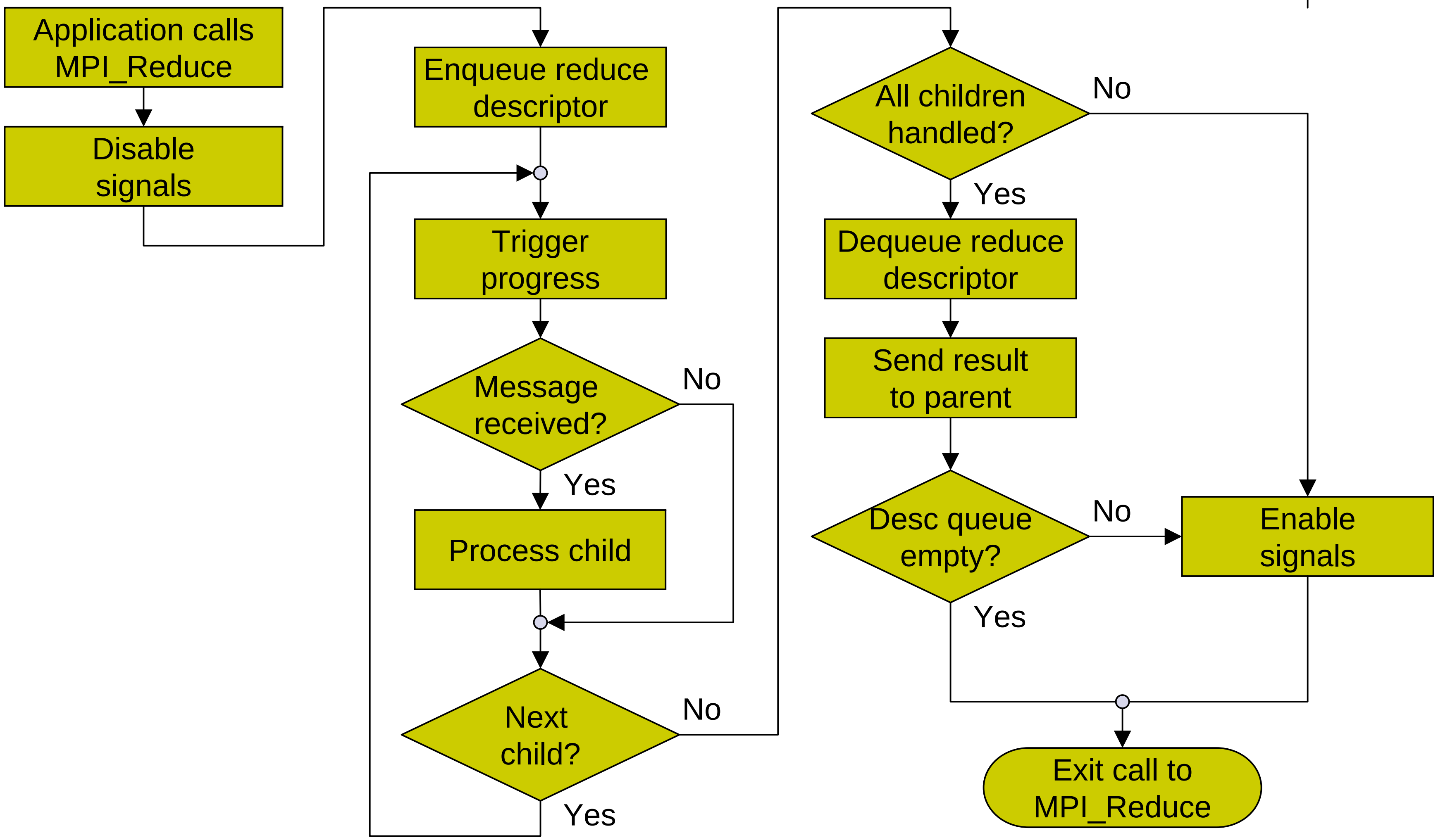
Progress in MPICH



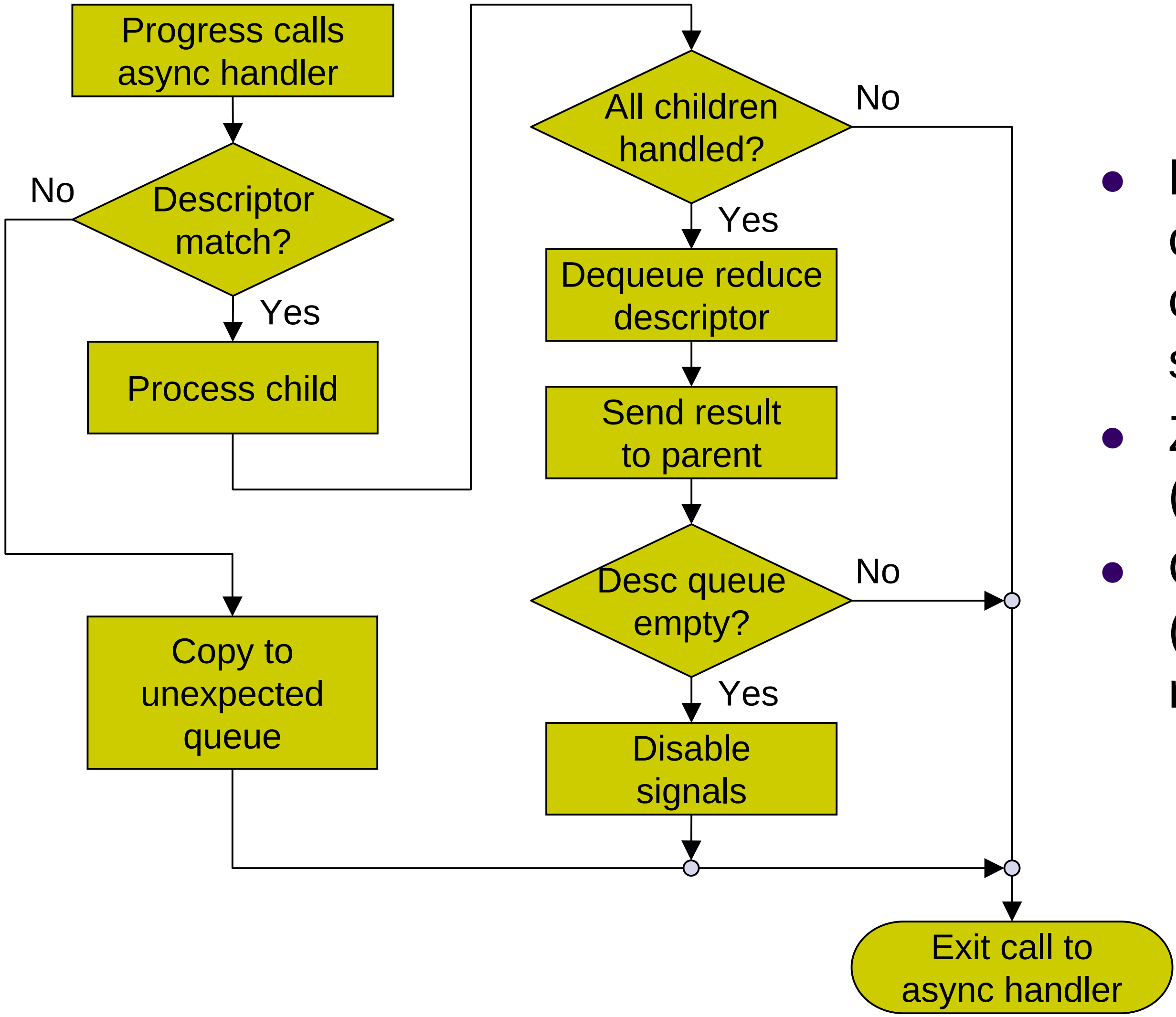
Infrastructure to Support Asynchronous Processing

- Custom unexpected queue:
  - Allows messages associated with reduction to be managed separately from other messages
  - Reduces number of message copies
- Reduce descriptor queue:
  - Supports management of reduction instances that are being processed asynchronously
  - Each descriptor maintains state for a reduction instance:
    - Running result of reduction
    - Operator associated with reduction
    - Identity of parent for sending final result
    - List of children from which receives are still pending

Synchronous Processing



Asynchronous Processing



- Message matches oldest descriptor with child that matches sender
- Zero copies for late (expected) messages
- One copy for early (unexpected) messages

Overview

- Background & Motivation
- Design Challenges
- Our Implementation
- **Experimental Results**
- Conclusions & Future Work

Experimental Setup

- 16-node cluster of quad-SMP 700-MHz Pentium-III nodes with 66-MHz/64-bit PCI and 1-GB RAM
- Myrinet PCI64B NICs with 133 MHz Lanai 9.1 processors and 2-MB RAM
- 16 ports of 32-port Myrinet-2000 switch
- MPICH 1.2.4..8a over GM 1.5.2.1
- All tests performed with one process per node

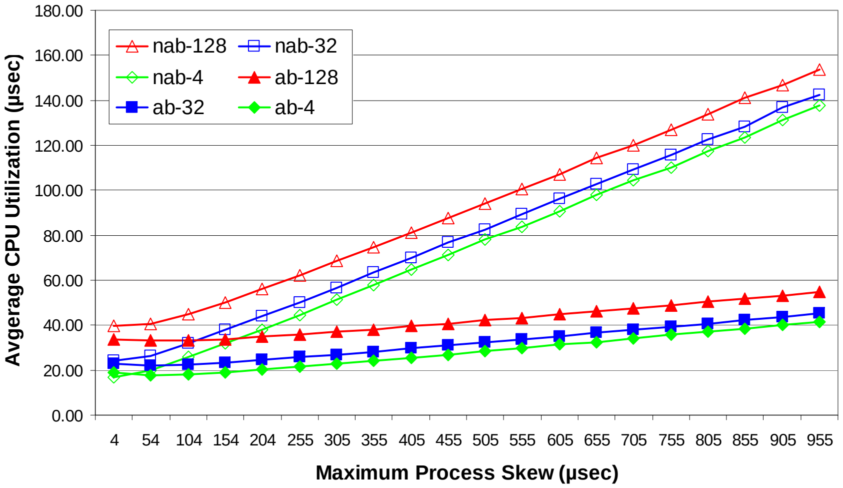
Microbenchmarks

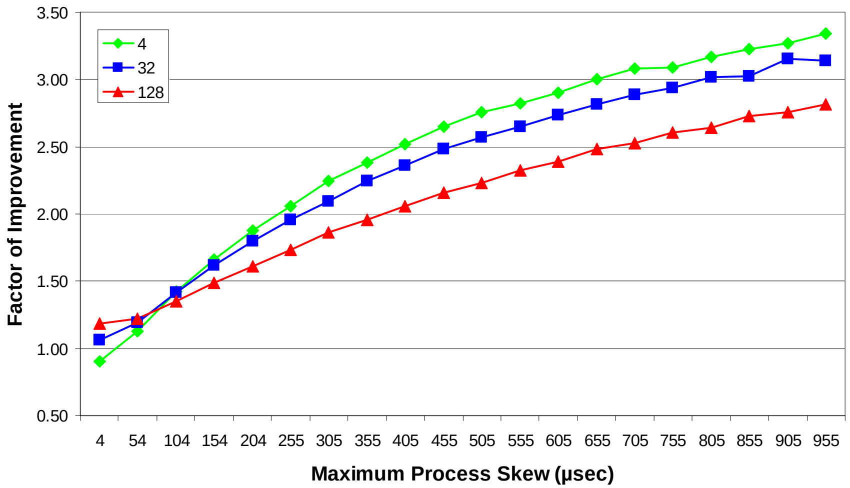
- Two microbenchmarks:
  - CPU utilization
    - Measures CPU utilization associated with reduction operation under varying amounts of process skew
    - Our target metric
  - Latency
    - Measures total time to perform a reduction operation in the absence of process skew
    - Something to keep an eye on
- Two scenarios for CPU utilization:
  - With skew – Common case and our focus
  - Without skew – Unrealistic for large-scale clusters

CPU Utilization Microbenchmark

- Artificial skew introduced as busy-loop delay before starting reduction operation
- For a given maximum amount of skew:
  - Start timing
  - Impose random skew delay between zero and maximum
  - Call MPI_Reduce
  - Impose catch-up delay
  - Stop timing and subtract both delays from measured time
  - Repeat 10,000 times and take average across all nodes
- Catch-up delay is long enough to capture any asynchronous processing

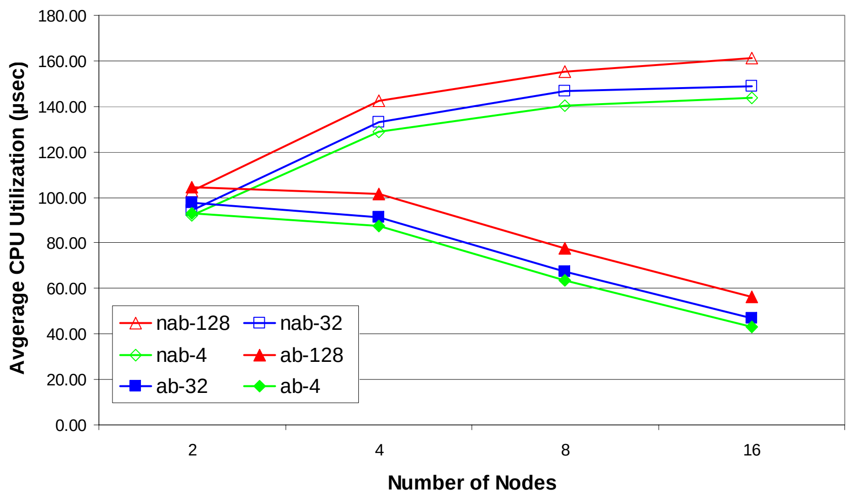
CPU Utilization for Varying Skew (16 nodes)

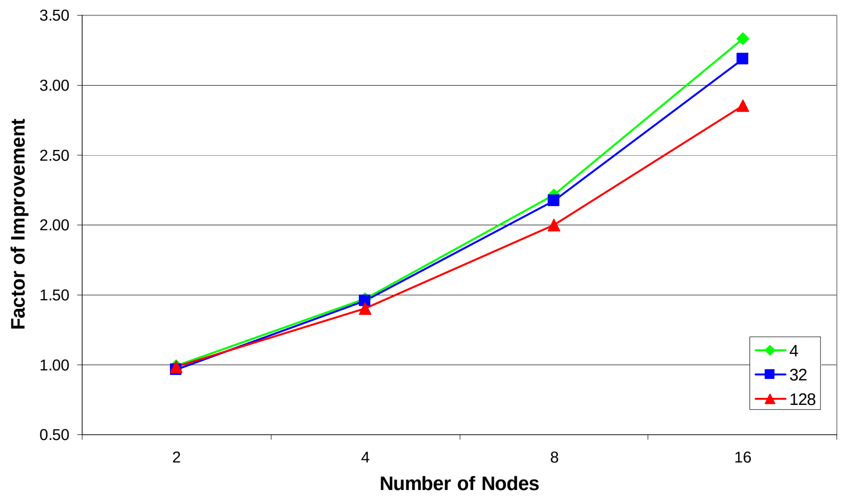




- Application-bypass implementation performs better for all but small-message, small-skew scenario
- As skew increases, small messages see the greatest factor of improvement
- Maximum factor of improvement of 3.3

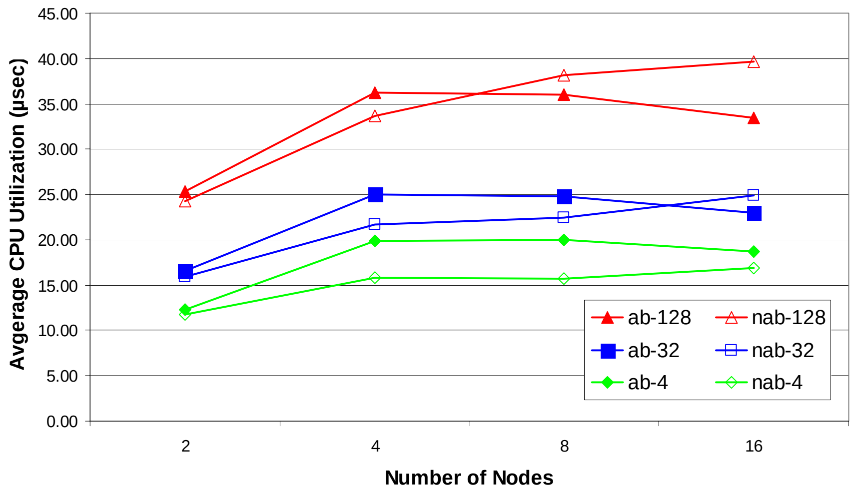
CPU Utilization for Maximal Skew (1000 µs)

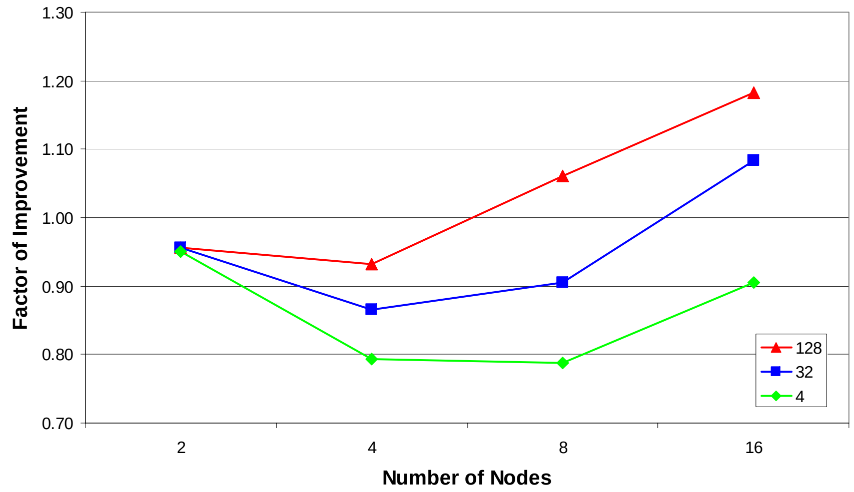




- Application-bypass implementation performs better for all system and message sizes
- Again, greatest factor of improvement observed for small messages
- Maximum factor of improvement of 3.3

CPU Utilization Without Skew



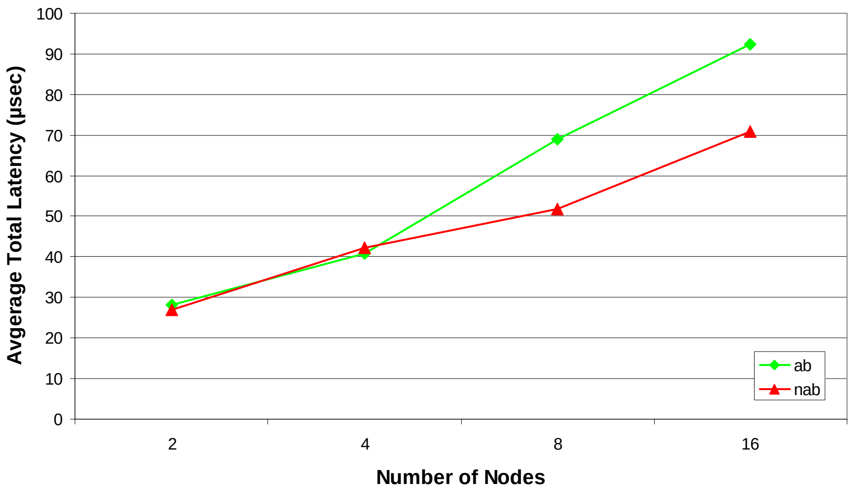


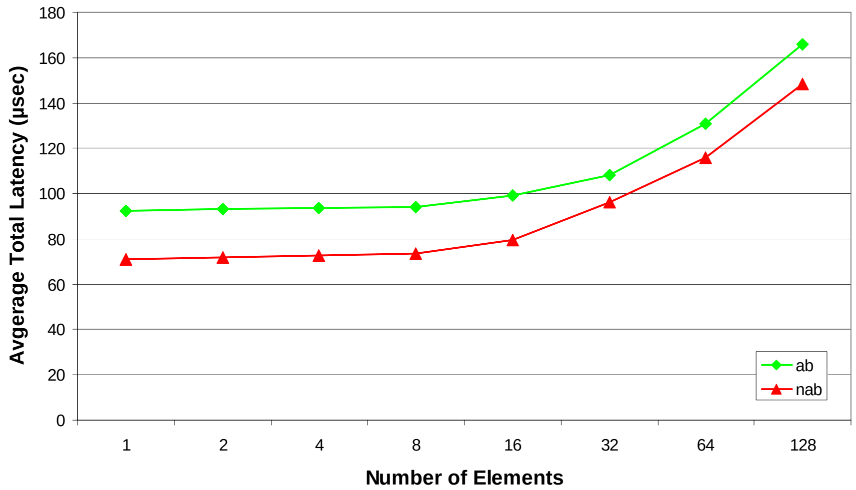
- As system size grows, natural skew is introduced
- Application-bypass scales with system size and message size, default implementation does not
- Maximum factor of improvement of 1.18

Latency Microbenchmark

- Calculate last node (node furthest away from root in logical tree)
- Determine one-way latency from root to last node
- At last node:
  - Start timing
  - Call MPI_Reduce
  - Wait for message that root has completed MPI_Reduce
  - Stop timing
  - Subtract one-way latency from measured time
  - Repeat 10,000 times and take average across all nodes

Total Latency Without Skew





- Interrupt overhead becomes visible at 8 nodes due to natural skew (Single-element messages)
- Overhead remains fairly constant with jump to 16 nodes
- Overhead decreases slightly for larger message sizes (System size of 16 nodes)

Overview

- Background & Motivation
- Design Challenges
- Our Implementation
- Experimental Results
- **Conclusions & Future Work**

Conclusions

- Designed and implemented application-bypass version of reduction in MPICH over GM
- Factor of improvement of up to 3.3 under conditions of process skew
- Improvement highest for small message sizes, which is common case
- Improvement increases with system size, indicating enhanced scalability on large-scale clusters
- Application-bypass is critical for scalability of reduction operations on large-scale clusters!

Future Work

- Evaluation on large-scale clusters
- Application-based evaluation
- SMP-based implementation
- Integration with other application-bypass operations into unified framework
- Integration of NIC-based techniques
- Portability to other communication subsystems

Contact Information

- Network-Based Computing Lab:
  http://nowlab.cis.ohio-state.edu/
- Authors:

  - wagnera@cis.ohio-state.edu
  - buntinas@mcs.anl.gov
  - bright@cs.sandia.gov
  - panda@cis.ohio-state.edu