

Message Passing in Hierarchical and Heterogeneous Environments: MPI-3 and Beyond

Pavan Balaji

Computer Scientist

Group lead, Programming Models and Runtime Systems

Argonne National Laboratory

balaji@mcs.anl.gov

<http://www.mcs.anl.gov/~balaji>

MPI Philosophy

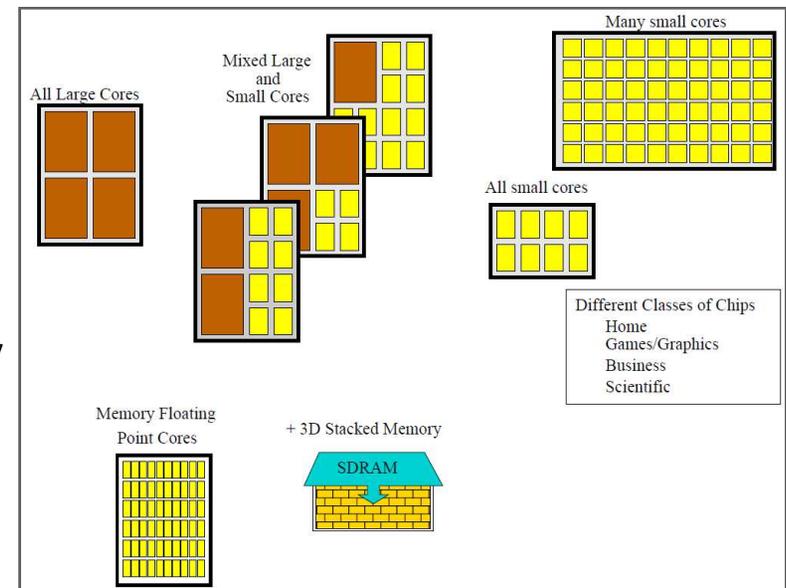
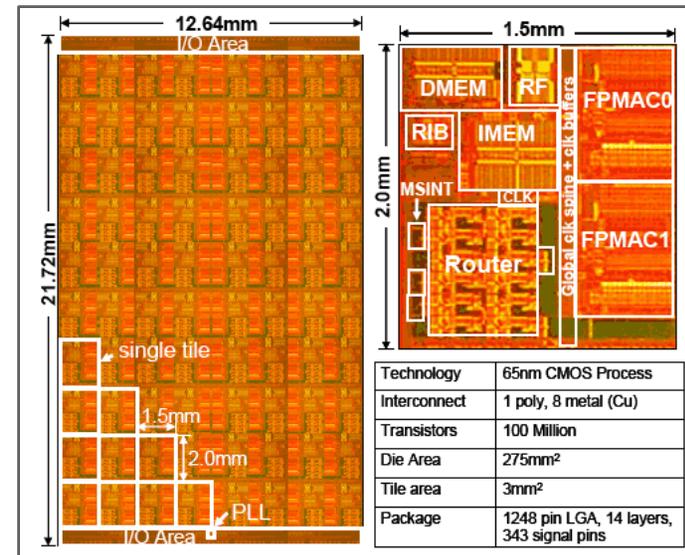
- Different programming models have picked different tradeoffs in the space of portability, performance, expressiveness, and ease of use
- Often contradicting goals
 - Most programming models try to pick a subset of this functionality
 - Trying to do everything is a pleasant but (mostly) a pipe dream
- MPI chose to be highly feature rich and portable, and has enabled an ecosystem to be built around it to provide domain-specific algorithms and simplistic use of a subset of the features (PETSc, Trilinos, FFTW, ADLB, Debuggers)
 - Do what you do well
 - Build a strong base to allow others to build on top of



Architectural Considerations

- Weird processor and memory architectures are coming
 - Hierarchical processors
 - Threads in cores in dies in sockets in NUMA domains in nodes in systems
 - Lots of light-weight cores
 - In-order execution (no speculation)
 - Small caches
 - No hardware cache coherency
 - Heterogeneous cores
 - General purpose processors + GPGPUs
 - Integrated heterogeneous processors
 - Hierarchical and Heterogeneous Memory
 - Accelerator memory
 - NVRAM, Less reliable memory

Courtesy William Gropp (UIUC)



Hardware Evolution Requires Software Evolution

- The current MPI standard (and implementations) lack in a number of features
- Improvements for hierarchical and heterogeneous architectures
 - Topology awareness for applications
 - Locality information
 - Improvements for asynchronous data and computation movement
 - Interoperability with threads and accelerators
 - Compiler support for optimizing MPI applications



Virtual Topologies (well, not entirely MPI-3)

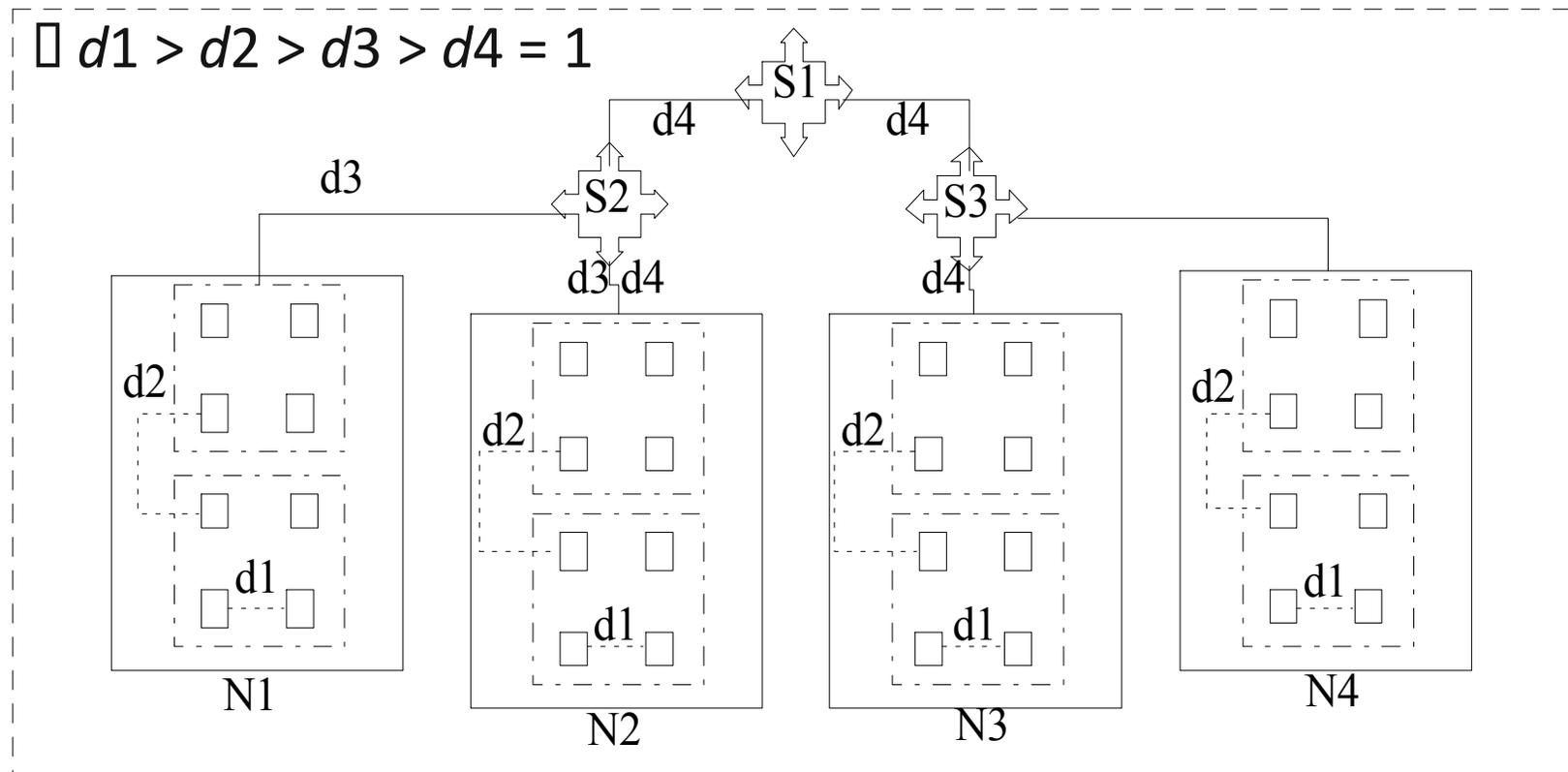
MPI Virtual Topology

- MPI topology functions:
 - Define the communication topology of the application
 - Logical process arrangement or **virtual topology**
 - Possibly reorder the processes to efficiently map over the system architecture (**physical topology**) for more performance
- Virtual topology models:
 - **Cartesian topology**: multi-dimensional Cartesian arrangement
 - **Graph topology**: non-specific graph arrangement
- Graph topology representation
 - Non-distributed: easier to manage, less scalable
 - Distributed: new to the standard, more scalable



Physical Topology Distance Example

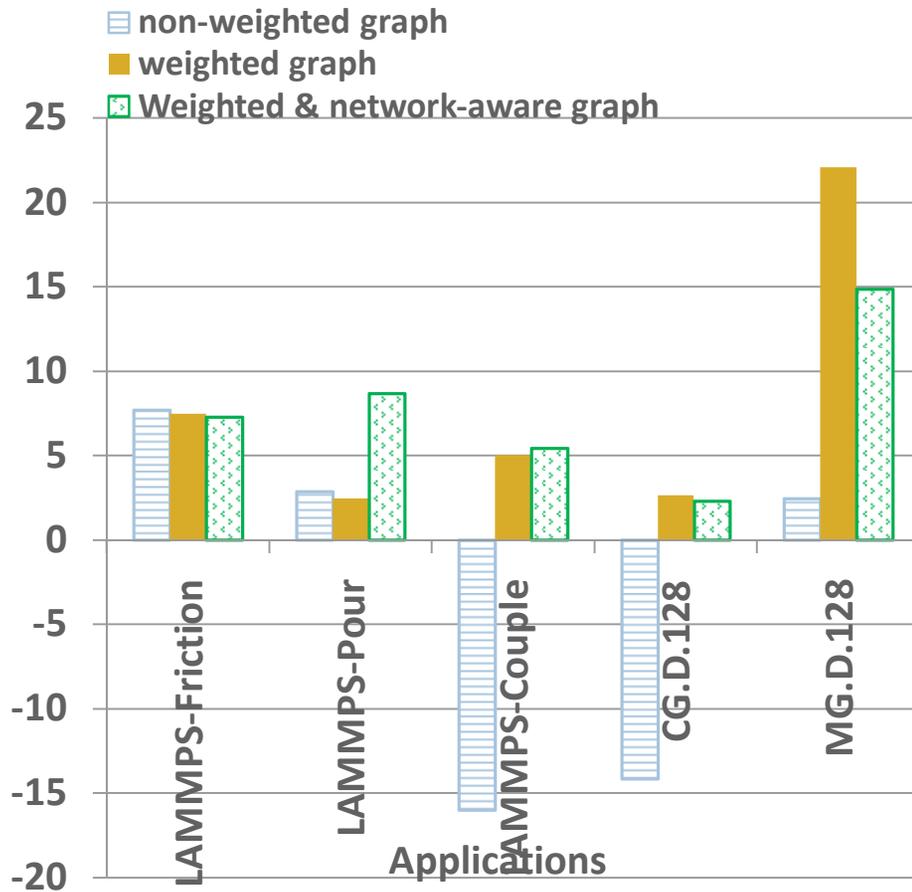
- $d1$ will have the highest load value in the graph.
- The path between $N2$ and $N3$ ($d4$) will have the lowest load value, indicating the lowest performance path.



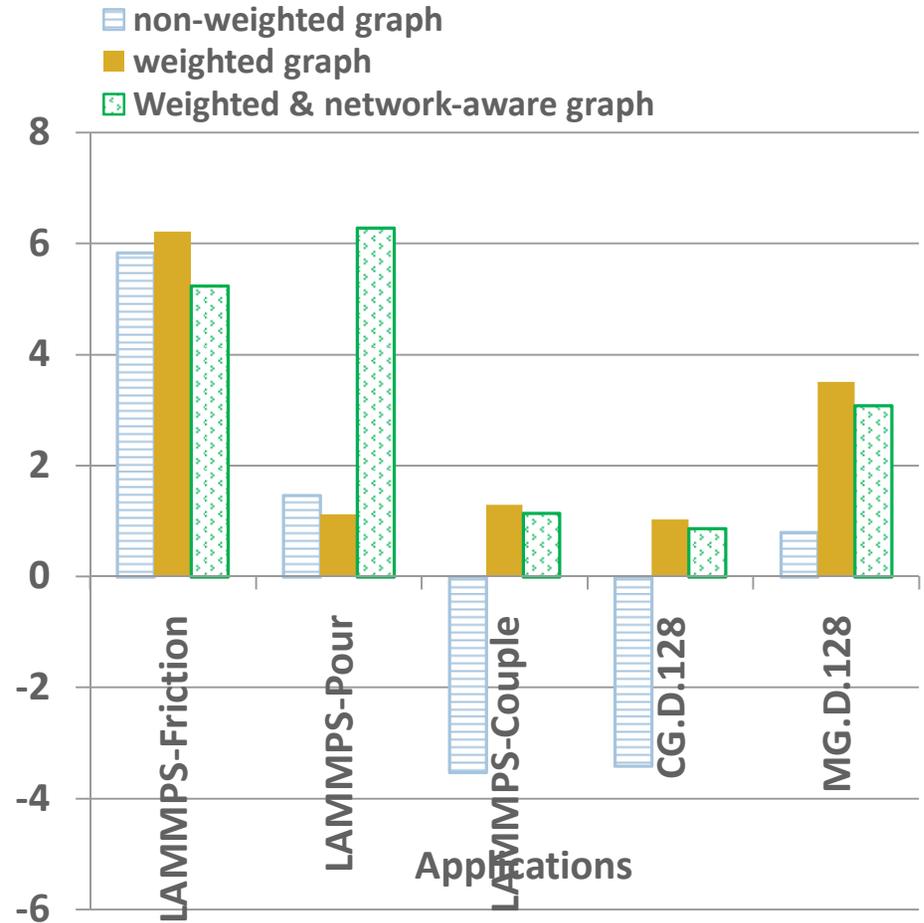
Applications: Topology-aware Mapping Improvement over Block Mapping (%)

128-core cluster B

Communication Time Improvement



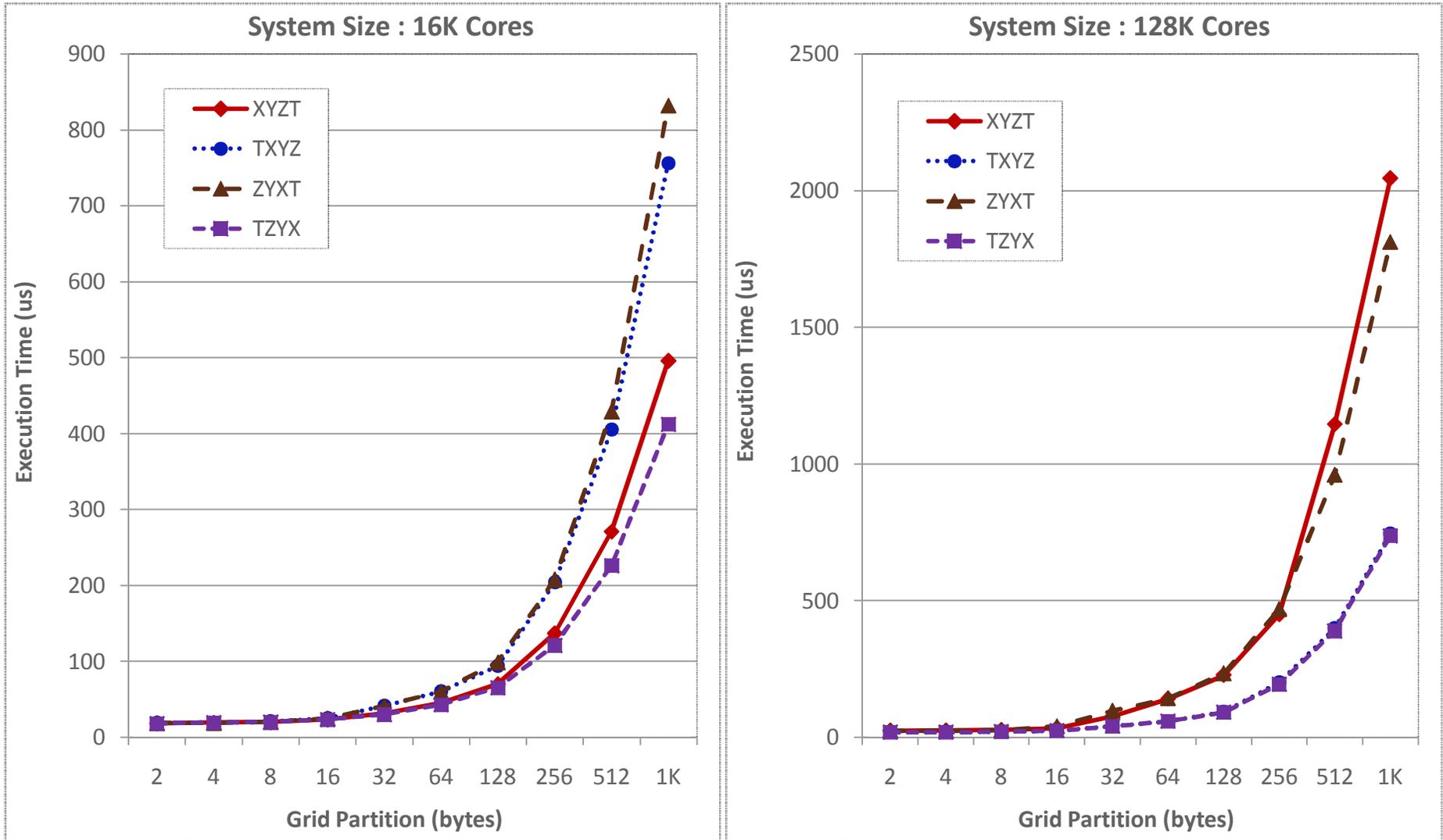
Run-time Improvement



Collaboration with Ahmad Afsahi, Queen's University, Canada



Communication Kernel for Ocean Modeling



Physical Topology Information Retrieval

- Virtual topology functionality relies on the user providing MPI with the application communication pattern
- What about work-stealing applications? Communication is pretty random
- MPI-3 introduced a new function called `MPI_Comm_split_type`
 - Idea is to split a communicator based on some physical hardware information
 - E.g., you can split a communicator to contain processes that can create a shared memory region
 - Implementations can extend it to allow any form of creation – same node, same NUMA socket, same cache domain, same switch, same rack

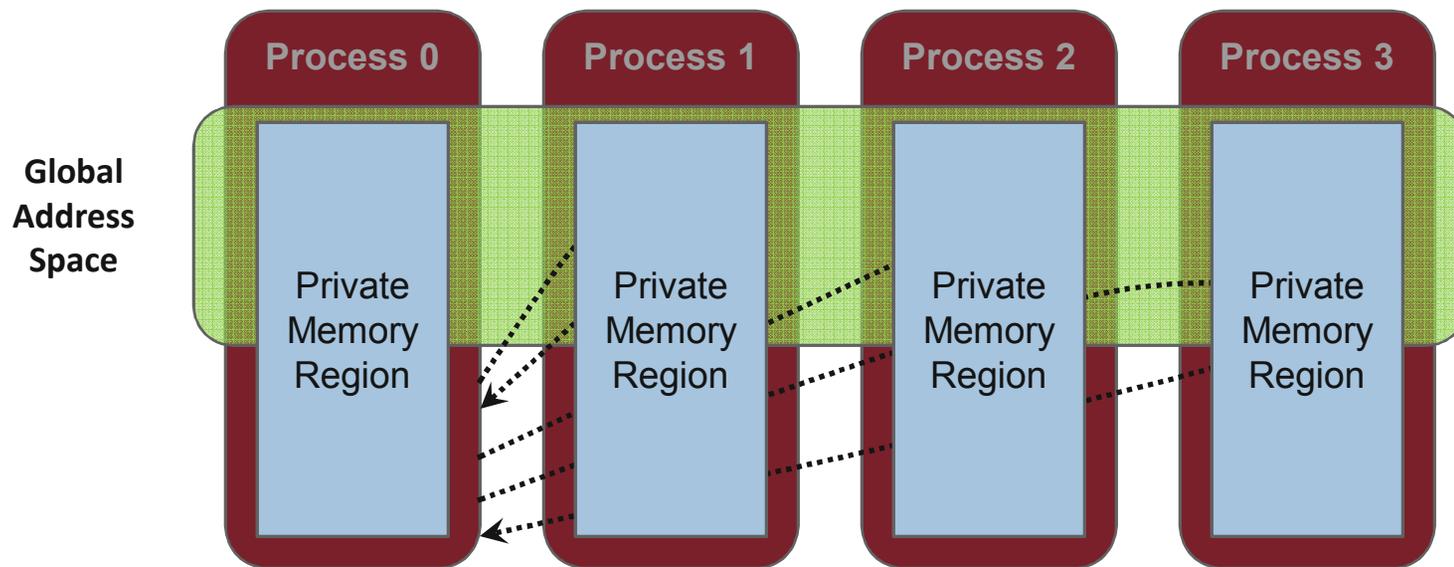
Collaboration with William Gropp, UIUC



MPI-3 RMA and Shared Memory

MPI-3 RMA (low-level one-sided communication)

- Per-process-pair one-sided communication
 - Should be able move data without requiring that the remote process synchronize
 - Each process exposes a part of its memory to other processes
 - Other processes can directly read from or write to this memory



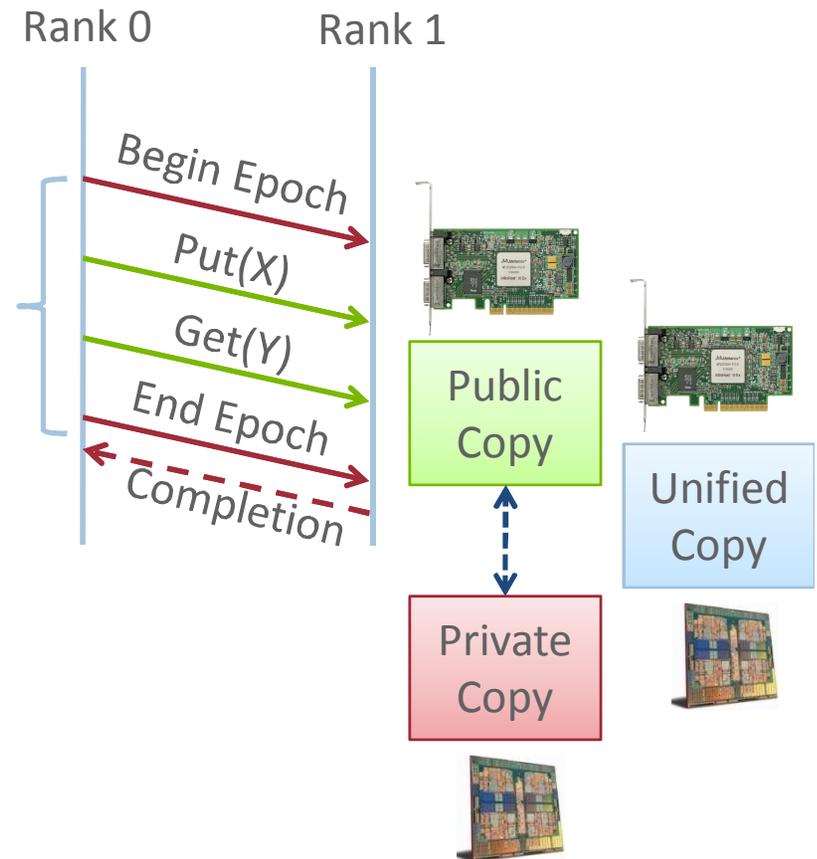
Data movement

- MPI_Get, MPI_Put, MPI_Accumulate, MPI_Get_accumulate, “atomic get”, “atomic put”, compare-and-swap, fetch-and-op
 - Move data between public copy of target window and origin local buffer
- **Nonblocking**, subsequent synchronization may block
- Distinct from load/store from/to private copy
- Specified ordering of operations (RAR, WAW, WAR, RAW ordered by default; can disable ordering as needed)



Generalized Memory Model

- Window: Expose memory for RMA
 - Logical public and private copies
 - Portable data consistency model
- Accesses must occur within an epoch
- Active and Passive synchronization modes
 - Active: target participates
 - Passive: target does not participate



Locality-awareness in MPI RMA

- Ability to allow for “locales” for public memory
- Can split a communicator into locales
 - Locales are implementation defined
 - Shared memory, NUMA, socket, L2 cache, ...
- MPI-3 allows “shared memory capable locales”
 - In MPI-3.1, we are planning to add other locales as well (e.g., same rack, same switch)
- Can create shared memory on these locales
 - Direct load/store access within the locale
 - Appropriate synchronization can be done with traditional MPI primitives



Comparison to other models (thanks Bill!)

- Bonachea's and Duell's criticism
 - Exposing public memory is collective (support for dynamically exposing public memory in MPI-3)
 - Exposed memory has to be allocated through MPI, e.g., expose stack memory (yes, we allow that now; go shoot yourself on the foot!)
 - Overlapping GET/PUT operations was erroneous (it is valid now, but data can be garbage; atomic GET/PUT provided for per-basic-datatype atomicity)
 - Too generic; loses functionality on architectures that support stronger memory models (two different memory models provided)



Comparison to other models (contd.)

- ARMCI/UPC/CAF
 - Semantics similar to MPI_RMA_UNIFIED (if available)
 - Local completion
 - Either blocking, implicit handles or Test/Wait(all)
 - Similar to “implicit handles”
 - Remote completion
 - (all)Fence == Flush(_all)
 - Ordering with collectives
 - ARMCI_Barrier combines barrier + allfence
 - Memory allocation
 - Remote memory allocation not provided
 - Can be done if remote progress is not required (e.g., GASNet)



MPI + Threads (OpenMP, etc.)

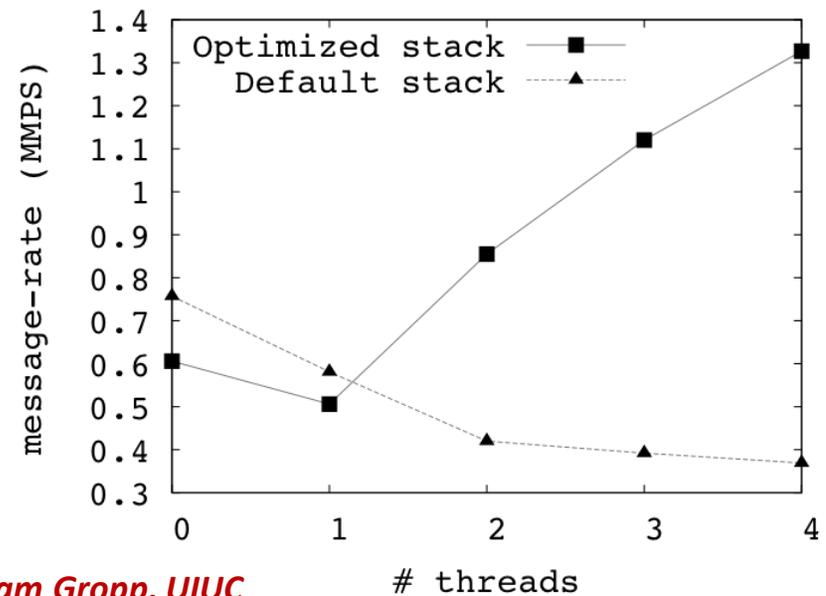
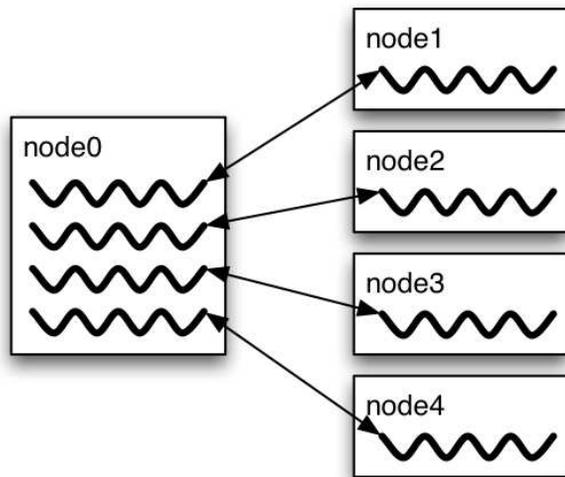
The Current Situation

- All MPI implementations support `MPI_THREAD_SINGLE` (duh).
- They probably support `MPI_THREAD_FUNNELED` even if they don't admit it.
 - Does require thread-safe malloc
 - Probably OK in OpenMP programs
- Many (but not all) implementations support `THREAD_MULTIPLE`
 - Hard to implement efficiently though (lock granularity issue)
- “Easy” OpenMP programs (loops parallelized with OpenMP, communication in between loops) only need `FUNNELED`
 - So don't need “thread-safe” MPI for many hybrid programs
 - But watch out for Amdahl's Law!



Performance with MPI_THREAD_MULTIPLE

- Thread safety does not come for free
- The implementation must protect certain data structures or parts of code with mutexes or critical sections
- We ran tests to measure communication performance when using multiple threads versus multiple processes
 - Details in our *Parallel Computing* (journal) paper (2009)



Collaboration with William Gropp, UIUC



Why is it hard to optimize `MPI_THREAD_MULTIPLE`

- MPI internally maintains several resources
- Because of MPI semantics, it is required that all threads have access to some of the data structures
 - E.g., thread 1 can post an `Irecv`, and thread 2 can wait for its completion
 - thus the request queue has to be shared between both threads
 - Since multiple threads are accessing this shared queue, it needs to be locked – adds a lot of overhead

Thread 0:

```
MPI_Irecv(..., &req);  
pthread_barrier(...);
```

Thread 1:

```
pthread_barrier(...);  
MPI_Wait(..., &req, ...);
```

- MPI semantics issue, not (only) an implementation issue
 - Mostly unused semantic, but the implementation has to support it, just in case



MPI-3.1 proposal: Threads as first-class citizens

- Background:
 - Matching semantics in MPI are based on source, tag and communicator
 - You have MPI_ANY_SOURCE and MPI_ANY_TAG, but not an “MPI_ANY_COMM”
 - Communicators are independent, but MPI semantics make it hard to decouple communicator specific data structures
- Idea is to add an attribute (or info argument) to a communicator to specify independence
 - Application tells the MPI implementation – “when I’m waiting on this communicator, you don’t need to check for progress on other communicators”
 - MPI implementation can create independent queues for each communicator



Active Messages in MPI

Active Messages/RMI in MPI

- Implementations of Active Messages over MPI exist
 - E.g., AM++ by Jeremiah, Torsten, Andrew
- We are looking at a slightly different model (AM support included inside the MPI implementation)
 - Datatype marshalling/demarshalling
 - Asynchronous progress without requiring a polling thread and additional locks
 - Integrated with the MPI-RMA infrastructure (better memory protection with remote locks)
 - Multiple internal implementation models (e.g., sender computes)

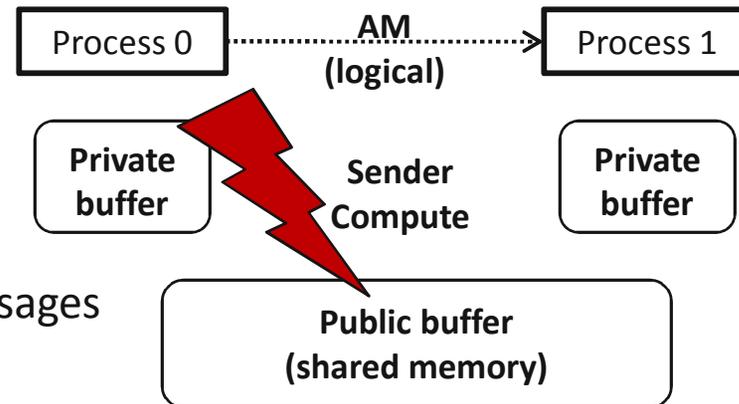
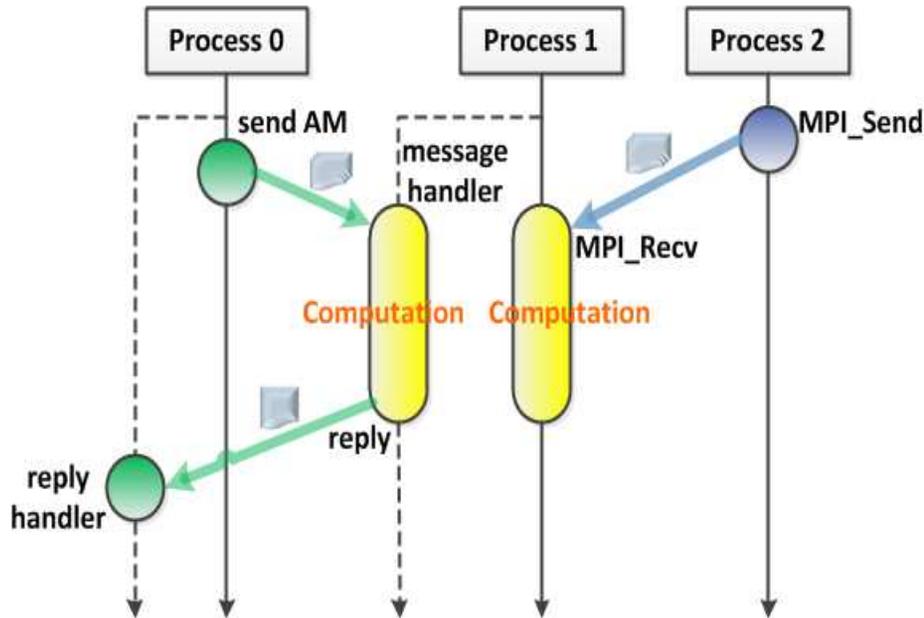


MPI-AM functionality

- Collective registration of AM functions
 - Each process specifies a local function (useful when different processes are different executables)
 - Remote operation (logically) triggers execution of the function local to the target process
 - Functions are equivalent (need not be identical); allows for sender-compute models
- Two models
 - Sender specifies which data will be touched
 - Overlapping active messages targeted to the same window
 - Can get data and compute locally if the memory touched is small
 - Sender does not specify which data will be touched
 - Can still do sender compute in shared memory domains



MPI-AM functionality (contd.)

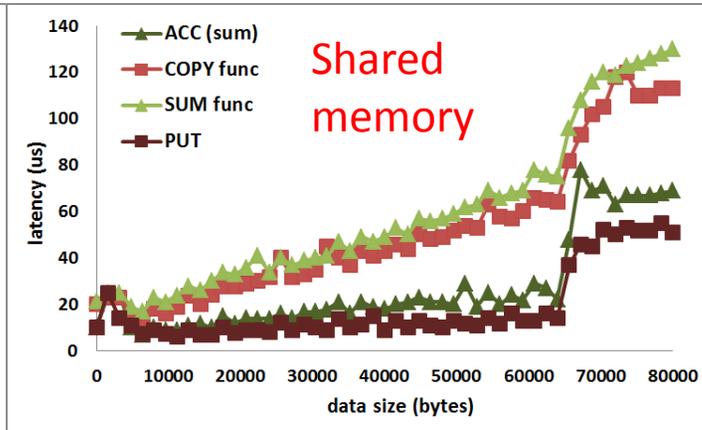
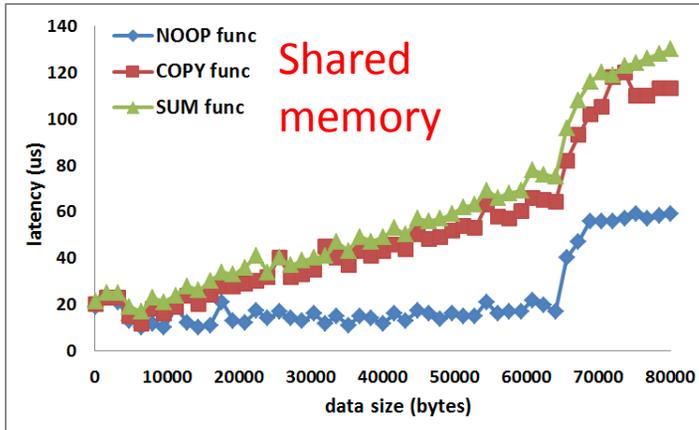


1. Issue: Asynchronous handling of active messages over the network and shared memory
2. We use a sleeping thread for the network to support RMA and AM messages (no shared data structures with send/recv)
3. No overhead on non-AM communication

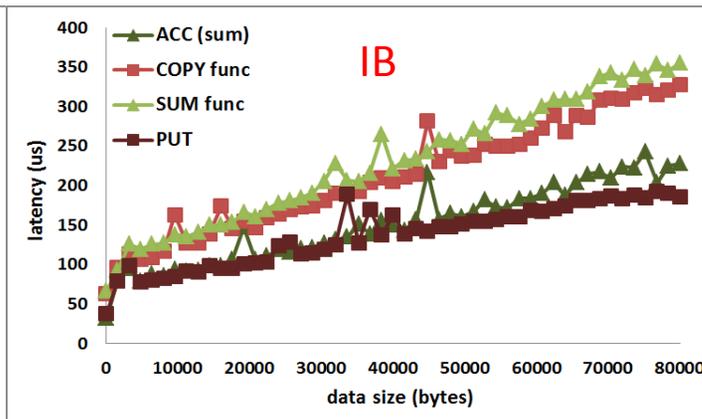
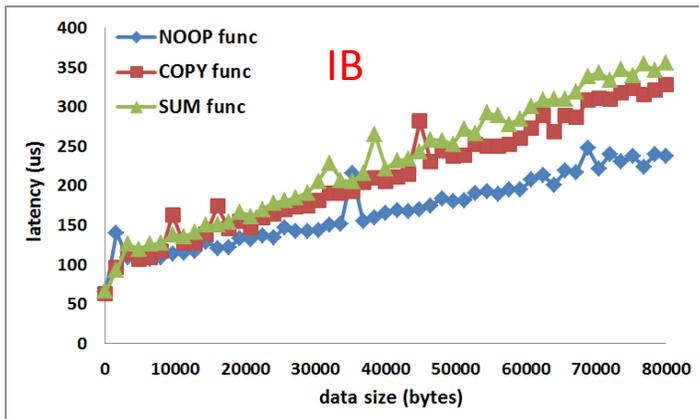


AM Performance

- User-defined operation



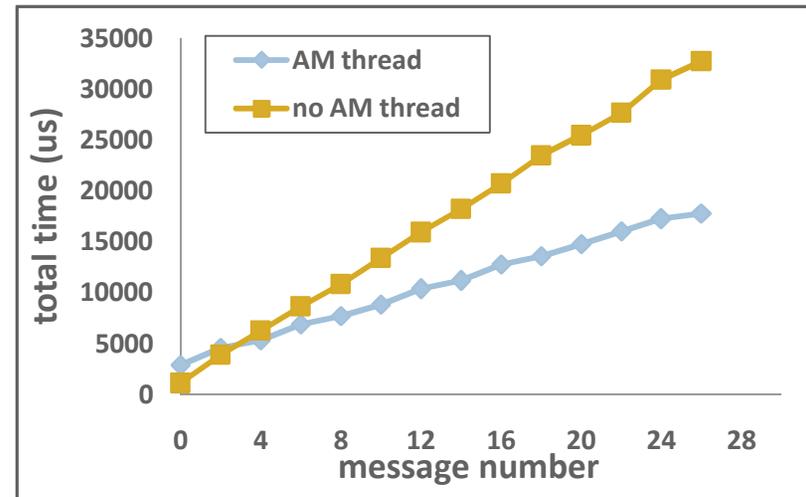
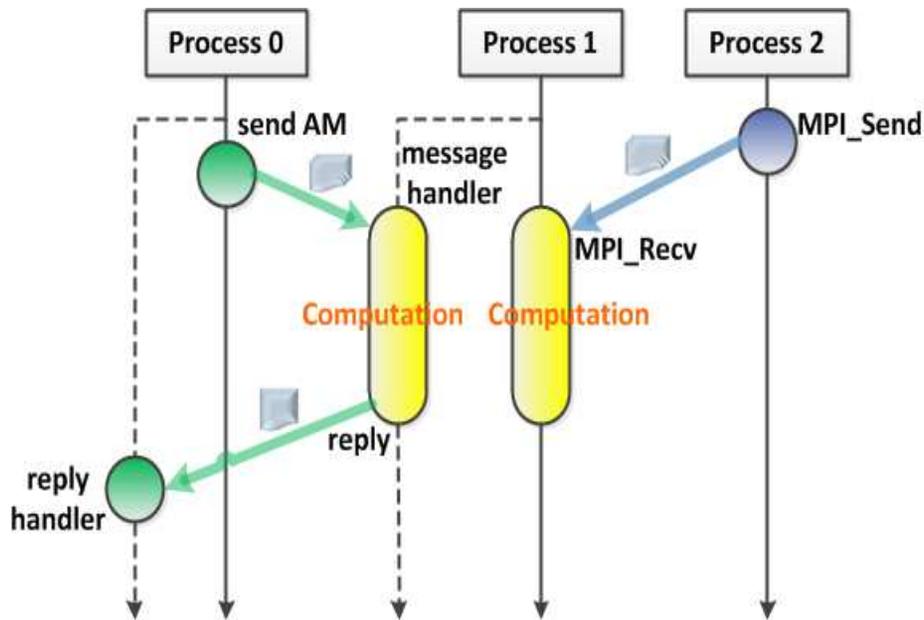
- Compare to PUT and predefined operation



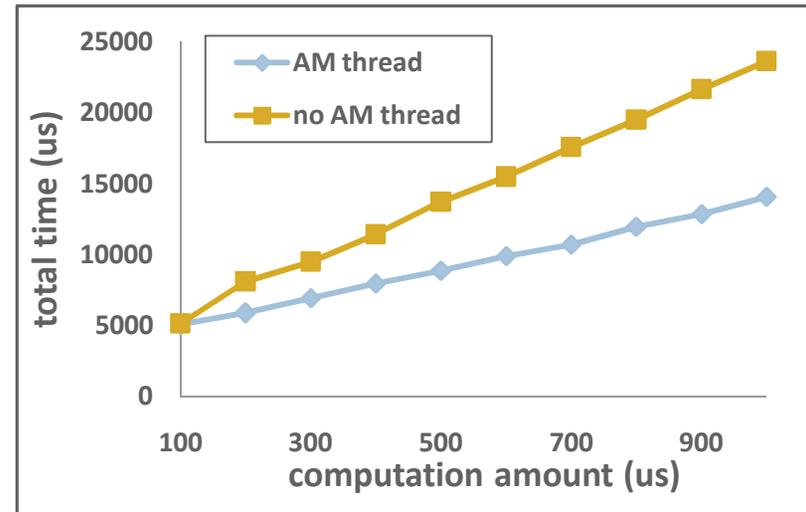
Collaboration with William Gropp, UIUC



Asynchronous Progress Performance (preliminary numbers)



Fixed computation amount (=500 us)



Fixed message number (= 10)



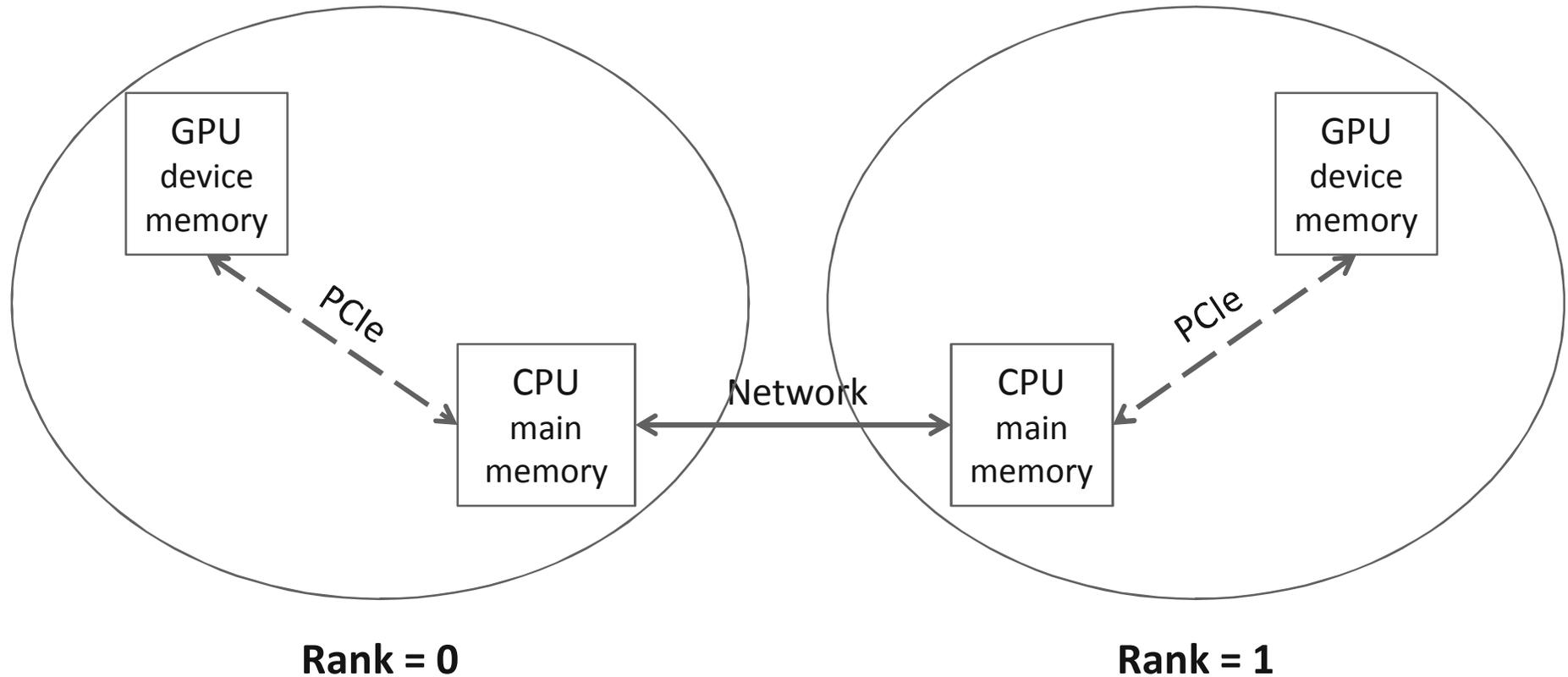
Integrated Data Movement in Heterogeneous Environments

Hybrid Programming with Accelerator models

- Simple GPU interoperability works out of the box
- Many MPI processes
- Each MPI process can launch CUDA/OpenCL/... kernels to compute on data
- Move data back to the process memory
- Use MPI to move data between processes



Interoperability with GPUs: Current Data Model



```
if (rank == 0)
{
    cudaMemcpy(s_buf, s_dev_buf, D2H);
    MPI_Send(s_buf, ...);
}
```

```
if (rank == 1)
{
    MPI_Recv(r_buf, ...);
    cudaMemcpy(r_dev_buf, r_buf, H2D);
}
```

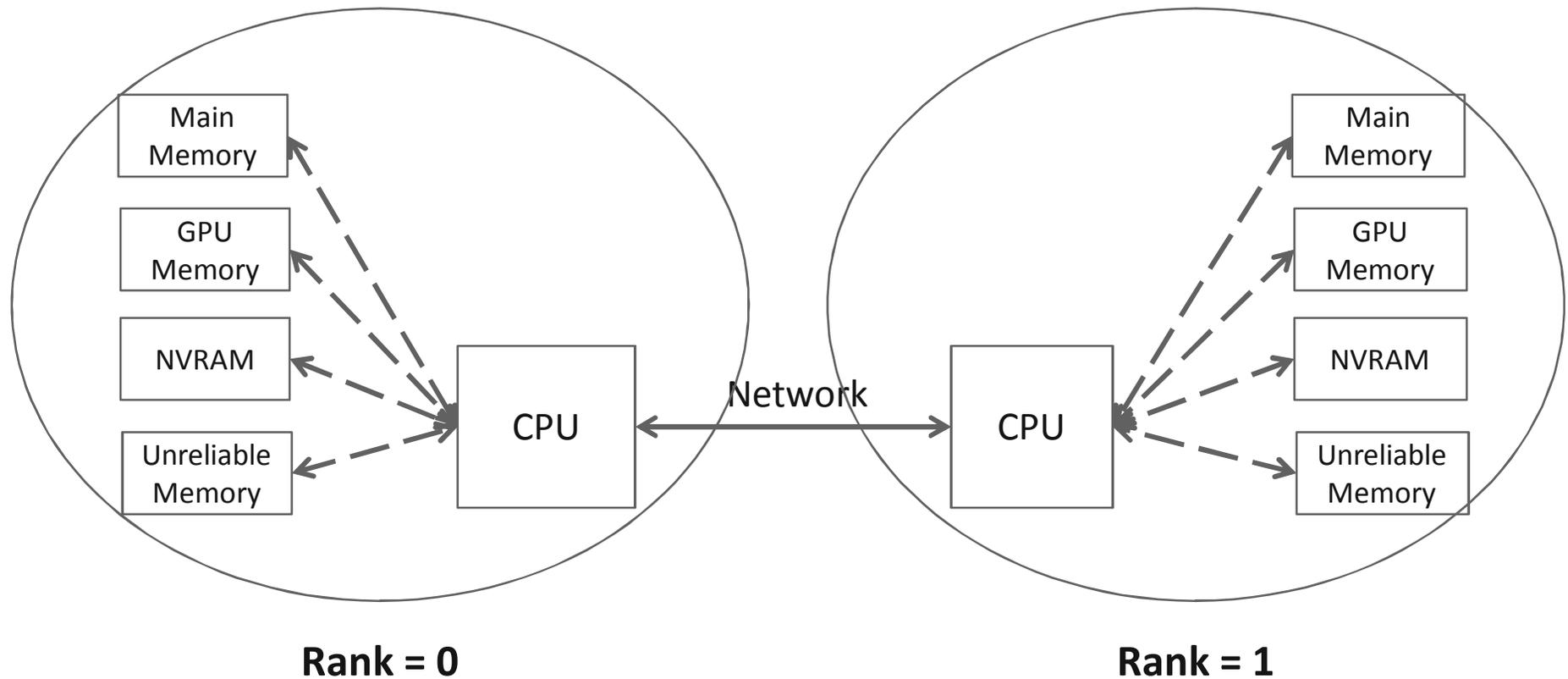


Tighter Interoperability: MPI-ACC (research project)

- Productivity Goal (API)
 - Implement the rich data transfer interface of MPI for CUDA/OpenCL/..
- Performance Goal
 - Pipeline the data movement between GPU memory, host memory and remote node using architecture specific enhancements
 - NVIDIA: GPU Direct
 - Multi-stream copies between GPU and memory (multiple command queues can benefit from parallelism in the DMA engine)
 - Future architectures:
 - Zero-copy data movement if accelerators have direct network access
 - Eliminate “GPU-to-host” data transfers if the heterogeneous processors share memory spaces
- All of the above should happen *automatically* within the MPI implementation, i.e. applications should not redo their data movement for each architecture



MPI-ACC: Generalized Runtime for Accelerator Systems

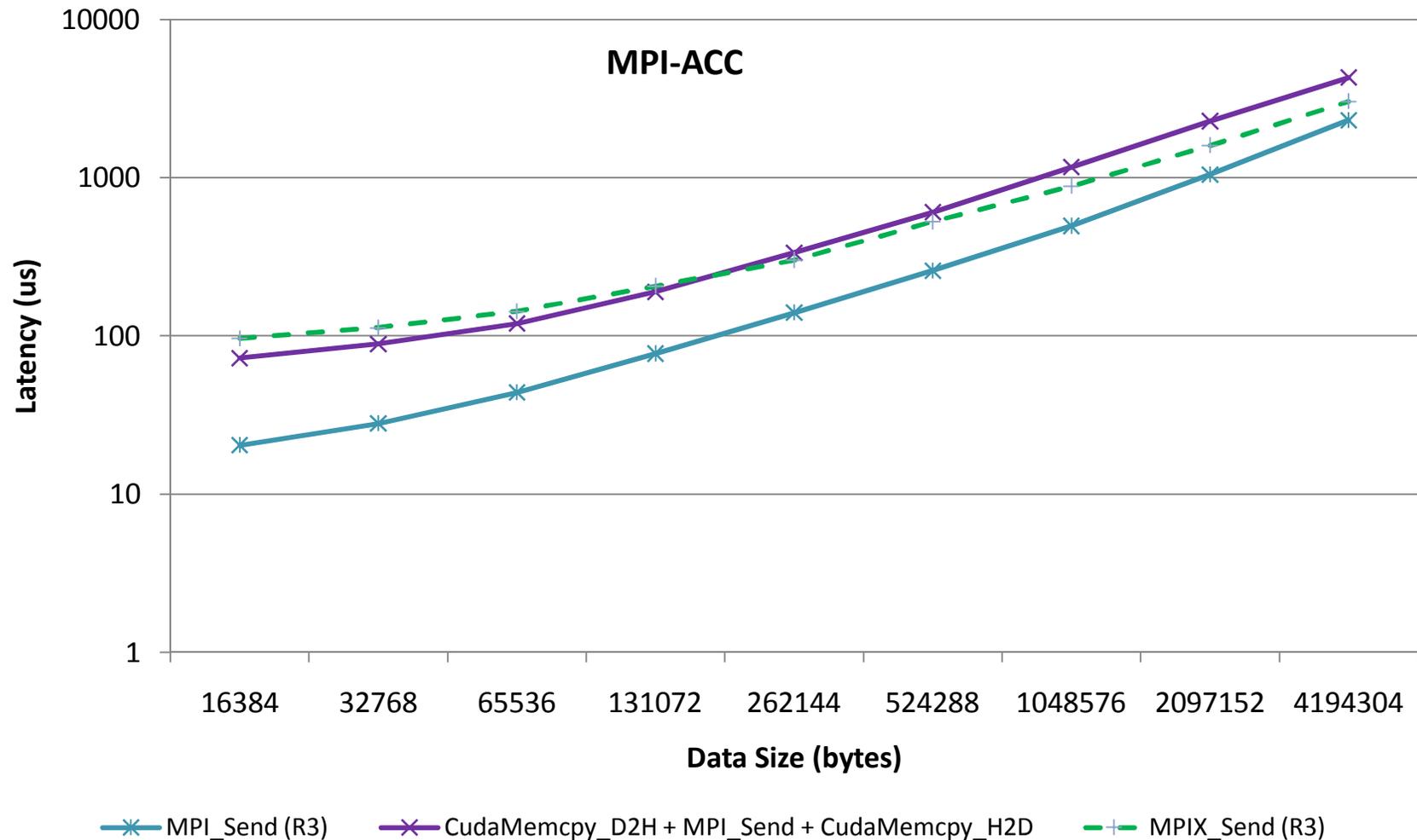


```
if (rank == 0)
{
  MPI_Send(s_buf, ...);
}
```

```
if (rank == 1)
{
  MPI_Recv(r_buf, ...);
}
```



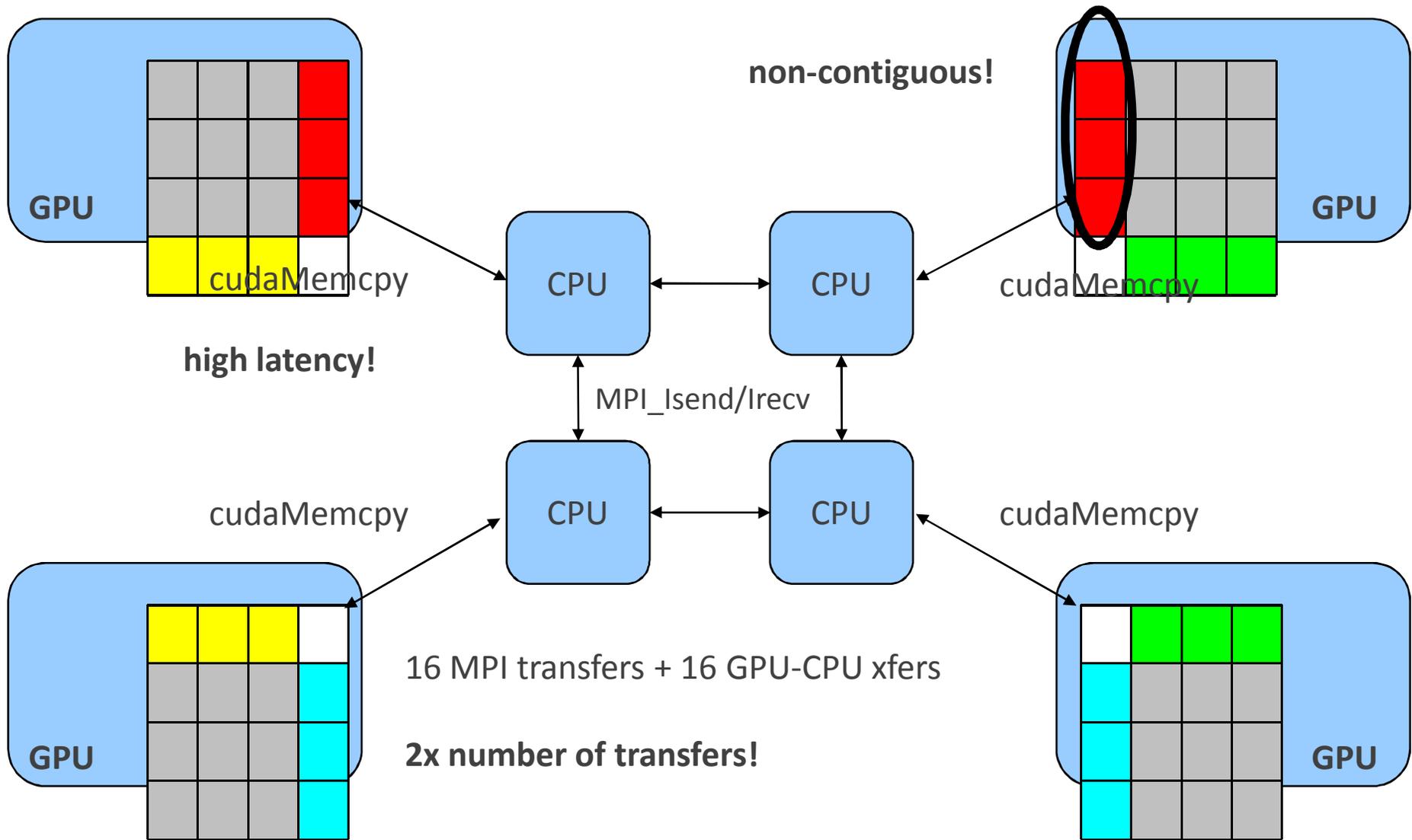
Experimental Results (CUDA - RNDV mode)



Collaboration with Wu-chun Feng (Virginia Tech), Xiaosong Ma (NCSU), Laxmikant Kale (UIUC), and recently Cray



MPI + GPU Example - Stencil Computation

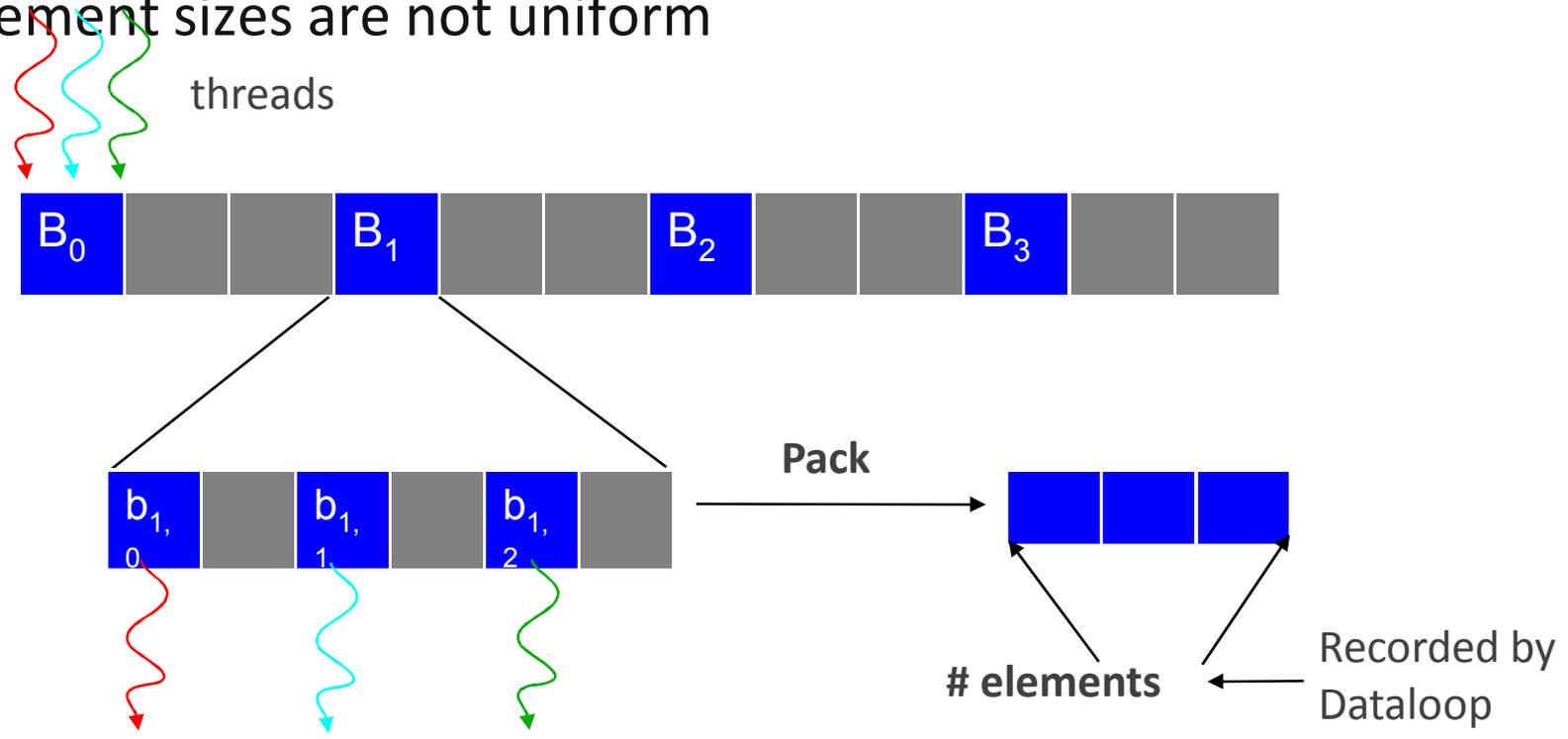


Collaboration with Nagiza Somatova, NCSU



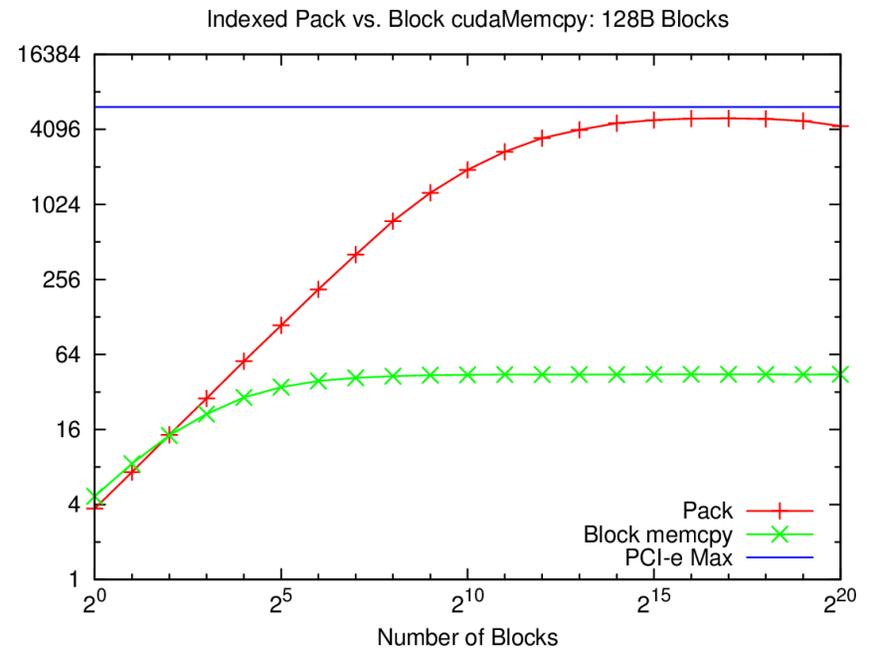
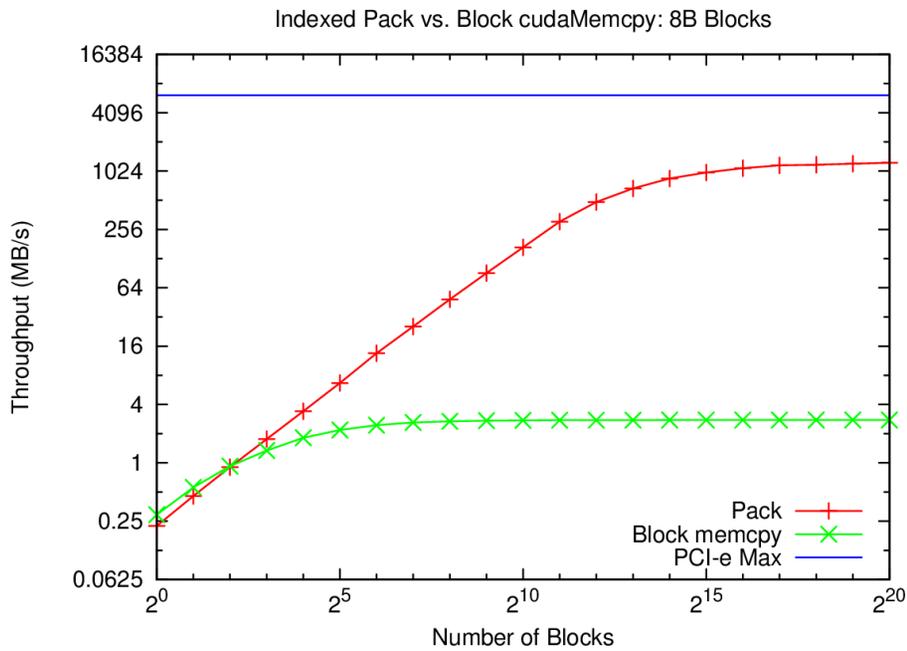
GPU optimizations for Data Packing

- Element-wise traversal by different threads
- Embarrassingly parallel problem, except for structs, where element sizes are not uniform



traverse by **element #**, read/write using **extent/size**

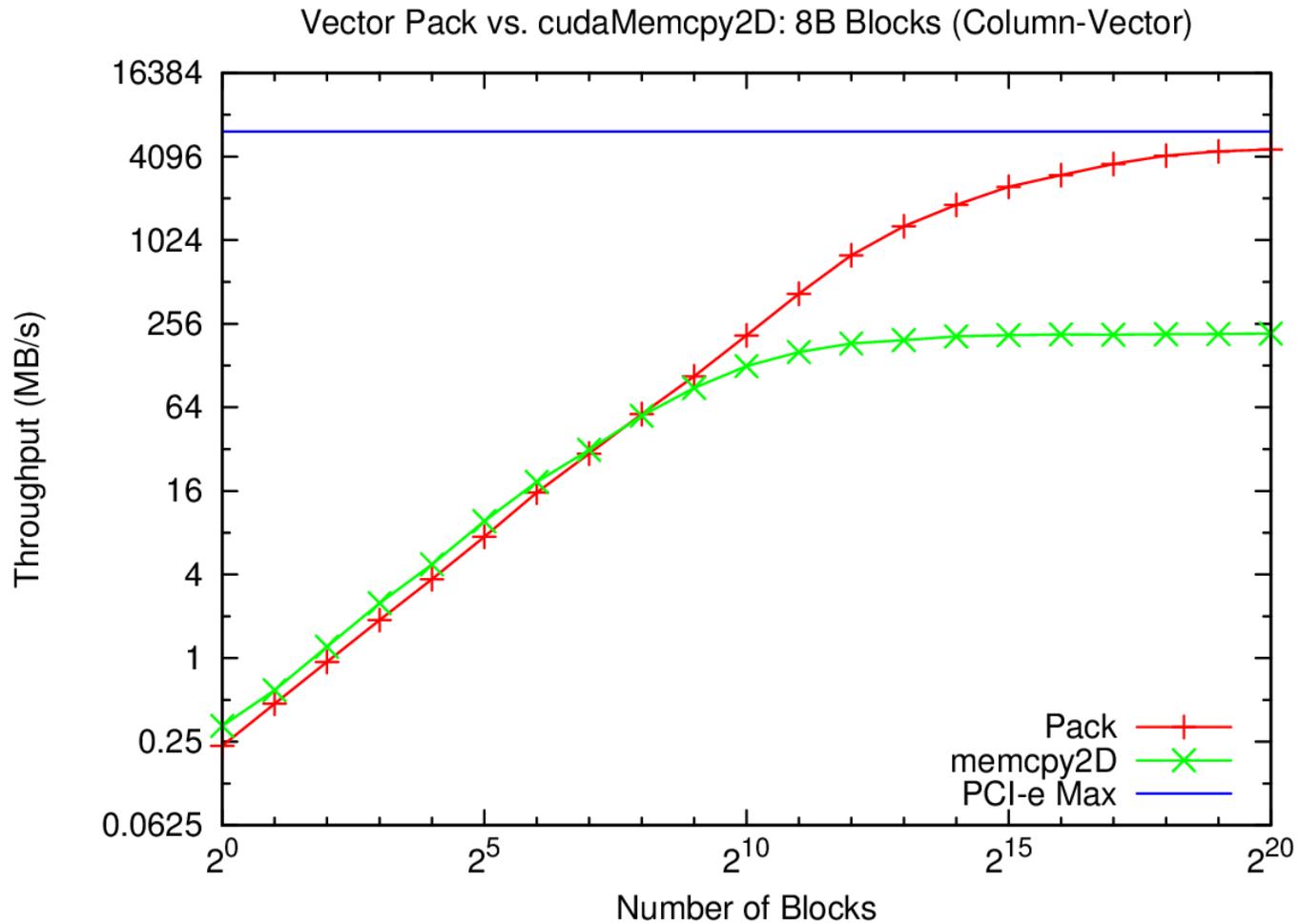
Packing Throughput (Indexed)



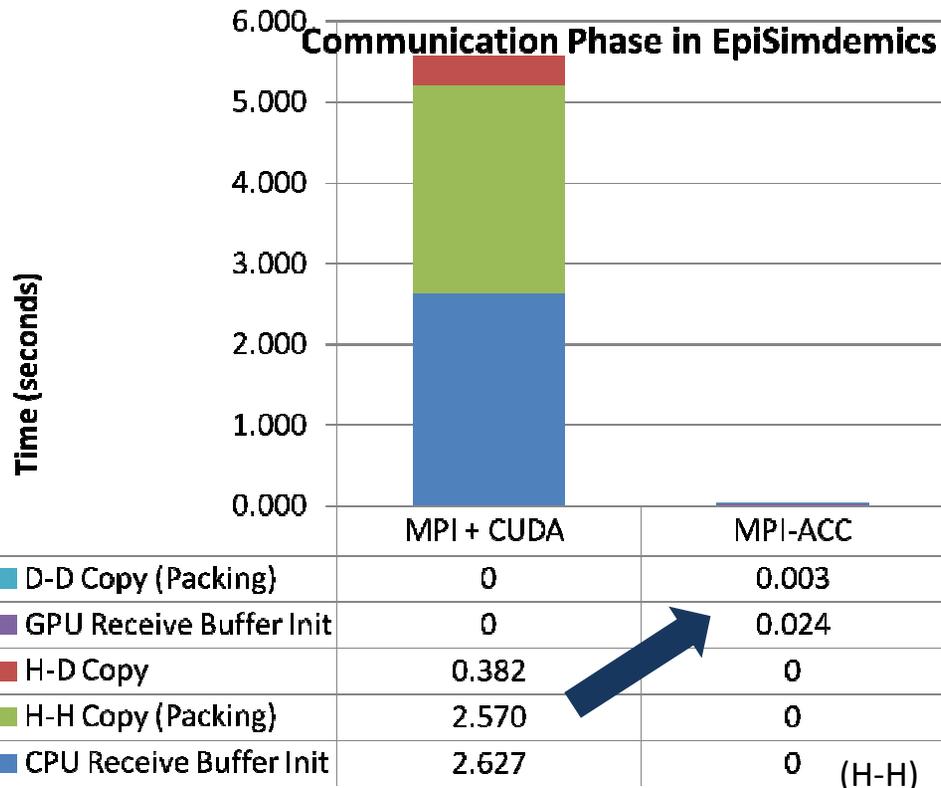
	displacement	0	512	1024	1536	2048
(bytes)	blocklength	8	8	8	8	8
		128	128	128	128	128



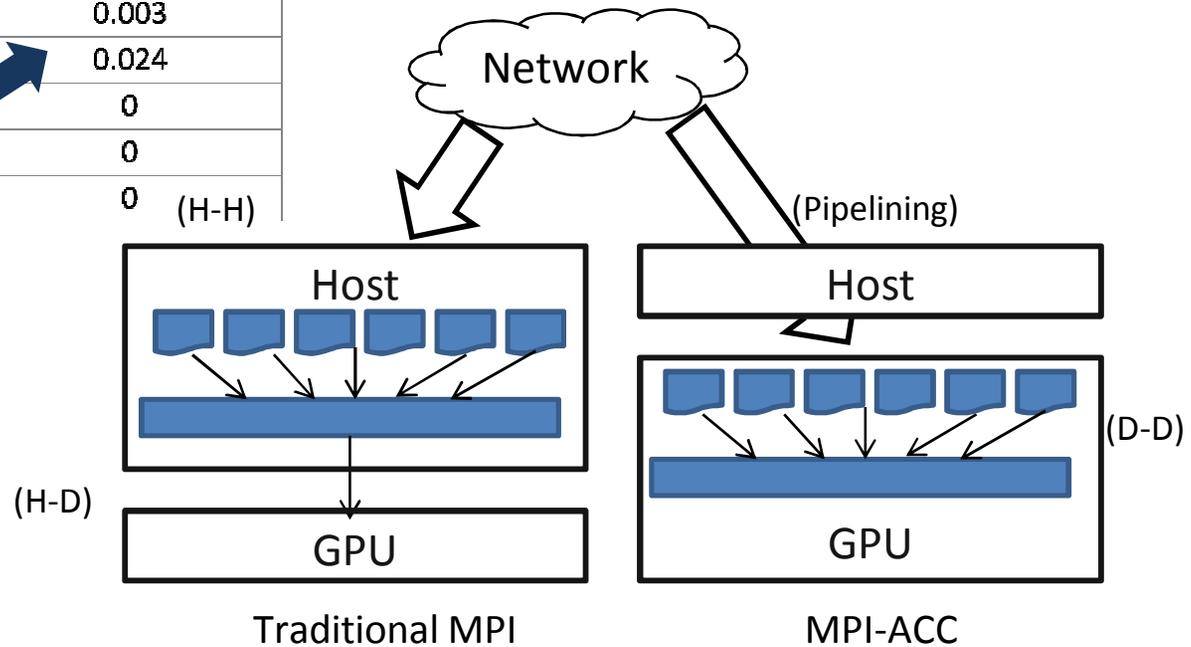
Packing Throughput (Column-Vector)



MPI-ACC Performance Comparison



- MPI-ACC accelerators data movement operations by two orders of magnitude
- *Enables* new application-level optimizations



Compiler Infrastructure for MPI

Beyond MPI-3: Compiler Support for MPI

- Compiler support for MPI
 - Idea is to provide performance portability and simplicity of use for MPI applications
- Pragmas allow applications to achieve performance portability across different platforms
- Collaboration with Qing Yi @ UT San Antonio

```
#pragma mpi coalesce safe_fn(foo)
for (i = 0; i < 100; i++) {
    MPI_Put();
    MPI_Put();

    /* some code */
    foo();

    MPI_Accumulate();
    MPI_Get();

    /* some other code */
    bar();

    MPI_Put();
}
```

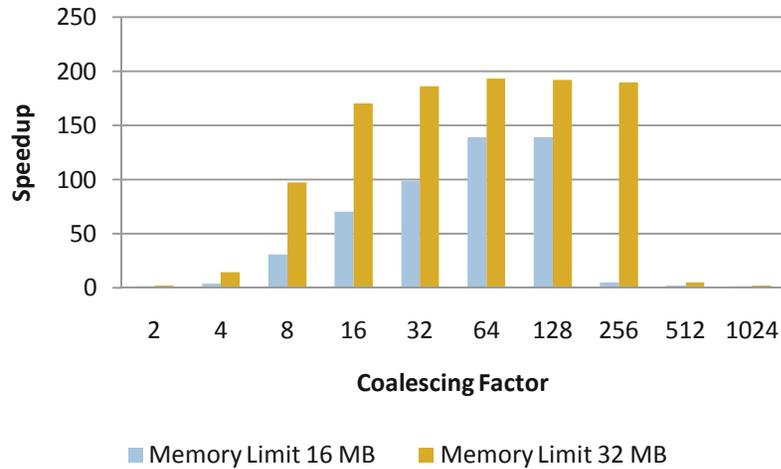


Issues and Considerations

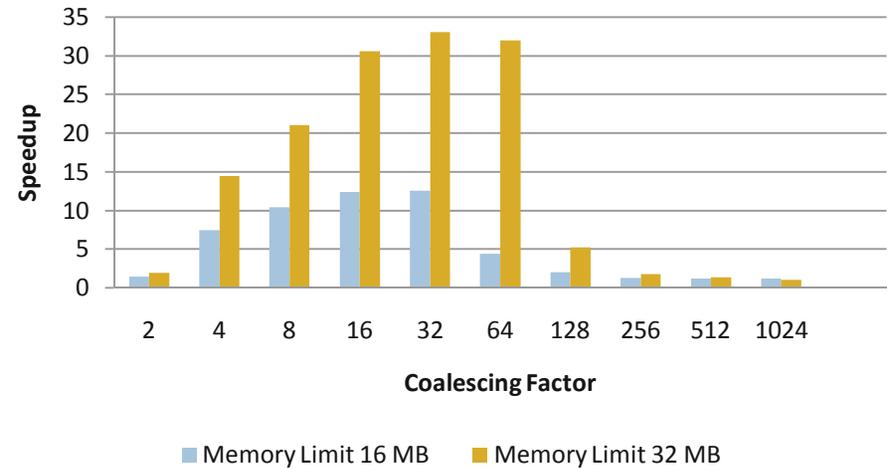
- Data coalescing
 - Why? Avoid function call overheads for each operation
 - Amount of coalescing required is architecture specific
 - Overlapping Operations:
 - Multiple accumulate operations to the same location is valid
 - Single accumulate operation writing multiple data elements to the same location is not
 - Handling unrecognized functions
- Transformation of bulk-synchronous to asynchronous GET/PUT models
- Load/store and PUT/GET interchangeability



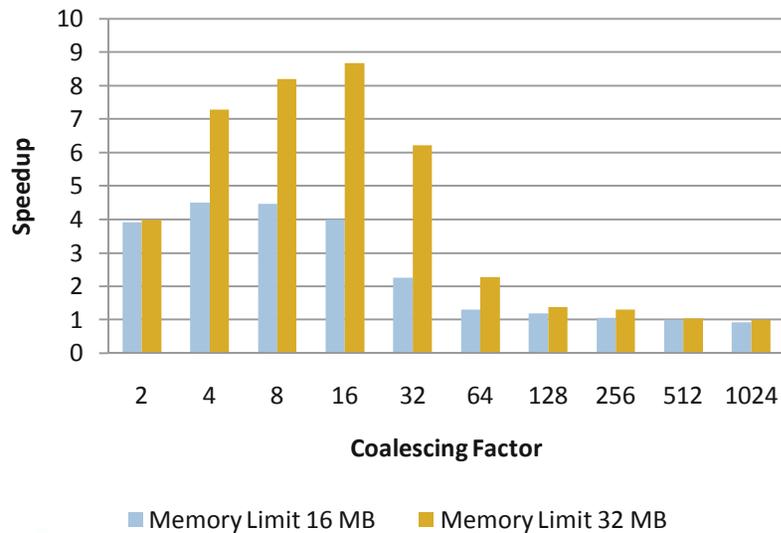
Machine: Fusion, #Proc: 64



Machine: Fusion, #Proc: 128

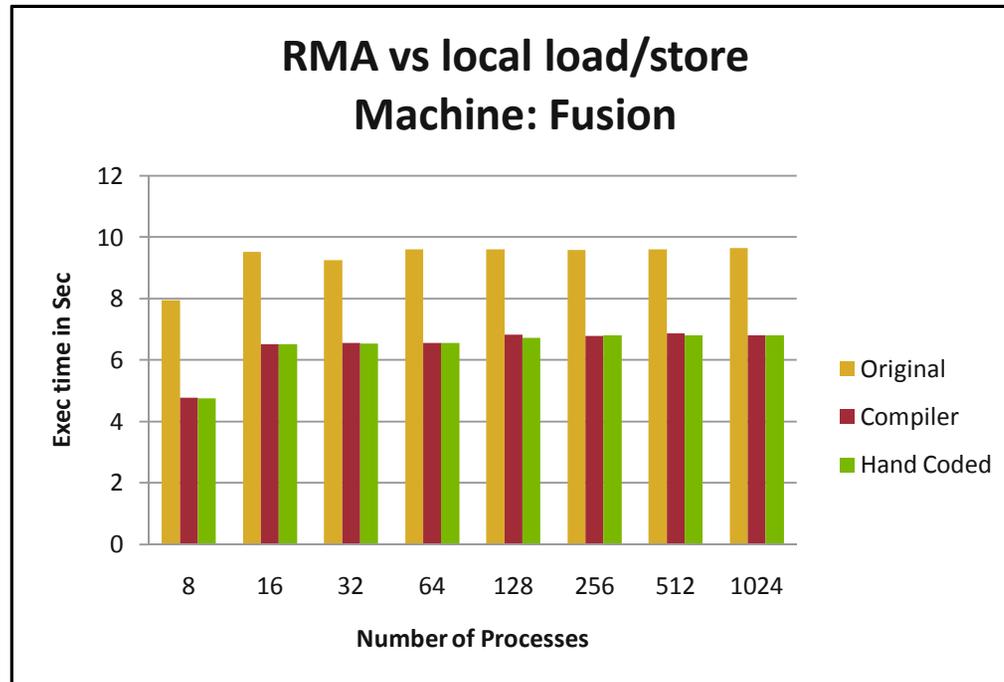


Machine: Fusion, #Proc: 256



- Benchmark: Graph500, BFS
- Graph size: 2^{16} vertices and $2^{16} * 12$ edges.
- Speed-up relative to original implementation.
- Network: InfiniBand.





- Network: InfiniBand
- 99% local access



Summary

- I don't know what the Exascale Programming Model will look like, but I know it will be called MPI (or "MPI+X")
 - Caveat: I don't really think "the Exascale programming model" makes much sense – it has and will always be a suite of (possibly vertically stacked) programming models
 - Whether it is visible to the "end-user" or not is a separate discussion
 - MPI supports and encourages high-level models to be built on top of it
- Different programming models have picked different tradeoffs in the space of portability, performance, expressiveness, and ease of use
 - MPI as a runtime system has chosen to be highly feature rich and portable, and has enabled high-level libraries to be built on top of it to provide domain-specific algorithms and simplistic use of a subset of the features (e.g., PETSc, Trilinos, FFTW, ADLB, ...)
 - This model has been extremely successful and has resulted in a wide and rich ecosystem built around MPI that includes high-level domain-specific libraries, performance and debugging tools, and applications in almost every domain of science
- Current MPI does not meet the requirements for some applications, but both the MPI standard and its implementations are evolving



Thank You!

- MPICH: shameless plug
- Leads
 - Argonne National Laboratory
 - University of Illinois, Urbana-Champaign
- Core MPICH developers
 - IBM
 - INRIA
 - Microsoft
 - Intel
 - University of British Columbia
 - Queen's University
- Derivative implementations
 - Cray
 - Myricom
 - Ohio State University
- Other Collaborators
 - Absoft, Pacific Northwest National Laboratory, Qlogic, Totalview Technologies, University of Utah

