Advanced MPI Capabilities

VSCSE Webinar (May 6-8, 2014)

Day 2

by

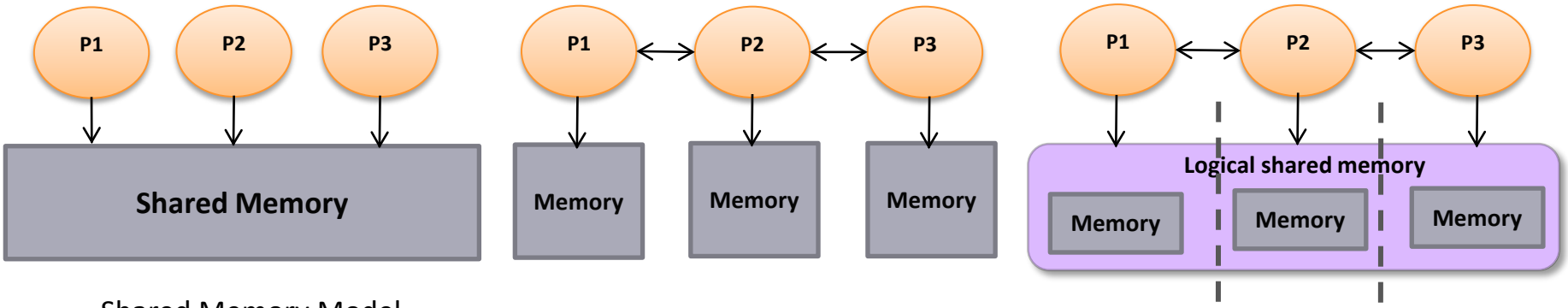
Dhabaleswar K. (DK) Panda
The Ohio State University
E-mail: panda@cse.ohio-state.edu
http://www.cse.ohio-state.edu/~panda

Karen Tomko
Ohio Supercomputer Center
E-mail: ktomko@osc.edu
http://www.osc.edu/~ktomko

Plans for Wednesday and Thursday

- Tuesday, May 6 – MPI-3 Additions to the MPI Spec
  - Updates to the MPI One-Sided Communication Model (RMA)
  - Non-Blocking Collectives
  - MPI Tools Interface
- Wednesday, May 7 – MPI/PGAS Hybrid Programming
  - MVAPICH2-X: Unified runtime for MPI+PGAS
  - MPI+OpenSHMEM
  - MPI+UPC
- Thursday, May 8 – MPI for many-core processor
  - MVAPICH2-GPU: CUDA-aware MPI for NVidia GPU
  - MVAPICH2-MIC Design for Clusters with InfiniBand and Intel Xeon Phi

Parallel Programming Models Overview



Shared Memory Model

SHMEM, DSM

Distributed Memory Model

MPI (Message Passing Interface)

Partitioned Global Address Space (PGAS)

Global Arrays, UPC, Chapel, X10, CAF, …

- Programming models provide abstract machine models
- Models can be mapped on different types of systems
  - e.g. Distributed Shared Memory (DSM), MPI within a node, etc.
- We concentrated on MPI yesterday
- Today's Focus: PGAS and Hybrid MPI+PGAS

Partitioned Global Address Space (PGAS) Models

- Key features
  - Simple shared memory abstractions
  - Light weight one-sided communication
  - Easier to express irregular communication
- Different approaches to PGAS
  - Languages
    - Unified Parallel C (UPC)
    - Co-Array Fortran (CAF)
    - X10
    - Chapel
  - Libraries
    - OpenSHMEM
    - Global Arrays

MPI+PGAS for Exascale Architectures and Applications

- Hierarchical architectures with multiple address spaces
- (MPI + PGAS) Model
  - MPI across address spaces
  - PGAS within an address space
- MPI is good at moving data between address spaces
- Within an address space, MPI can interoperate with other shared memory programming models

- Applications can have kernels with different communication patterns
- Can benefit from different models

- Re-writing complete applications can be a huge effort
- Port critical kernels to the desired model instead

Can High-Performance Interconnects, Protocols and Accelerators Benefit from PGAS and Hybrid MPI+PGAS Models?

- MPI designs have been able to take advantage of high-performance interconnects, protocols and accelerators
- Can PGAS and Hybrid MPI+PGAS models take advantage of these technologies?
- What are the challenges?
- Where do the bottlenecks lie?
- Can these bottlenecks be alleviated with new designs (similar to the designs adopted for MPI)?

Presentation Overview

- **PGAS Programming Models and Runtimes**
  - PGAS Languages: Unified Parallel C (UPC)
  - PGAS Libraries: OpenSHMEM
- Hybrid MPI+PGAS Programming Models and Benefits
- High-Performance Runtime for Hybrid MPI+PGAS Models
- Application-level Case Studies and Evaluation

Compiler-based: Unified Parallel C

- UPC: a parallel extension to the C standard
- UPC Specifications and Standards:
  - Introduction to UPC and Language Specification, 1999
  - UPC Language Specifications, v1.0, Feb 2001
  - UPC Language Specifications, v1.1.1, Sep 2004
  - UPC Language Specifications, v1.2, June 2005
  - UPC Language Specifications, v1.3, Nov 2013
- UPC Consortium
  - Academic Institutions: GWU, MTU, UCB, U. Florida, U. Houston, U. Maryland…
  - Government Institutions: ARSC, IDA, LBNL, SNL, US DOE…
  - Commercial Institutions: HP, Cray, Intrepid Technology, IBM, …
- Supported by several UPC compilers
  - Vendor-based commercial UPC compilers: HP UPC, Cray UPC, SGI UPC
  - Open-source UPC compilers: Berkeley UPC, GCC UPC, Michigan Tech MuPC
- Aims for: high performance, coding efficiency, irregular applications, …

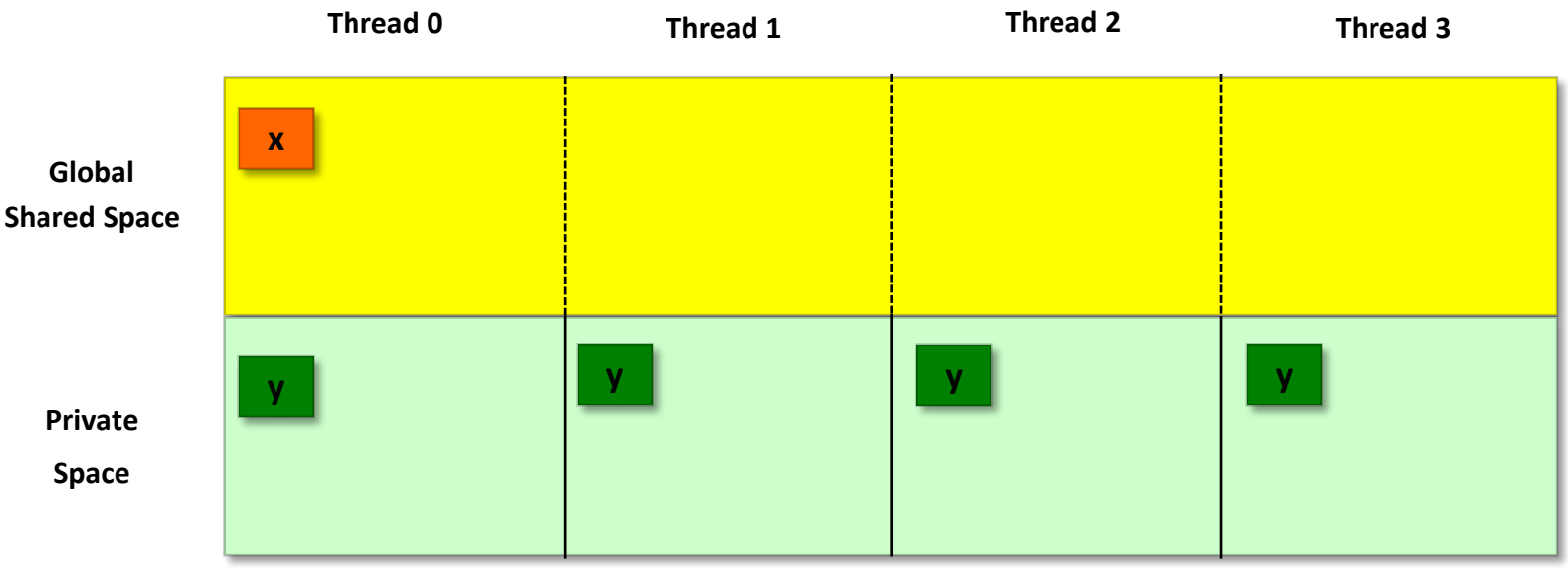
UPC: Execution Model

- A UPC program is translated under a *static* or *dynamic* THREADS environment:
  - THREADS: number of threads working independently SPMD mode
  - MYTHREAD: a unique thread index, ranges from 0 to THREADS-1
  - In static THREADS mode: THREADS is specified at compile time
  - In dynamic THREADS mode: THREADS can be specified at run time

- **Hello World:**

```
#include <upc.h>
#include <stdio.h>
int main() {
  printf(" - Hello from thread %d of %d\n", MYTHREAD, THREADS);
  return 0;
}
```

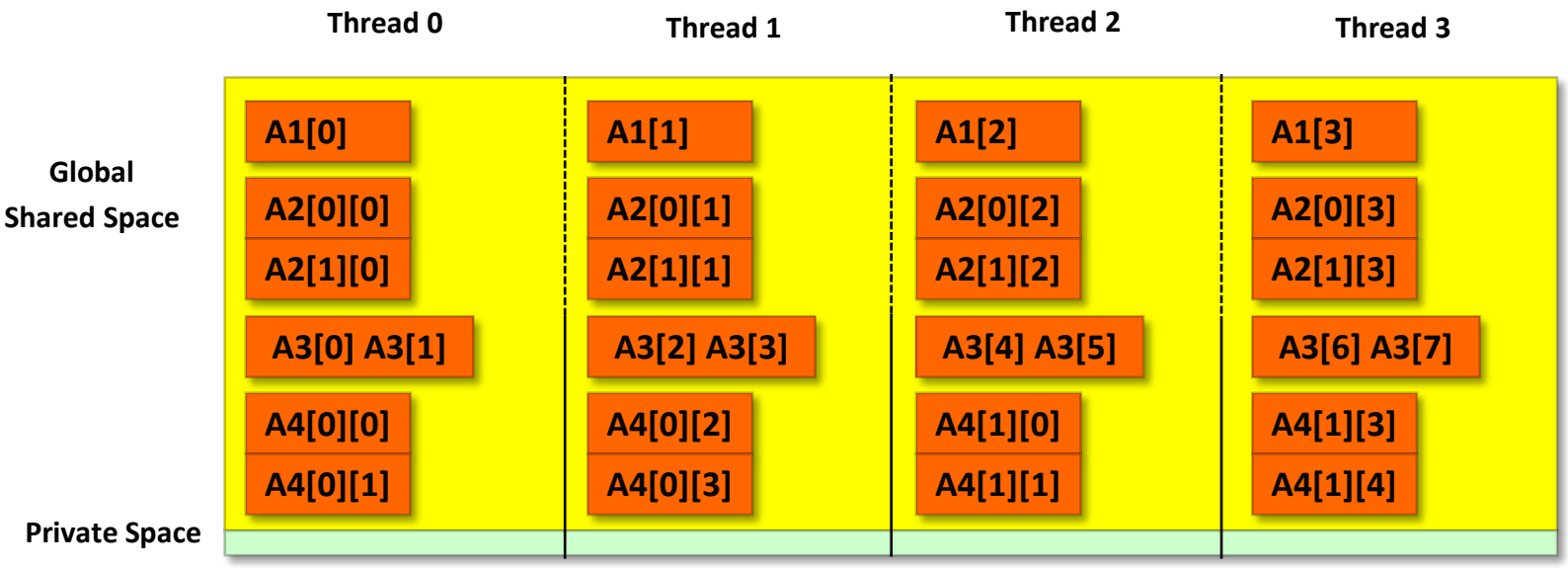
UPC: Memory Model



- Global Shared Space: can be accessed by all the threads
- Private Space: hold all the normal variables; can only be accessed by the local thread
- Examples:

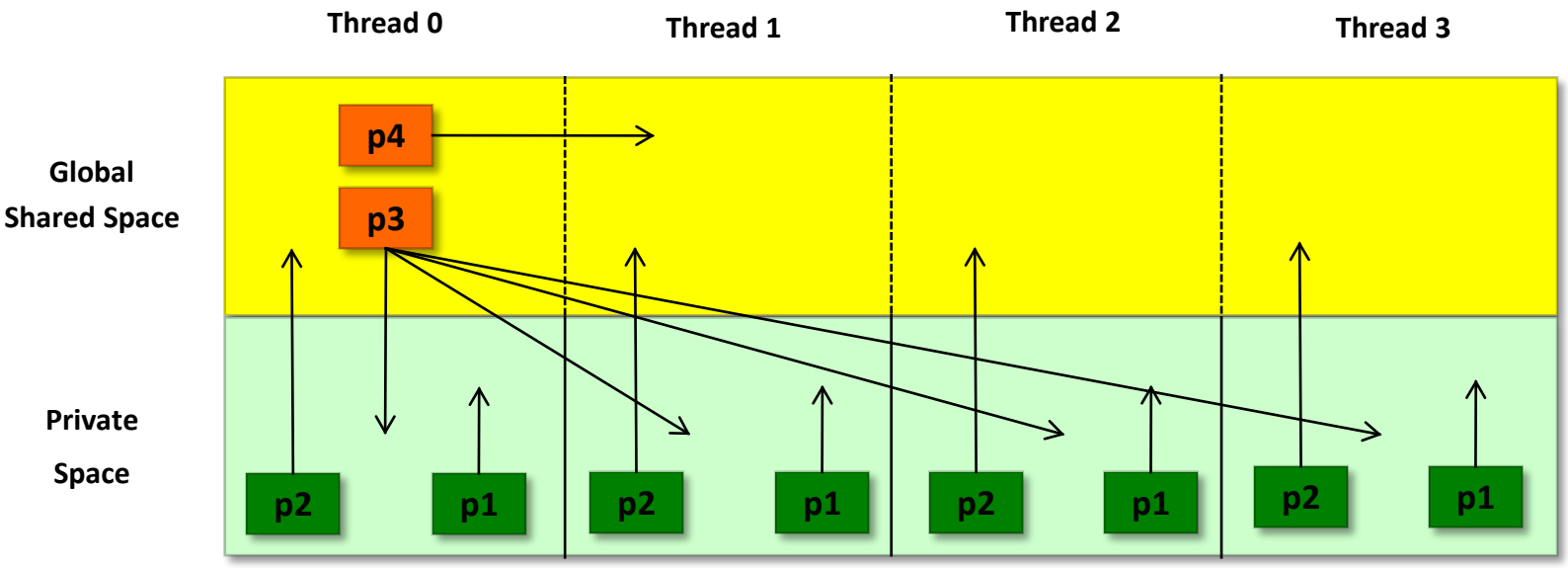
```
shared int x;    //shared variable; allocated with affinity to Thread 0
int main() {
  int y;         //private variable
}
```

UPC: Memory Model



- Shared Array: cyclic layout (by default)
  - shared int A1[THREADS]
  - shared int A2[2][THREADS]
- Shared Array: block layout
  - shared [*] int A3[2*THREADS]
  - shared [2] int A4[2][THREADS]

UPC: Pointers



- Private pointer to private space: int * p1;
  - Fast as normal C pointers
- Private pointer to shared space: shared int * p2; /*a pointer-to-shared*/
  - Slower for test whether the address is local or for communication
- Shared pointer to private space: int * shared p3;
  - Not recommended
- Shared pointer to shared space: shared int * shared p4;
  - Used for shared linked structures

UPC: Dynamic Memory Management

- shared void *upc_all_alloc (size_t nblocks, size_t nbytes);
  - Collective function; the call returns the same pointer on all threads
  - Allocates shared space compatible with the following declaration:

    shared [nbytes] char[nblocks * nbytes]
- shared void *upc_global_alloc (size_t nblocks, size_t nbytes);
  - Not a collective function
  - Allocate shared space compatible with the declaration:

    shared [nbytes] char[nblocks * nbytes]
- shared void *upc_alloc (size_t nbytes);
  - Not a collective function; like malloc() but returns a pointer-to-shared
  - Allocates shared space of at least nbytes bytes with affinity to the calling thread
- void upc_free (shared void *ptr);
  - Free the dynamically allocated shared storage pointed to by ptr

UPC: Consistency Model

- The ordering of shared operations is decided by user-controlled consistency models
- Strict consistency model
  - All threads observe the effects of strict accesses in a manner consistent with a single, global total order
  - #include<upc_strict.h> /* control at the program level*/
  - #pragma upc strict /* for a statement or a block of statements */
  - Type qualifiers: strict /* for a variable definition */
- Relaxed consistency model
  - Any sequence of purely relaxed shared access issued by a given thread in an execution may be arbitrarily reordered by the compiler or runtime
  - #include<upc_relaxed.h> /* control at the program level*/
  - #pragma upc relaxed /* for a statement or a block of statements */
  - Type qualifiers: relaxed /* for a variable definition */

UPC: Barriers and Locks

- Non-blocking barrier:

```
upc_notify;  /* upc_notify and upc_wait are collective functions */
…
upc_wait; /* the next collective after upc_notify must be upc_wait */
```

- Blocking barrier:

```
upc_barrier;
```

  – Equivalent to: {upc_notify barrier_value; upc_wait barrier_value;}

- Fence:

```
upc_fence;
```

  – All shared accesses issued before the fence are complete before any after it are issued

- Locks:

```
void upc_lock (upc_lock_t *ptr);
int upc_lock_attempt (upc_lock_t *ptr);
void upc_unlock (upc_lock_t *ptr);
```

  – Protect shared data from being accessed by multiple writers

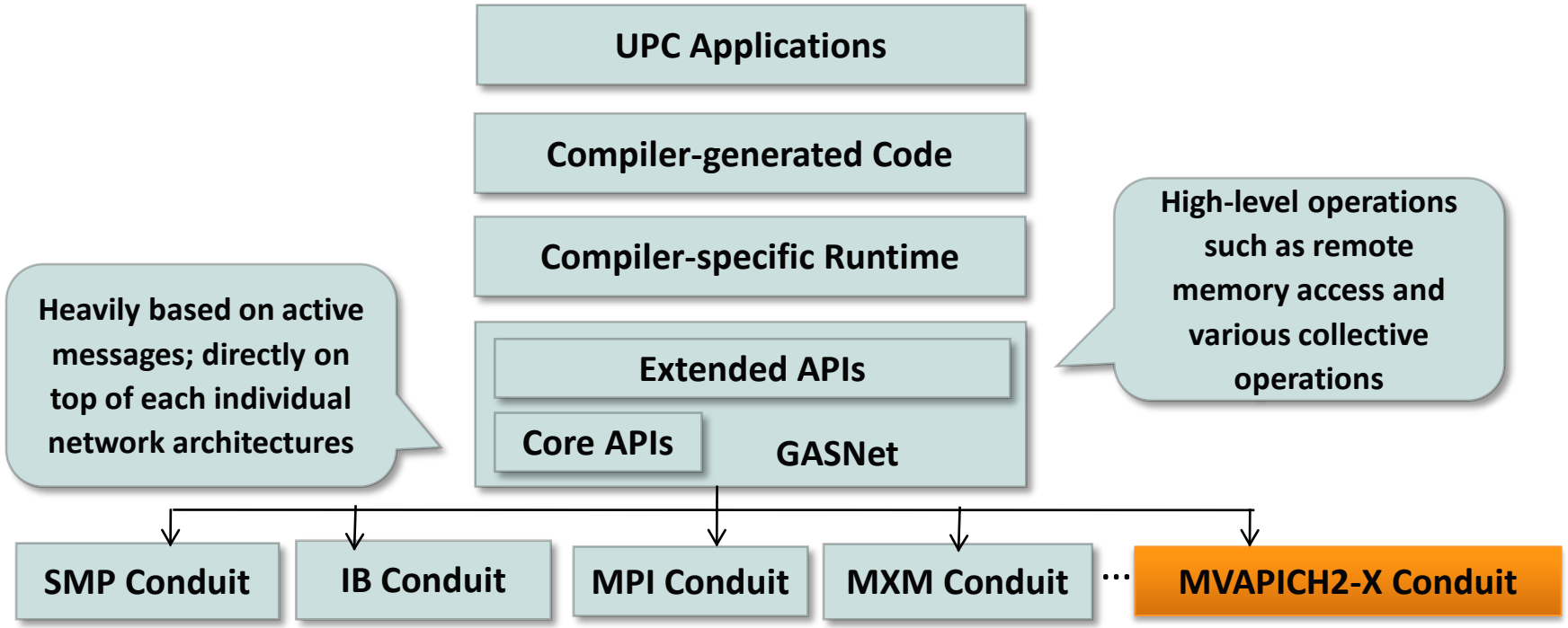
  – Locks are allocated by:

```
upc_lock_t *upc_global_lock_alloc (void); /* non-collective */
upc_lock_t *upc_all_lock_alloc (void); /* collective */
```

UPC Data Movement and Work Sharing

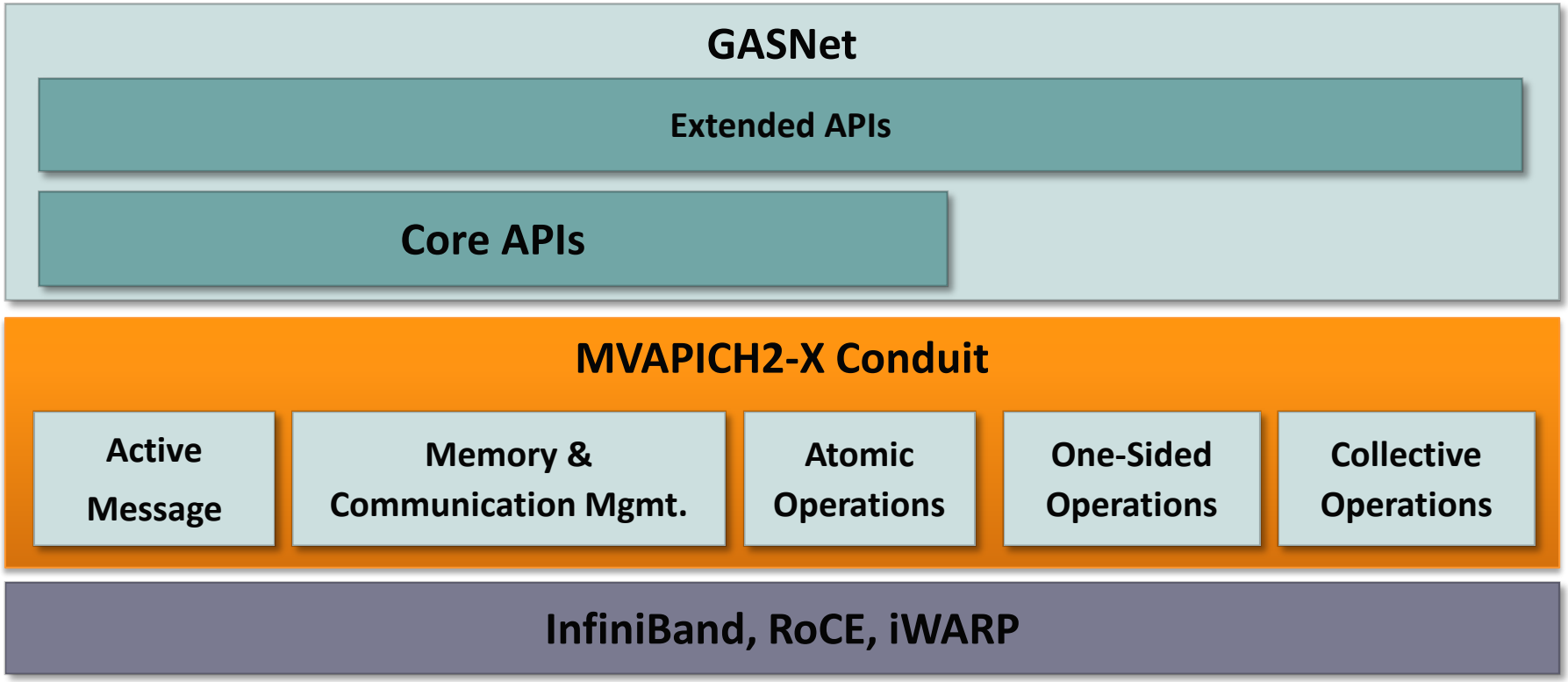
- Data Movement
  - **upc_memput:** Write data to remote memory location
    - void upc_memput(shared void *dst, const void *src, size_t n);
  - **upc_memget:** Read data from remote memory location
    - void upc_memget(void *dst, shared const void *src, size_t n);
  - **upc_memset:** Fills remote memory with the value ‘c’
    - void upc_memset(shared void *dst, int c, size_t n);
  - **Shared variable assignments**
    - Compiler translates these into remote memory operations
- Work Sharing
  - upc_forall(expression1; expression2; expression3; affinity)
  - The affinity field specifies the executions of the loop body that are to be performed by a thread.

Berkeley UPC Runtime Overview



- GASNet (Global-Address Space Networking) is a language-independent, low-level networking layer that provides support for PGAS language
- Support multiple networks through different conduit: MVAPICH2-X Conduit is available in MVAPICH2-X release, which support UPC/OpenMP/MPI on InfiniBand

MVAPICH2-X Conduit Support to GASNet



- Support core APIs and extended APIs through various utility functions
- Fully utilize InfiniBand features
- In Berkeley UPC Runtime, UPC threads can be mapped to either an OS process or an OS pthread

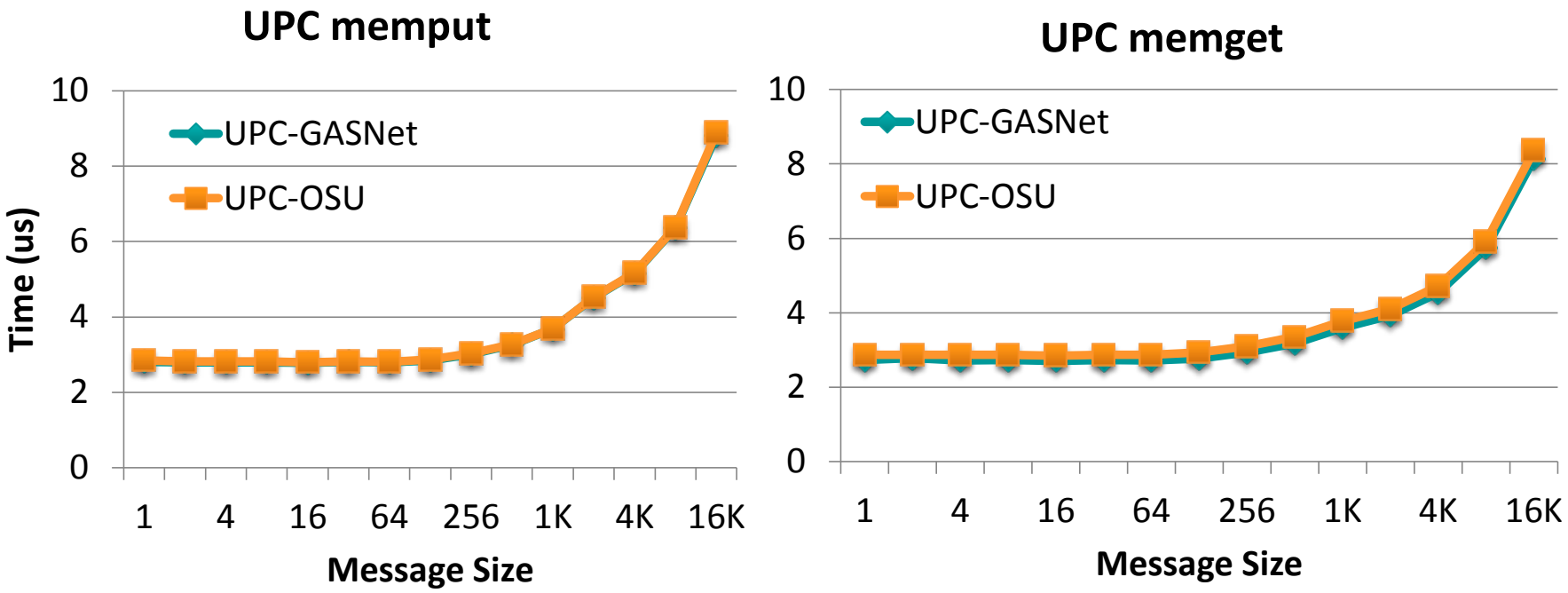
J. Jose, M. Luo, S. Sur and D. K. Panda, Unifying UPC and MPI Runtimes: Experience with MVAPICH, PGAS 2012

Support for UPC Operations in OSU Micro-Benchmarks (OMB)

- Point-to-point benchmarks
  - osu upc memput – Put latency
  - osu upc memget - Get latency
- Collective benchmarks
  - osu_upc_all_barrier – Barrier Latency
  - osu_upc_all_broadcast – Broadcast Latency
  - osu_upc_all_exchange – Exchange (Alltoall) Latency
  - osu_upc_all_gather – Gather Latency
  - osu_upc_all_gather_all – AllGather Latency
  - osu_upc_all_reduce – Reduce Latency
  - osu_upc_all_scatter – Scatter Latency
- OMB is publicly available from:

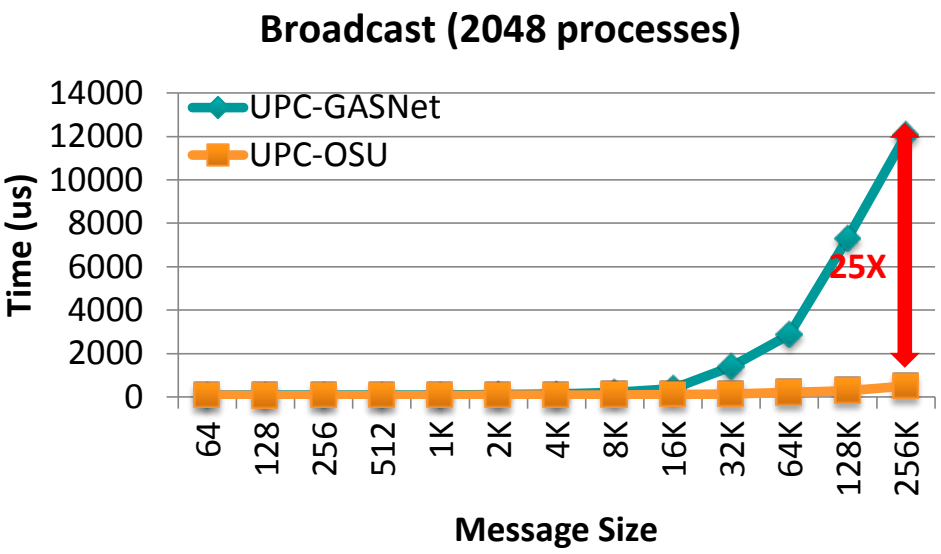
  http://mvapich.cse.ohio-state.edu/benchmarks/

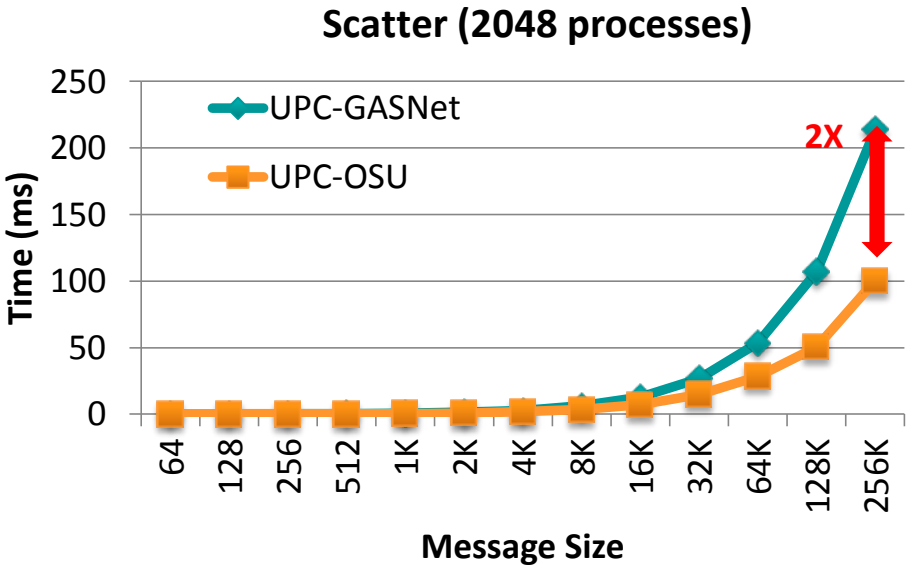
UPC Micro-benchmark Performance

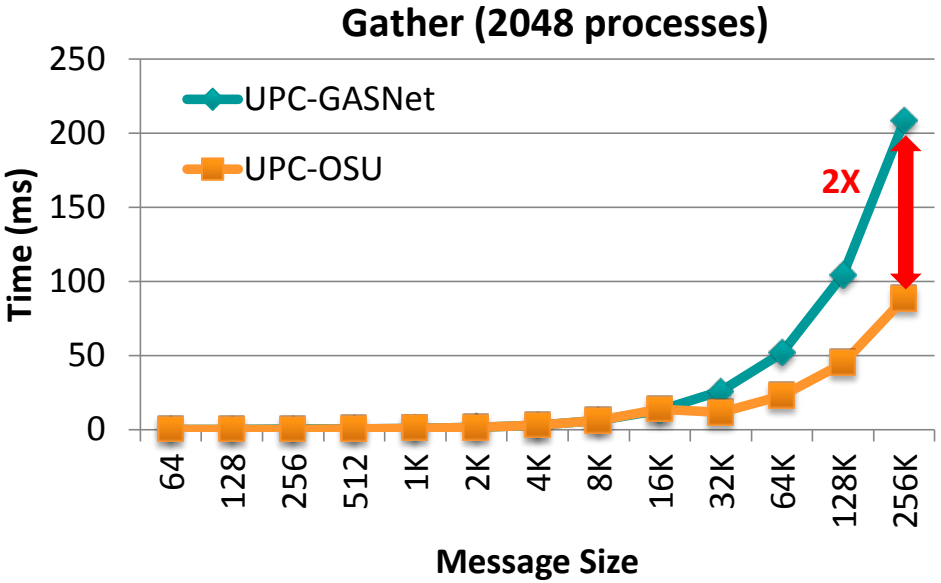


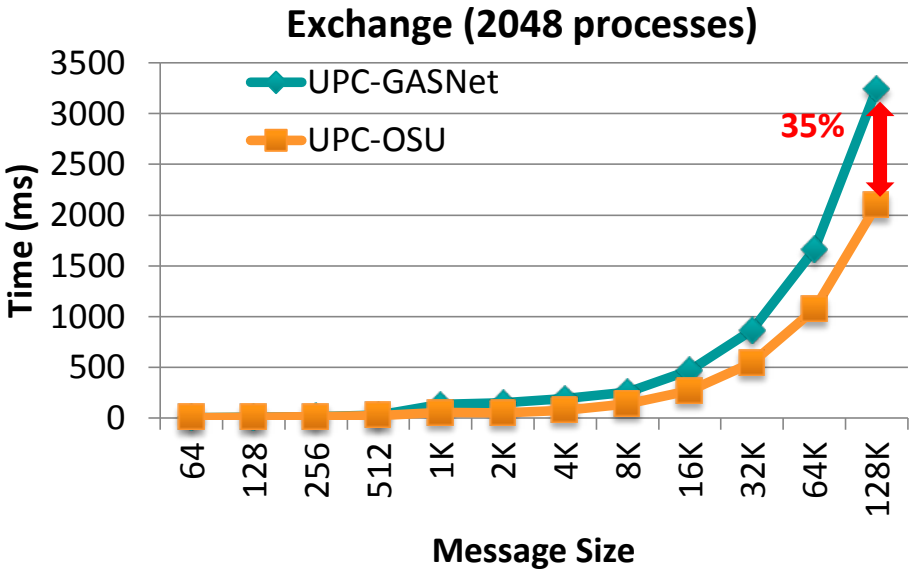
- OSU UPC micro-benchmarks (OMB v4.2)
- Similar performance for UPC memput/memget performance for UPC-OSU and UPC-GASNet-IB conduits

UPC Collective Performance









J. Jose, K. Hamidouche, J. Zhang, A. Venkatesh, and D. K. Panda, Optimizing Collective Communication in UPC (HiPS'14, in association with IPDPS'14)

Presentation Overview

- **PGAS Programming Models and Runtimes**
  - PGAS Languages: Unified Parallel C (UPC)
  - PGAS Libraries: OpenSHMEM
- Hybrid MPI+PGAS Programming Models and Benefits
- High-Performance Runtime for Hybrid MPI+PGAS Models
- Application-level Case Studies and Evaluation

SHMEM

- SHMEM: Symmetric Hierarchical MEMory library
- One-sided communications library – had been around for a while
- Similar to MPI, processes are called PEs, data movement is explicit through library calls
- Provides globally addressable memory using symmetric memory objects (more in later slides)
- Library routines for
  - Symmetric object creation and management
  - One-sided data movement
  - Atomics
  - Collectives
  - Synchronization

OpenSHMEM

- SHMEM implementations – Cray SHMEM, SGI SHMEM, Quadrics SHMEM, HP SHMEM, GSHMEM
- Subtle differences in API, across versions – example:

	SGI SHMEM	Quadrics SHMEM	Cray SHMEM
Initialization	*start_pes(0)*	*shmem_init*	*start_pes*
Process ID	*_my_pe*	*my_pe*	*shmem_my_pe*

- Made applications codes non-portable
- OpenSHMEM is an effort to address this:

"A new, open specification to consolidate the various extant SHMEM versions into a widely accepted standard." – OpenSHMEM Specification v1.0

by University of Houston and Oak Ridge National Lab

SGI SHMEM is the baseline

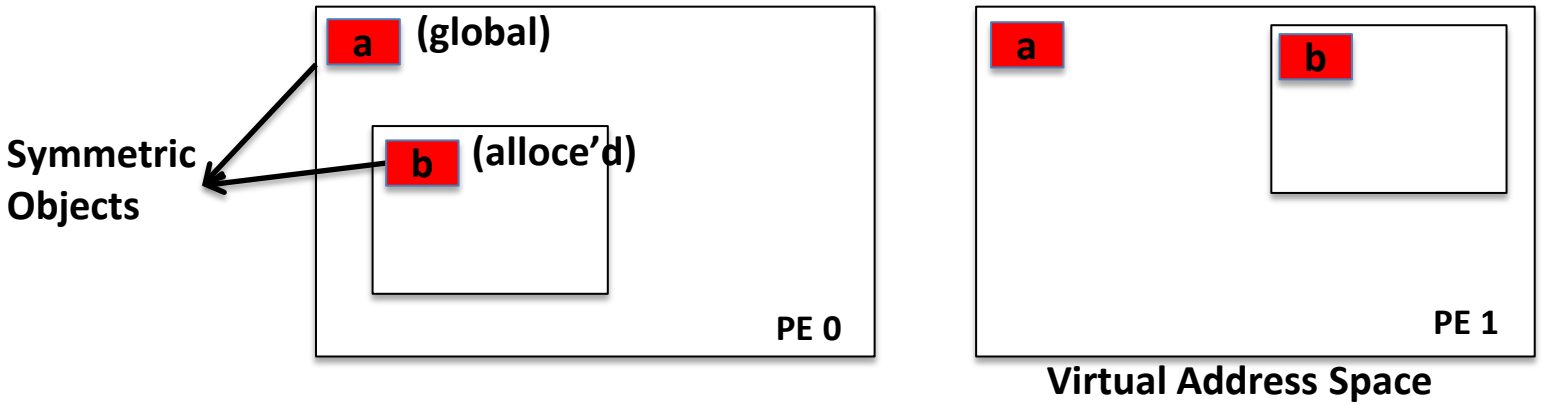
OpenSHMEM Hello World

- Hello World:

```
#include <shmem.h>
#include <stdio.h>
int main() {
  start_pes(0);
  fprintf(stderr, ”Hello from thread %d of %d\n", _my_pe(),
_num_pes());
  return 0;
}
```

The OpenSHMEM Memory Model

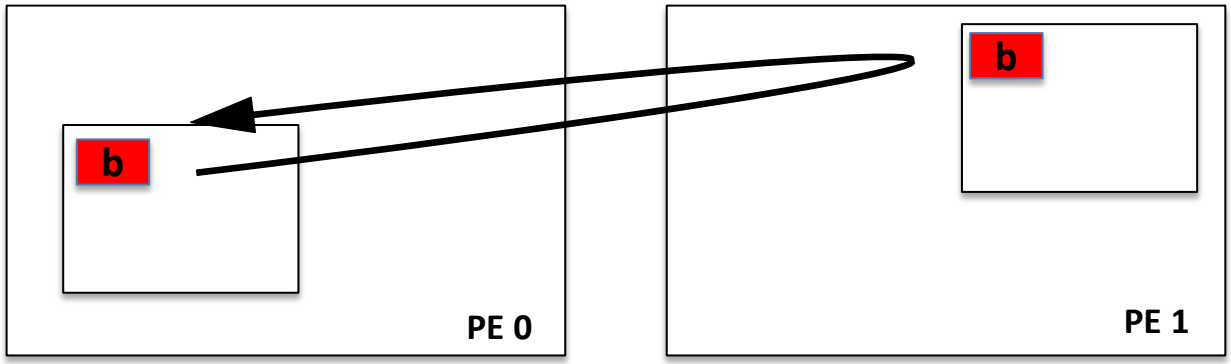
- Symmetric data objects
  - Global Variables
  - Allocated using collective *shmalloc, shmemalign, shrealloc* routine



- Globally addressable – objects have same
  - Type
  - Size
  - Same virtual address or offset at all PEs
  - Address of a remote object can be calculated based on info of local object

Data Movement: Basic

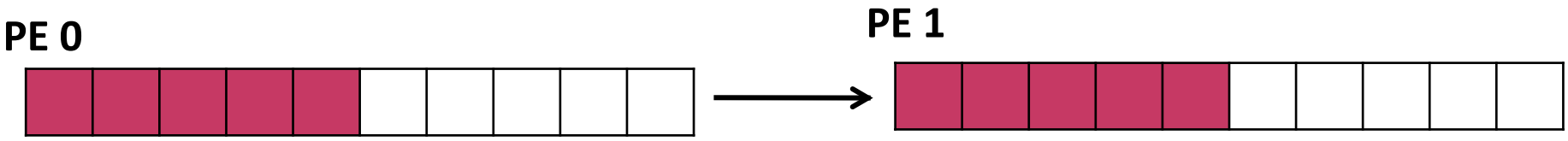
- Put and Get – single element
  - void shmem_*TYPE*_p (*TYPE* *ptr, int PE)
  - void shmem_*TYPE*_g (*TYPE* *ptr, int PE)
  - *TYPE* can be short, int, long, float, double, longlong, longdouble




```
int *b;
b =  (int *) shmalloc (sizeof(int));

if ((_my_pe() == 0) {
    shmem_int_g (b, 1);
}
```

Data Movement: Contiguous

- Block Put and Get – Contiguous
  - void shmem_*TYPE*_put (TYPE* target, const TYPE*source, size_t nelems, int pe)
    - *TYPE* can be char, short, int, long, float, double, longlong, longdouble
  - shmem_put*SIZE* – elements of SIZE: 32/64/128
  - shmem_putmem - bytes
  - Similar get operations

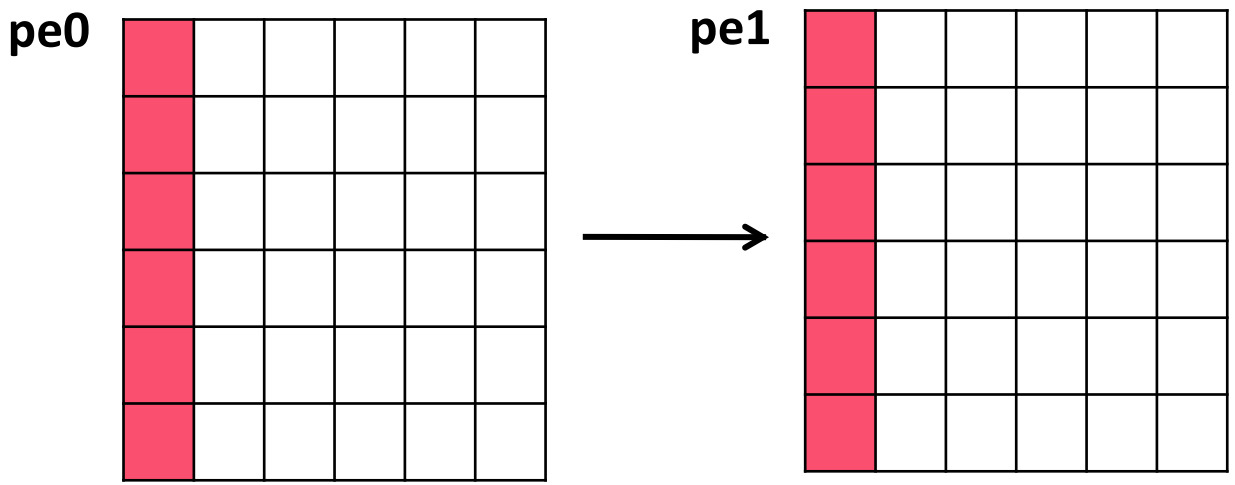



```
int *b;
b =  (int *) shmalloc (10*sizeof(int));

if ((_my_pe() == 0) {
   shmem_int_put (b, b, 5, 1);
}
```

Data Movement: Non-contiguous

- Strided Put and Get
  - shmem_*TYPE*_iput (TYPE* target, const TYPE*source, ptrdiff_t tst, ptrdiff_t sst, size_t nelems, int pe)
    - sst is stride at source, tst is stride at target
    - *TYPE* can be char, short, int, long, float, double, longlong, longdouble
  - shmem_iput*SIZE*
    - *SIZE* can be 32/64/128
  - Similar get operations

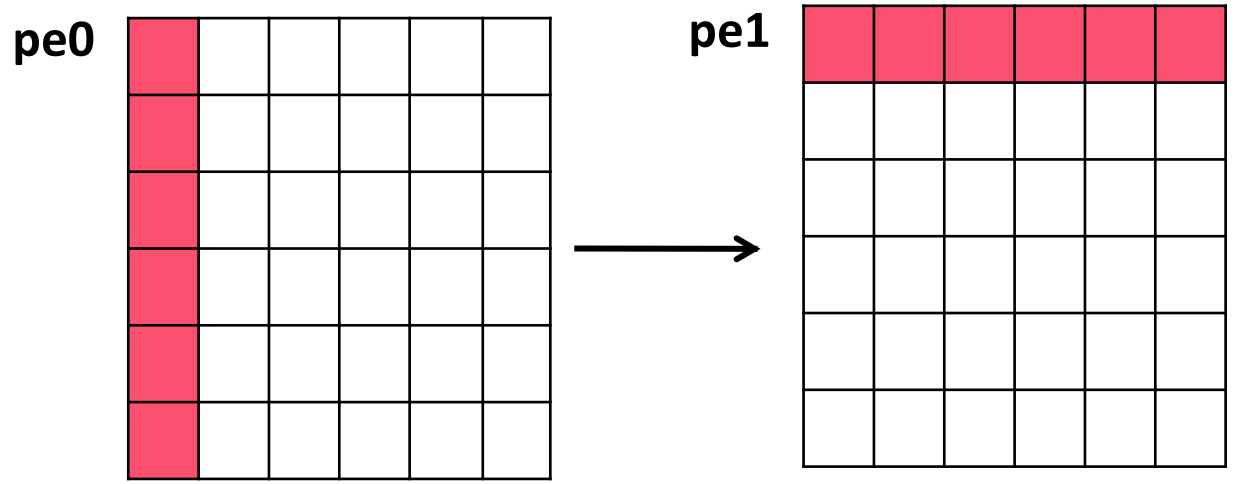
Data Movement: Non-contiguous



symmetric object 't'

6x6 integer array

Target stride: 6, Source stride: 6, Num. of elements: 6
shmem_int_iput(t, t, 6, 6, 6, 1)



Target stride: 1, Source stride: 6, Num. of elements: 6
shmem_int_iput(t, t, 1, 6, 6, 1)

Data Movement - Completion

- When Put operations return
  - Data has been copied out of the source buffer object
  - Not necessarily written to the target buffer object
  - Additional synchronization to ensure remote completion
- When Get operations return
  - Data has been copied into the local target buffer
  - Ready to be used

Collective Synchronization

- Barrier ensures completion of all previous operations
- Global Barrier
  - void shmem_barrier_all()
  - Does not return until called by all PEs
- Group Barrier
  - Involves only an *"ACTIVE SET"* of PEs
  - Does not return until called by all PEs in the *"ACTIVE SET"*
  - void shmem_barrier (int PE_start, */* first PE in the set */*
    int logPE_stride, */* distance between two PEs*/*
    int PE_size, */*size of the set*/*
    long *pSync /*symmetric work array*/);
  - pSync allows for overlapping collective communication

One-sided Synchronization

- Fence
  - void shmem_fence (void)
  - Enforces ordering on Put operations issued by a PE to each destination PE
  - Does not ensure ordering between Put operations to multiple PEs
- Quiet
  - void shmem_quiet (void)
  - Ensures remote completion of Put operations to all PEs
- Other point-to-point synchronization
  - shmem_wait and shmem_wait_until – poll on a local variable

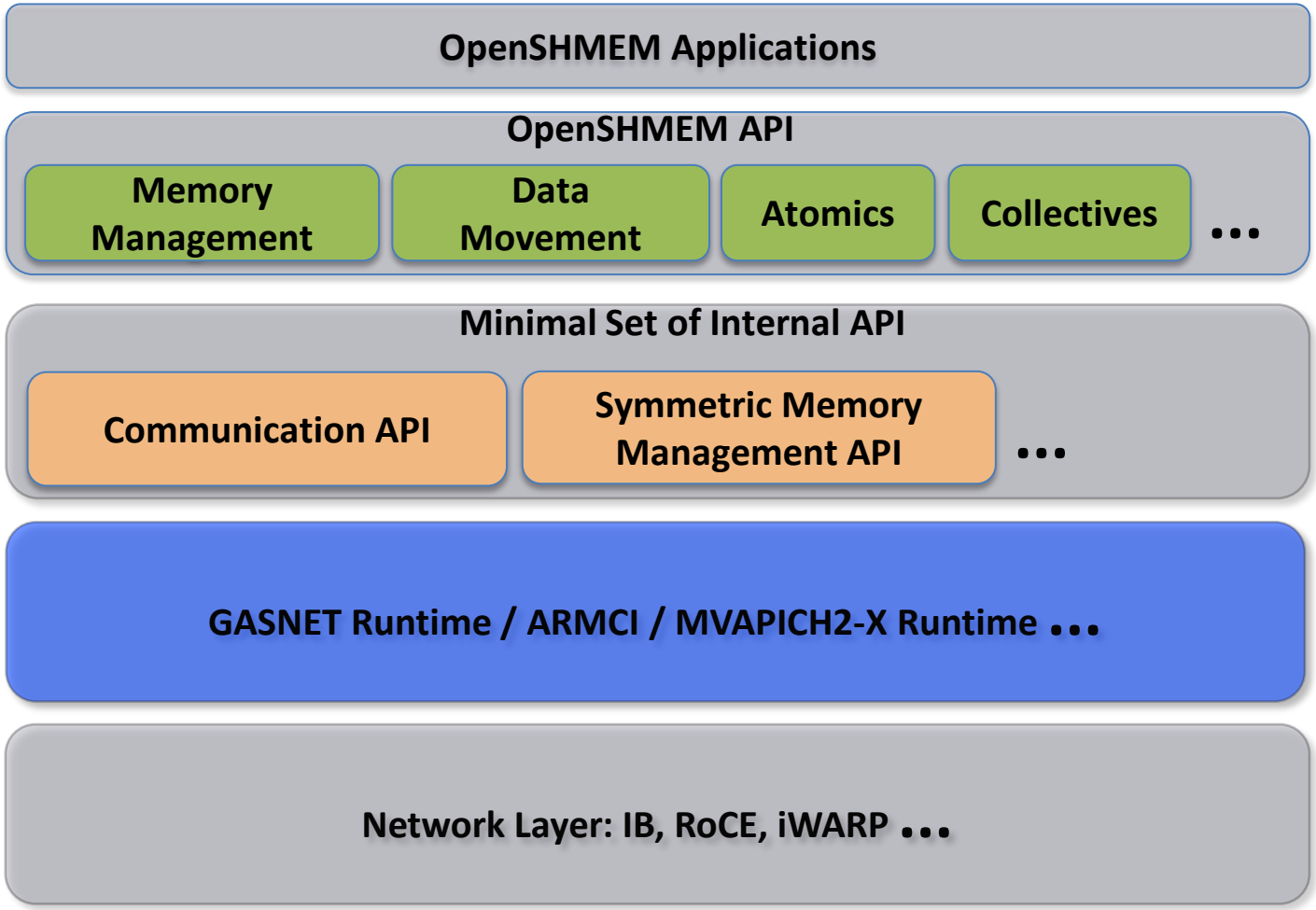
Collective Operations and Atomics

- Broadcast – one-to-all
- Collect – allgather
- Reduction – allreduce (and, or, xor; max, min; sum, product)
- Work on an active set – start, stride, count
- Unconditional - Swap Operation
  - long shmem_swap (long *target, *long* value, int pe)
  - *TYPE* shmem_TYPE_swap (*TYPE* *target, *TYPE* value, int pe)
  - *TYPE* can be int, long, longlong, float, double
- Conditional - Compare and Swap Operation
- Arithmetic – Fetch & Add, Fetch & Increment, Add, Increment

Remote Pointer Operations

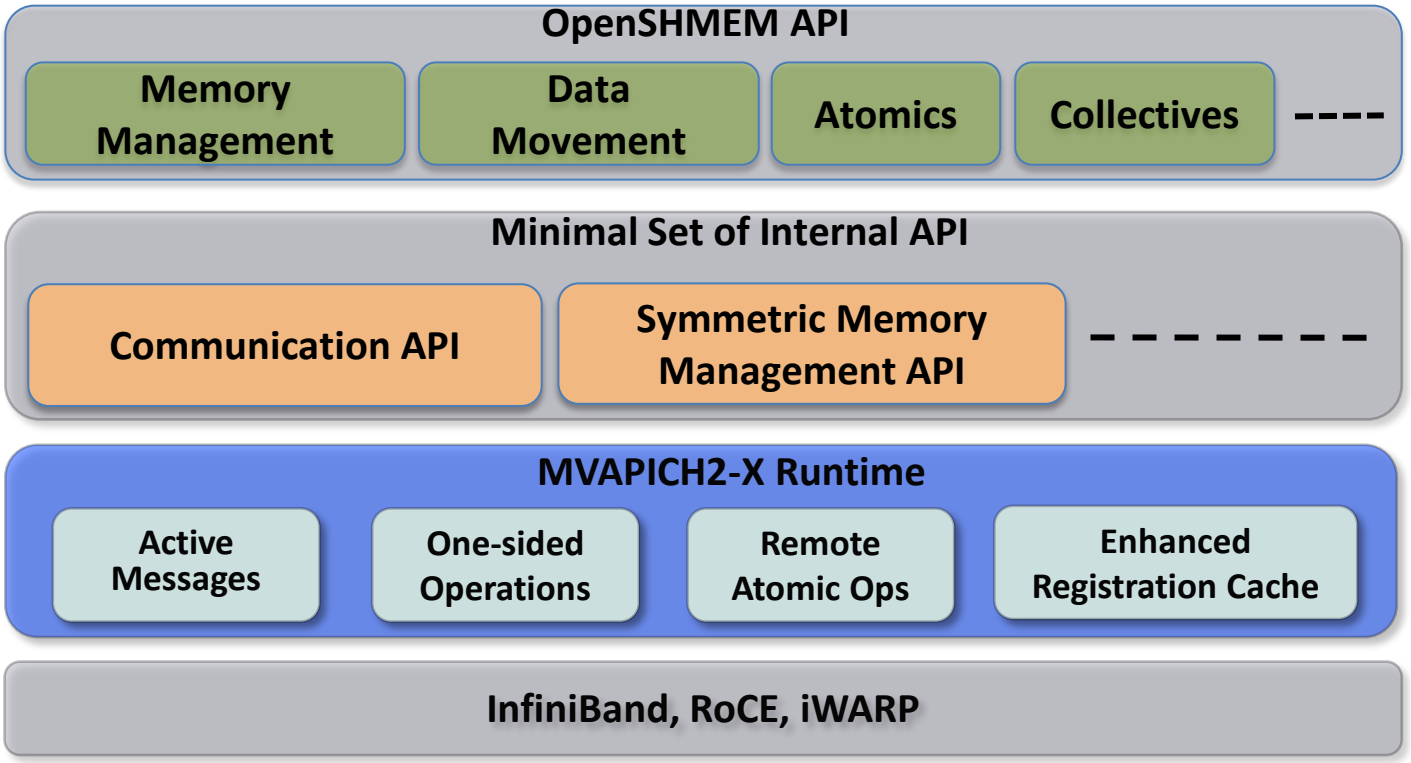
- **void *shmem_ptr (void *target, int pe)**
  - Allows direct load/stores on remote memory
  - Useful when PEs are running on same node
  - Not supported in all implementations
  - Returns NULL if not accessible for loads/stores

OpenSHMEM Reference Implementation Framework



Reference: OpenSHMEM: An Effort to Unify SHMEM API Library Development , Supercomputing 2010

OpenSHMEM Design in MVAPICH2-X



- OpenSHMEM Stack based on OpenSHMEM Reference Implementation
- OpenSHMEM Communication over MVAPICH2-X Runtime
  - Uses active messages, atomic and one-sided operations and remote registration cache

J. Jose, K. Kandalla, M. Luo and D. K. Panda, Supporting Hybrid MPI and OpenSHMEM over InfiniBand: Design and Performance Evaluation, Int'l Conference on Parallel Processing (ICPP '12), September 2012.

Implementations for InfiniBand Clusters

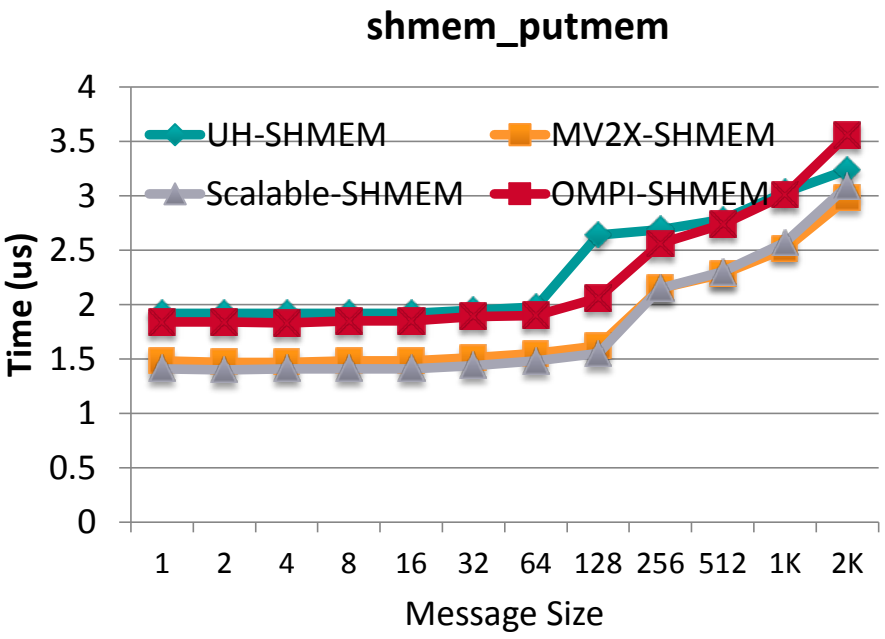
- Reference Implementation
  - University of Houston
  - Based on the GASNet runtime
- MVAPICH2-X
  - The Ohio State University
  - Uses the upper layer of reference implementations
  - Derives the runtime from widely used MVAPICH2 MPI library
  - Available for download: http://mvapich.cse.ohio-state.edu/download/mvapich2x
- OMPI-SHMEM
  - Based on OpenMPI runtime
  - Available in OpenMPI 1.7.5
- ScalableSHMEM
  - Mellanox technologies

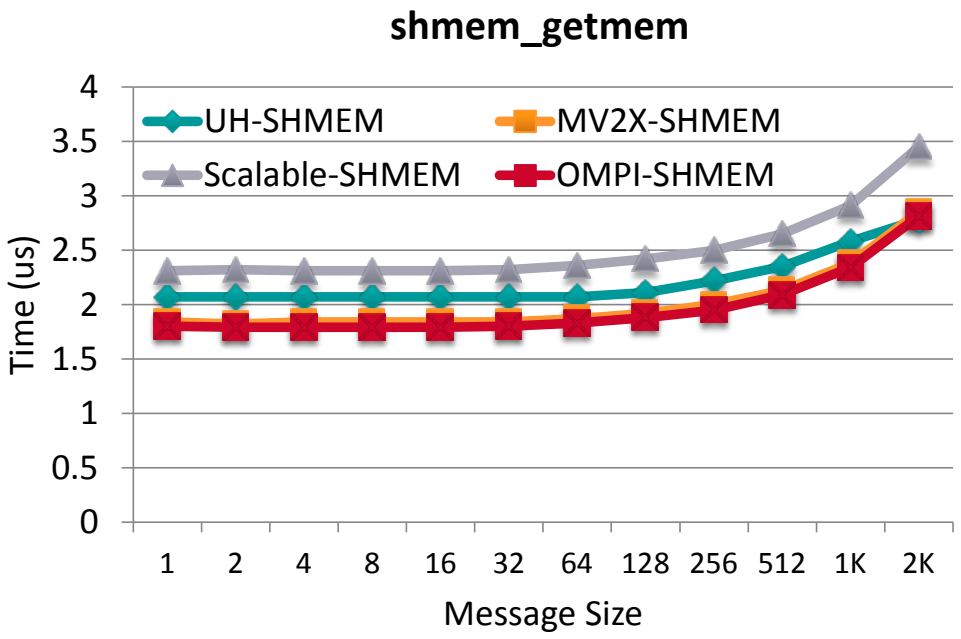
Support for OpenSHMEM Operations in OSU Micro-Benchmarks (OMB)

- Point-to-point Operations
  - osu_oshm_put – Put latency
  - osu_oshm_get – Get latency
  - osu_oshm_put_mr – Put message rate
  - osu_oshm_atomics – Atomics latency
- Collective Operations
  - osu_oshm_collect – Collect latency
  - osu_oshm_broadcast – Broadcast latency
  - osu_oshm_reduce - Reduce latency
  - osu_oshm_barrier - Barrier latency
- OMB is publicly available from:

  http://mvapich.cse.ohio-state.edu/benchmarks/

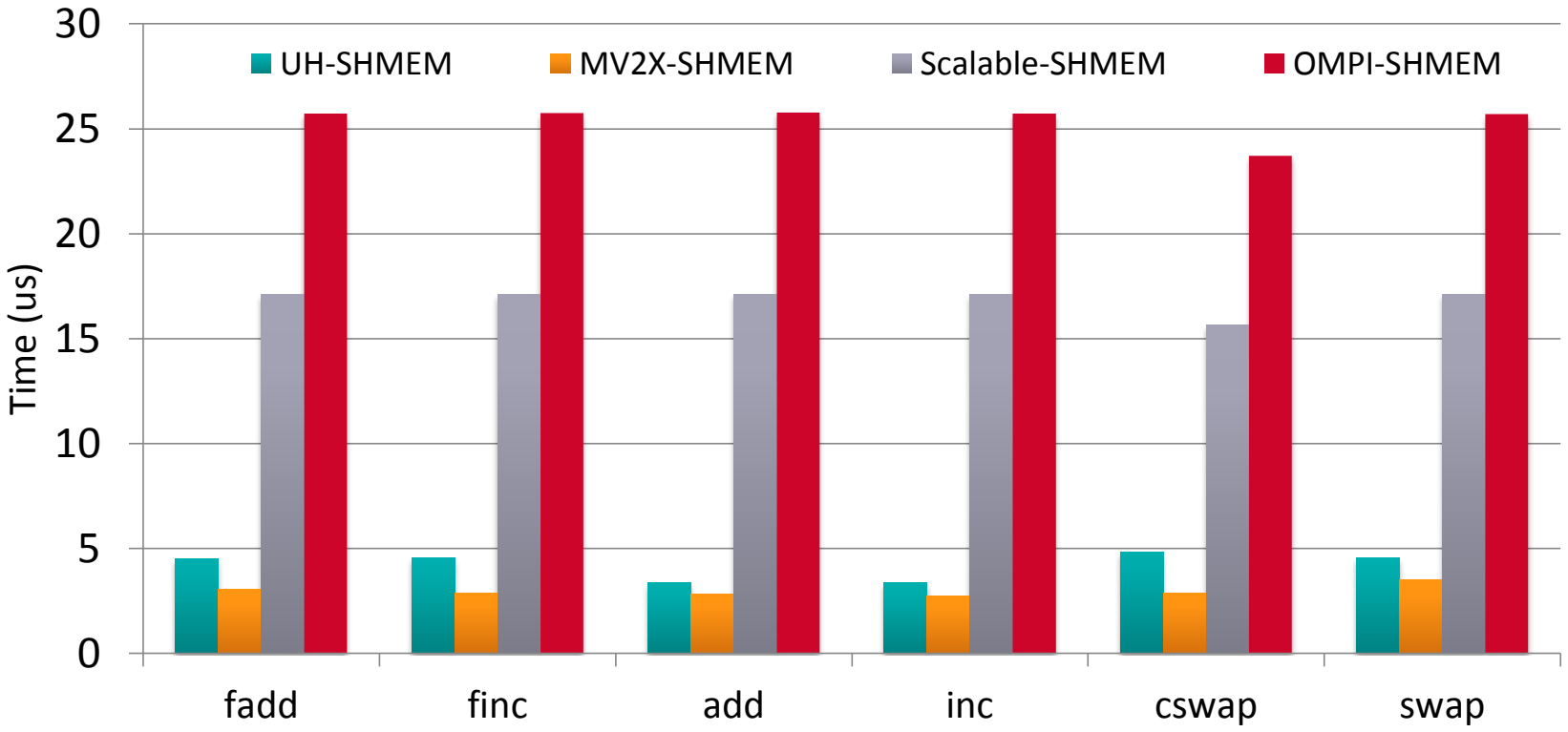
OpenSHMEM Data Movement: Performance





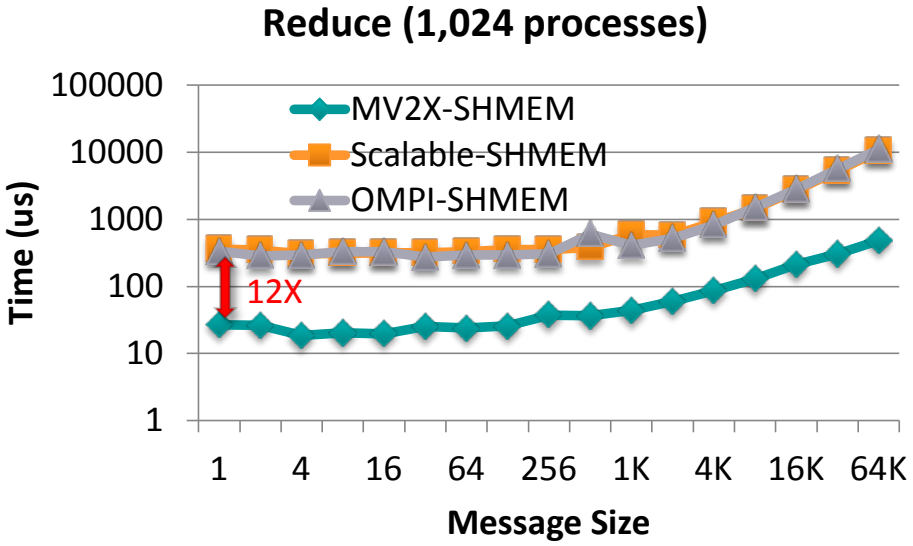
- OSU OpenSHMEM micro-benchmarks
  http://mvapich.cse.ohio-state.edu/benchmarks/
- Slightly better performance for putmem and getmem with MVAPICH2-X

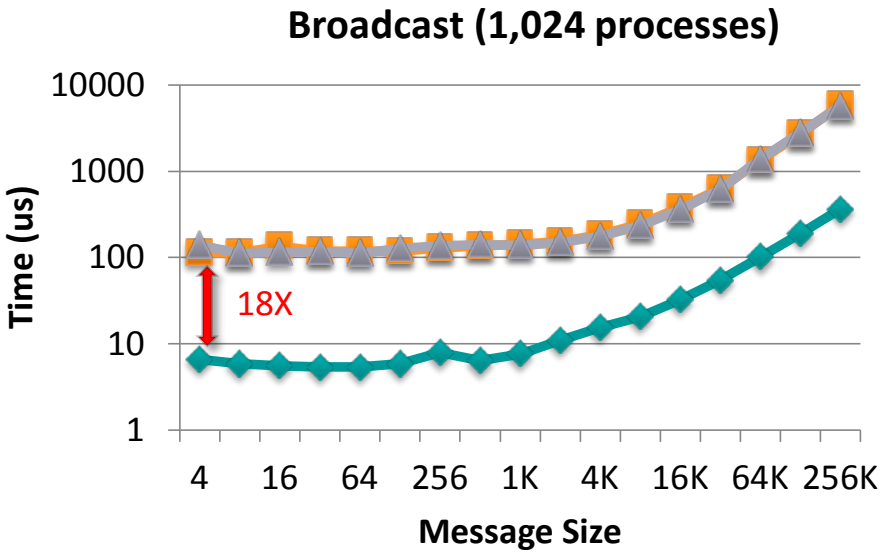
OpenSHMEM Atomic Operations: Performance

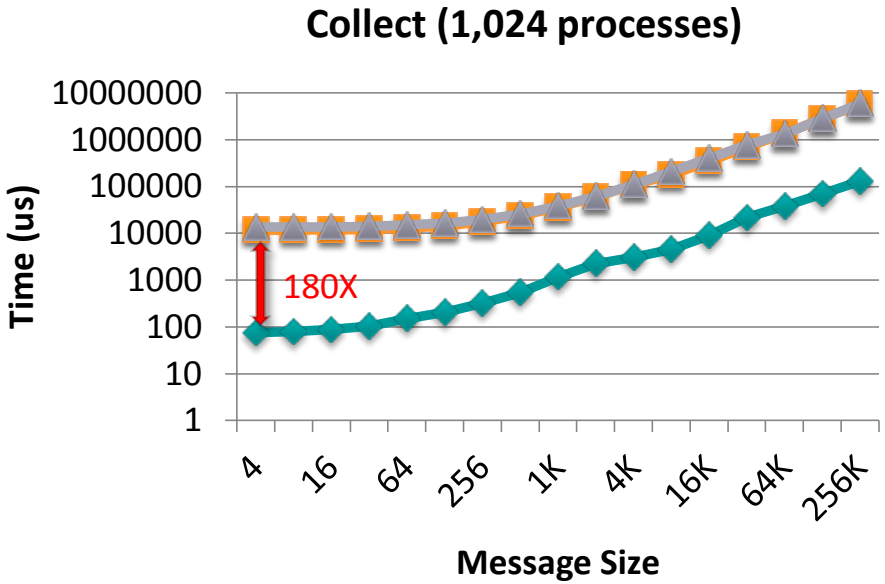


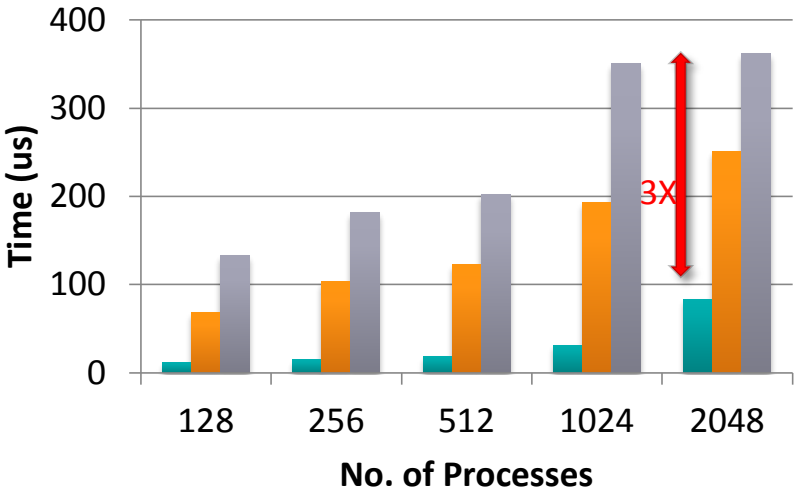
- OSU OpenSHMEM micro-benchmarks
- MV2-X SHMEM performs up to **40%** better compared to UH-SHMEM

Collective Communication: Performance







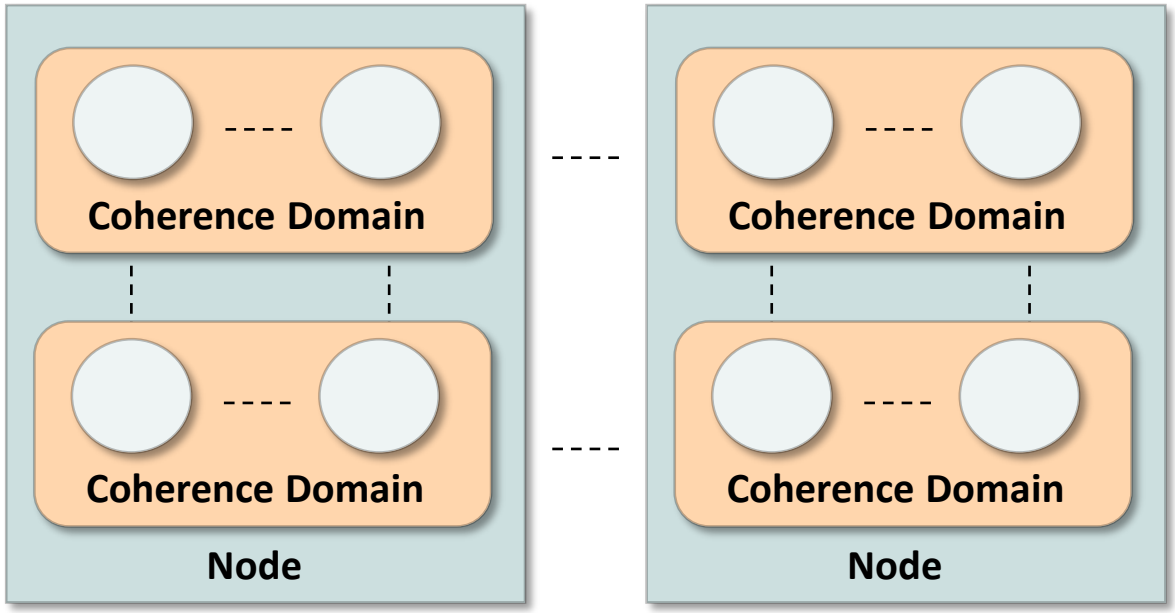


J. Jose, K. Kandalla, S. Potluri, J. Zhang, and D. K. Panda, Optimizing Collective Communication in OpenSHMEM, PGAS'13.

Presentation Overview

- PGAS Programming Models and Runtimes
  - PGAS Languages: Unified Parallel C (UPC)
  - PGAS Libraries: OpenSHMEM
- Hybrid MPI+PGAS Programming Models and Benefits
- High-Performance Runtime for Hybrid MPI+PGAS Models
- Application-level Case Studies and Evaluation

Architectures for Exascale Systems



Hypothetical Future Architecture*

- Modern architectures have increasing number of cores per node, but have limited memory per core
  - Memory bandwidth per core decreases
  - Network bandwidth per core decreases
  - Deeper memory hierarchy
  - More parallelism within the node

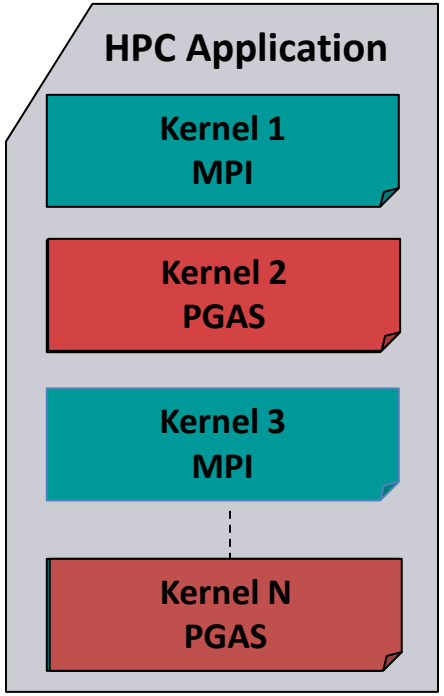
**Marc Snir, Keynote Talk – Programming Models for High Performance Computing, Cluster, Cloud and Grid Computing (CCGrid 2013)*

Maturity of Runtimes and Application Requirements

- MPI has been the most popular model for a long time
  - Available on every major machine
  - Portability, performance and scaling
  - Most parallel HPC code is designed using MPI
  - Simplicity - structured and iterative communication patterns
- PGAS Models
  - Increasing interest in community
  - Simple shared memory abstractions and one-sided communication
  - Easier to express irregular communication
- Need for hybrid MPI + PGAS
  - Application can have kernels with different communication characteristics
  - Porting only part of the applications to reduce programming effort

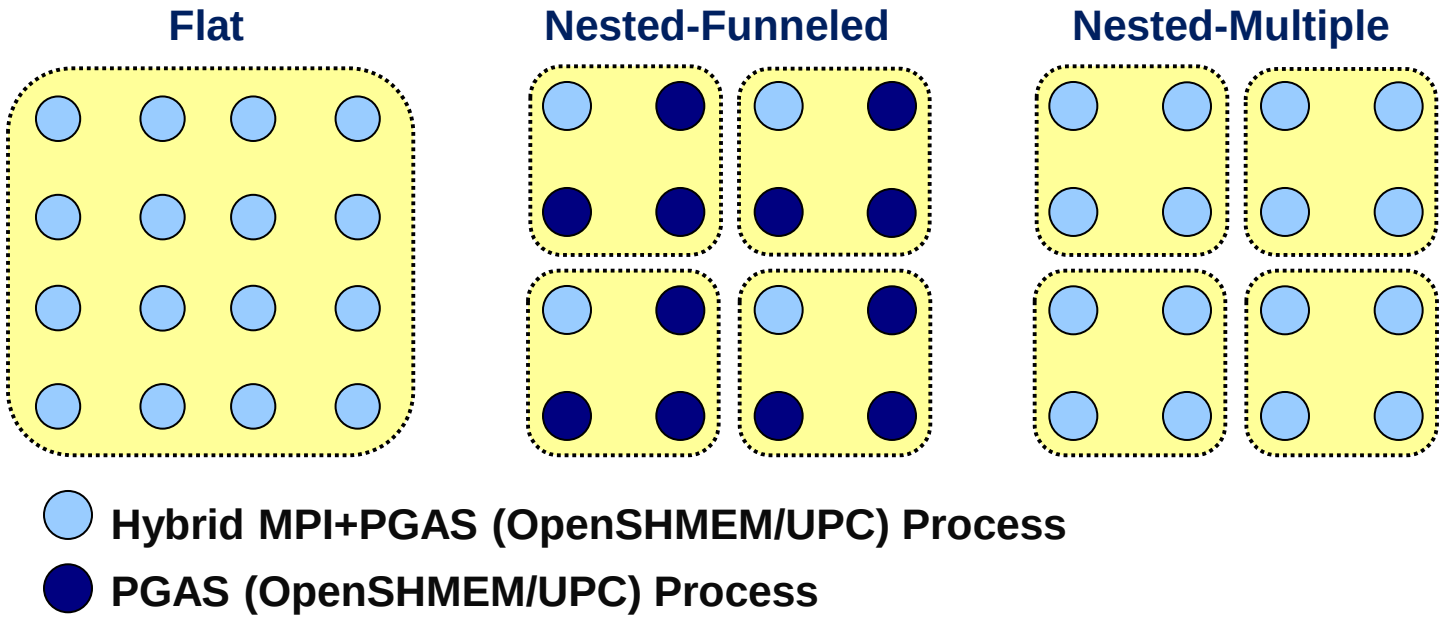
Hybrid (MPI+PGAS) Programming

- Application sub-kernels can be re-written in MPI/PGAS based on communication characteristics
- Benefits:
  - Best of Distributed Computing Model
  - Best of Shared Memory Computing Model
- Exascale Roadmap*:
  - "Hybrid Programming is a practical way to program exascale systems"



** The International Exascale Software Roadmap, Dongarra, J., Beckman, P. et al., Volume 25, Number 1, 2011, International Journal of High Performance Computer Applications, ISSN 1094-3420*

Hybrid MPI+PGAS Programming Model: Alternatives



- Many possible ways to combine MPI
- Focus on:
  - Flat: One global address space
  - Nested-multiple: Multiple global address spaces (UPC groups)

J. Dinan, P. Balaji, E. Lusk, P. Sadayappan and R. Thakur, Hybrid Parallel Programming with MPI and Unified Parallel C, ACM Computing Frontiers, 2010

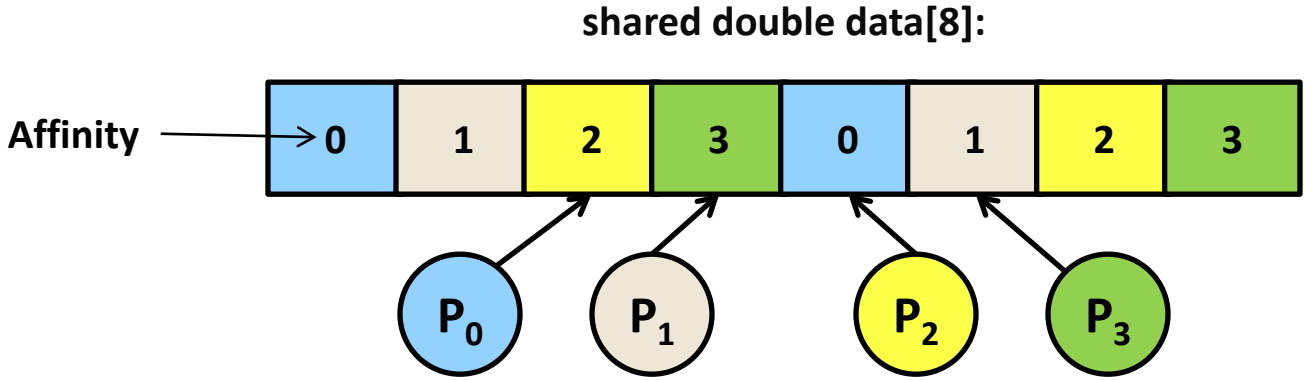
Simple MPI + OpenSHMEM Hybrid Example

```
int main(int c, char *argv[])
{
    int rank, size;

    /* SHMEM init */
    start_pes(0);

    /* fetch-and-add at root */
    shmem_int_fadd(&sum, rank, 0);

    /* MPI barrier */
    MPI_Barrier(MPI_COMM_WORLD);

    /* root broadcasts sum */
    MPI_Bcast(&sum, 1, MPI_INT, 0, MPI_COMM_WORLD);

    fprintf(stderr, "(%d): Sum: %d\n", rank, sum);

    shmem_barrier_all();
    return 0;
}
```

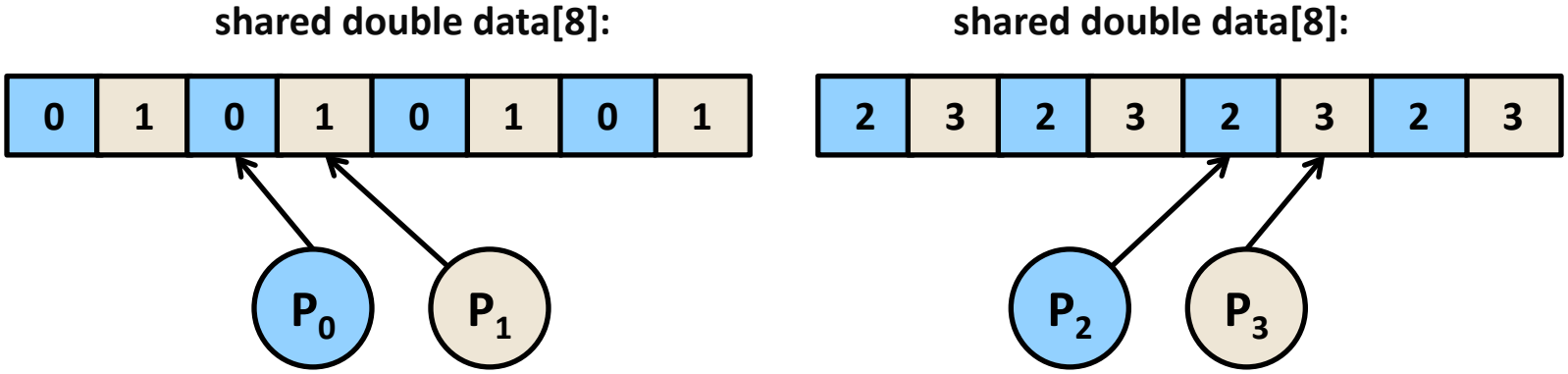
- **OpenSHMEM atomic fetch-add**
- **MPI_Bcast for broadcasting result**

Random Access Benchmark

- Threads access random elements of distributed shared array
- UPC Only: One copy distribute across all procs. Lesser local accesses



- Hybrid: Array is replicated on every group. All accesses are local
- Global co-ordination using MPI



J. Dinan, P. Balaji, E. Lusk, P. Sadayappan and R. Thakur, Hybrid Parallel Programming with MPI and Unified Parallel C, ACM Computing Frontiers, 2010

Hybrid 2D Heat benchmark

Pure OpenSHMEM version

```
while(true){
  <Gauss-Seidel Kernel>
  compute convergence locally
  sum_all =
   sumshmem_float_sum_to_all()
  Compute std. deviation
  shmem_broadcast(method to use)
}
```

Hybrid MPI+OpenSHMEM version

```
while(true){
  <Gauss-Seidel Kernel>
  compute convergence locally
  sum_all =
   MPI_Reduce()
  Compute std. deviation
   MPI_Bcast(method to use)
}
```

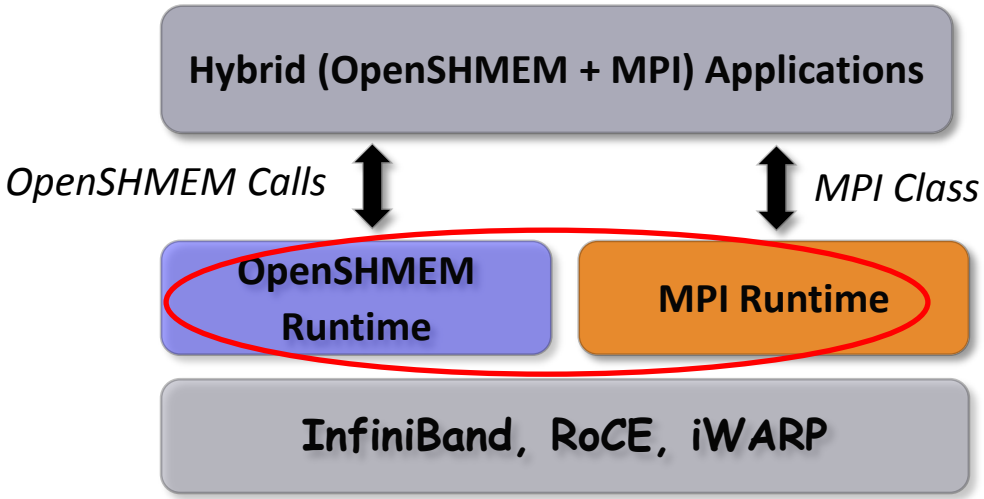
- **MPI Collectives have been optimized significantly**
  - **Performs better than OpenSHMEM collectives**
- **Improves performance of benchmark significantly**

Presentation Overview

- PGAS Programming Models and Runtimes
  - PGAS Languages: Unified Parallel C (UPC)
  - PGAS Libraries: OpenSHMEM
- Hybrid MPI+PGAS Programming Models and Benefits
- High-Performance Runtime for Hybrid MPI+PGAS Models
- Application-level Case Studies and Evaluation

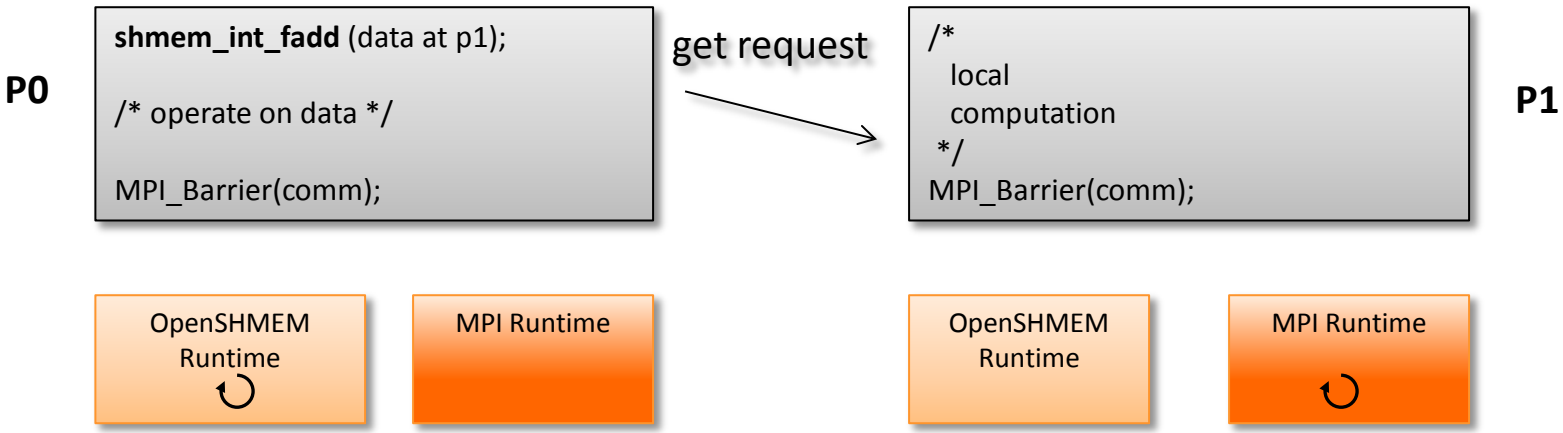
Current approaches for Hybrid Programming

- Layering one programming model over another
  - Poor performance due to semantics mismatch
  - MPI-3 RMA tries to address
- Separate runtime for each programming model



- Need more network and memory resources
- Might lead to deadlock!

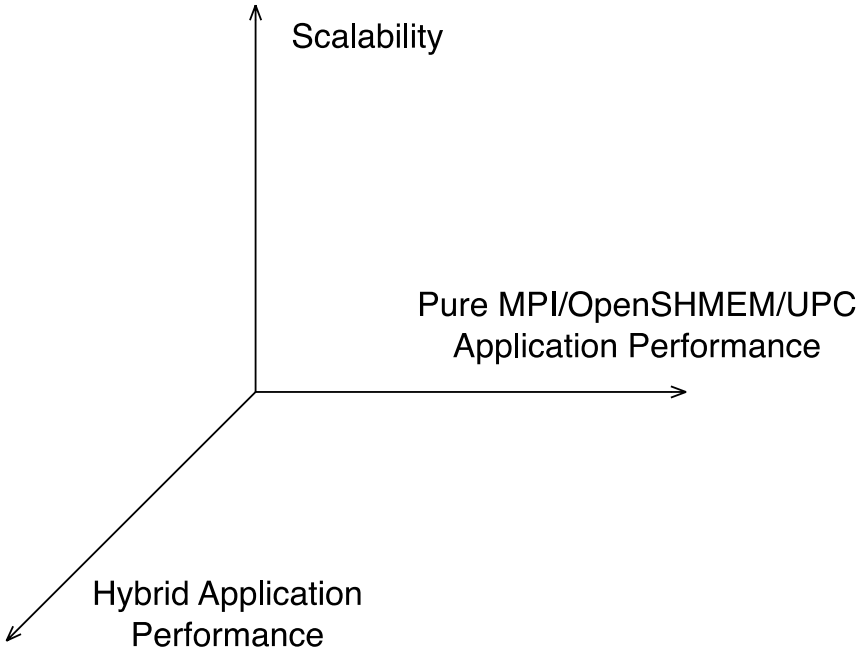
The Need for a Unified Runtime



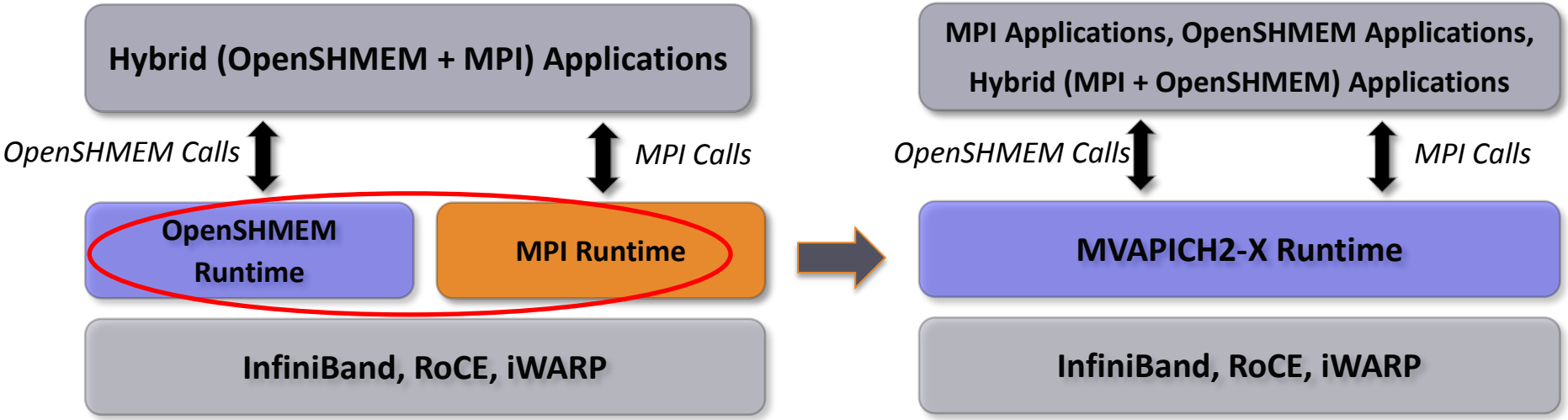
- Deadlock when a message is sitting in one runtime, but application calls the other runtime
- Prescription to avoid this is to barrier in one mode (either OpenSHMEM or MPI) before entering the other
- Or runtimes require progress threads
- Bad performance!!
- Similar issues for MPI + UPC applications over individual runtimes

Goals of a Unified Runtime

- Provide high performance and scalability for
  - MPI Applications
  - UPC Applications
  - OpenSHMEM Applications
  - Hybrid Applications



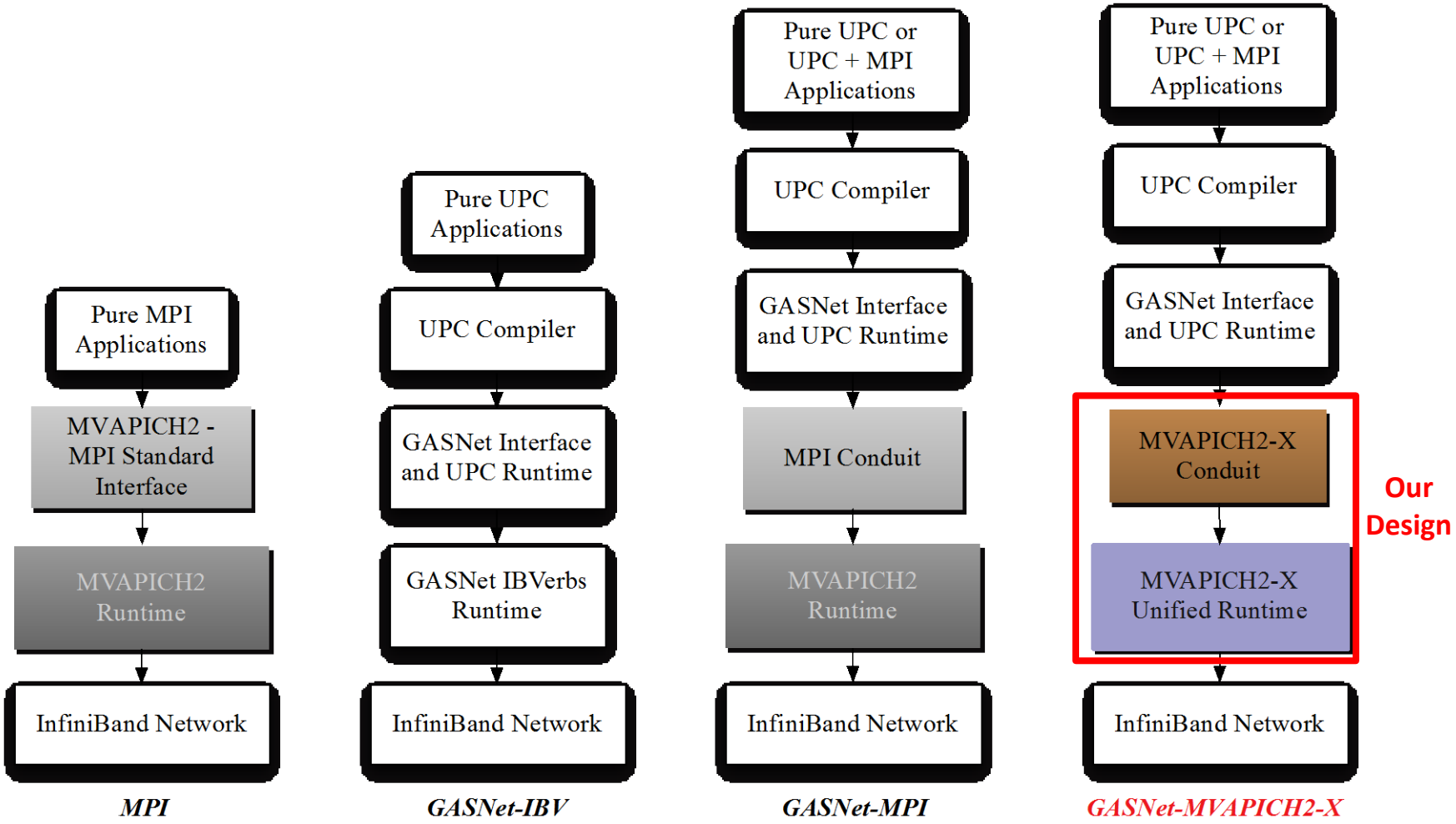
Unified Runtime for Hybrid MPI + OpenSHMEM Applications



- Optimal network resource usage
- No deadlock because of single runtime
- Better performance

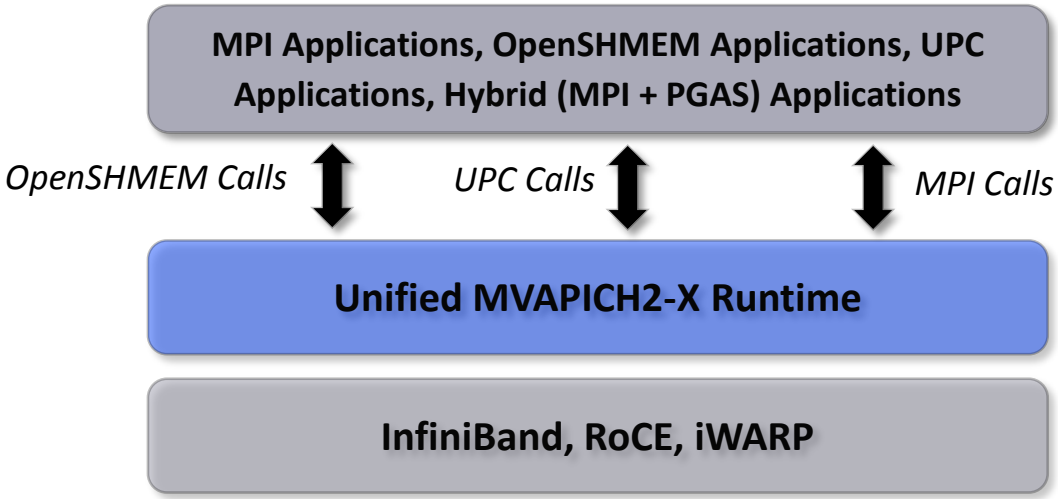
J. Jose, K. Kandalla, M. Luo and D. K. Panda, Supporting Hybrid MPI and OpenSHMEM over InfiniBand: Design and Performance Evaluation, Int'l Conference on Parallel Processing (ICPP '12), September 2012.

Unified Runtime for Hybrid MPI + UPC Applications



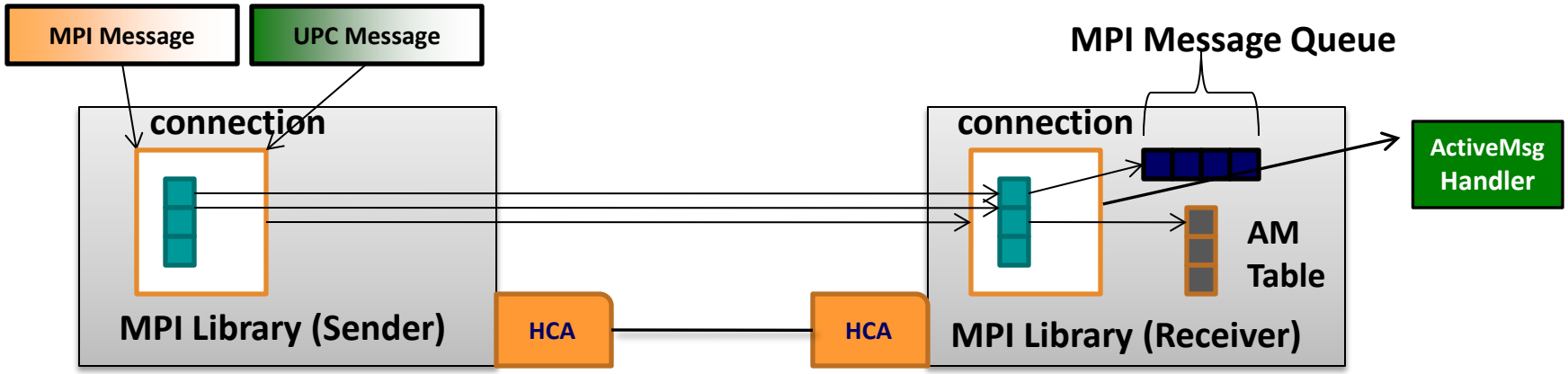
J. Jose, M. Luo, S. Sur and D. K. Panda, Unifying UPC and MPI Runtimes: Experience with MVAPICH, PGAS 2010

MVAPICH2-X for Hybrid MPI + PGAS Applications



- Unified communication runtime for MPI, UPC, OpenSHMEM available with MVAPICH2-X 1.9 onwards!
  - http://mvapich.cse.ohio-state.edu
- Feature Highlights
  - Supports MPI(+OpenMP), OpenSHMEM, UPC, MPI(+OpenMP) + OpenSHMEM, MPI(+OpenMP) + UPC
  - MPI-3 compliant, OpenSHMEM v1.0 standard compliant, UPC v1.2 standard compliant with initial support for UPC v1.3
  - Scalable Inter-node and intra-node communication – point-to-point and collectives

Unified Runtime Implementation



- All resources are shared between MPI and UPC
  - Connections, buffers, memory registrations
  - Schemes for establishing connections (fixed, on-demand)
  - RDMA for large AMs and for PUT, GET

Presentation Overview

- PGAS Programming Models and Runtimes
  - PGAS Languages: Unified Parallel C (UPC)
  - PGAS Libraries: OpenSHMEM
- Hybrid MPI+PGAS Programming Models and Benefits
- High-Performance Runtime for Hybrid MPI+PGAS Models
- Application-level Case Studies and Evaluation

Incremental Approach to exploit one-sided operations

- Identify the communication critical section
- Allocate memory in shared address space
- Convert MPI Send/Recvs to assignment operations or one-sided operations
  - Non-blocking operations can be utilized
  - Coalescing for reducing the network operations
- Introduce synchronization operations for data consistency
  - After Put operations or before get operations
- Load balance through global view of data

Introduction to Graph500

- Graph500 Benchmark
  - Represents data intensive and irregular applications that use graph algorithm-based processing methods
  - Bioinformatics and life sciences, social networking, data mining, and security/intelligence rely on graph algorithmic methods
  - Exhibits highly irregular and dynamic communication pattern
  - Earlier research have indicated scalability limitations of the MPI-based Graph500 implementations

Graph500 Benchmark – The Algorithm

- Breadth First Search (BFS) Traversal
- Uses ‘Level Synchronized BFS Traversal Algorithm
  - Each process maintains – ‘*CurrQueue*’ and ‘*NewQueue*’
  - Vertices in *CurrQueue* are traversed and newly discovered vertices are sent to their owner processes
  - Owner process receives edge information
    - If not visited; updates parent information and adds to *NewQueue*
  - Queues are swapped at end of each level
  - Initially the ‘root’ vertex is added to *currQueue*
  - Terminates when queues are empty

MPI-based Graph500 Benchmark

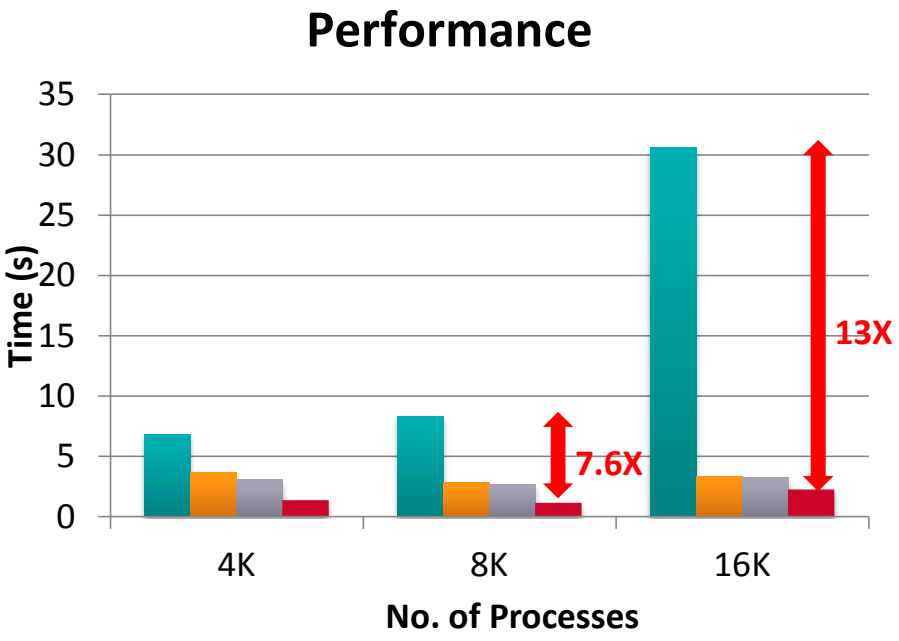
- MPI_Isend/MPI_Test-MPI_Irecv for transferring vertices
- Implicit barrier using zero length message
- MPI_Allreduce to count number of *newqueue* elements
- Major Bottlenecks:
  - Overhead in send-recv communication model
    - More CPU cycles consumed, despite using non-blocking operations
    - Most of the time spent in MPI_Test
  - Implicit Linear Barrier
    - Linear barrier causes significant overheads

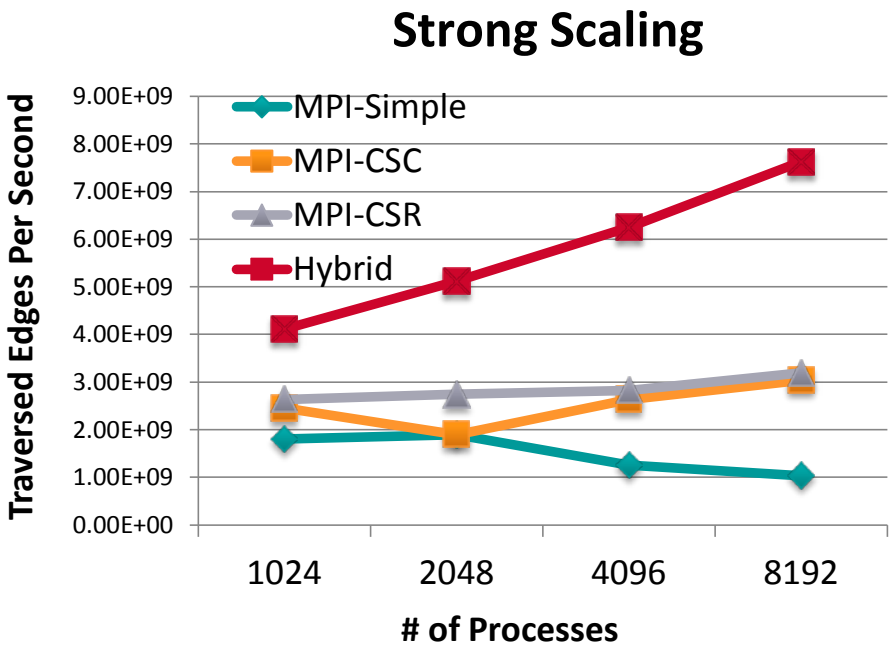
Hybrid Graph500 Design

- Communication and co-ordination using one-sided routines and fetch-add atomic operations
  - Every process keeps receive buffer
  - Synchronization using atomic fetch-add routines
- Level synchronization using non-blocking barrier
  - Enables more computation/communication overlap
- Load Balancing utilizing OpenSHMEM shmem_ptr
  - Adjacent processes can share work by reading shared memory

J. Jose, S. Potluri, K. Tomko and D. K. Panda, Designing Scalable Graph500 Benchmark with Hybrid MPI+OpenSHMEM Programming Models, International Supercomputing Conference (ISC '13), June 2013

Graph500 - BFS Traversal Time



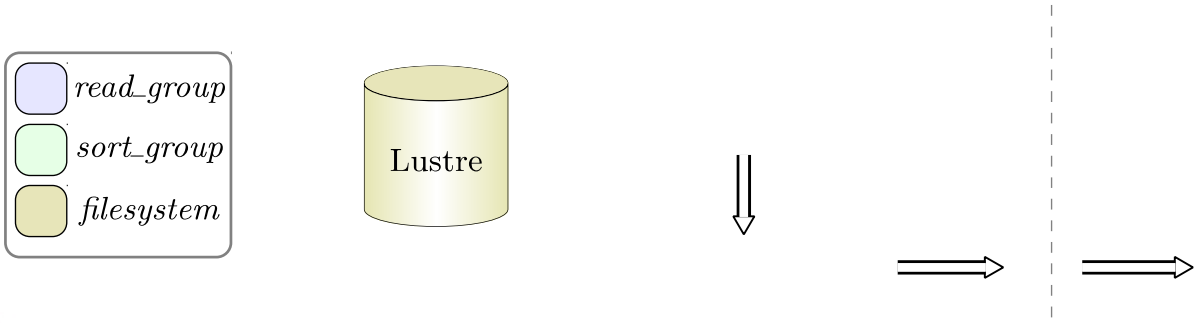


- Hybrid design performs better than MPI implementations
- 16,384 processes
  - **1.5X** improvement over MPI-CSR
  - **13X** improvement over MPI-Simple (Same communication characteristics)
- Strong Scaling

  Graph500 Problem Scale = 29

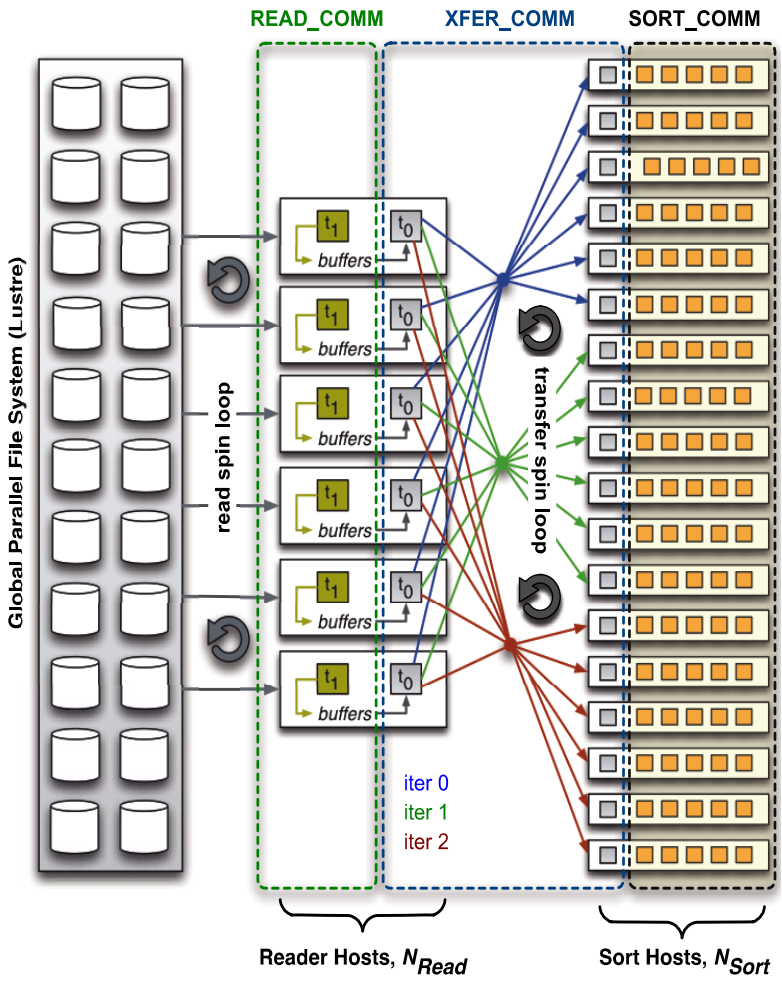
Out-of-Core Sorting

- Sorting: One of the most common algorithms in data analytics
- Sort Benchmark (sortbenchmark.org) ranks various frameworks available for large scale data analytics
- Read data from a global filesystem, sort it and write back to global filesystem



Overview of Existing Design

- Processes grouped into read and sort groups
- Read group processes read data and sends to sort group processes in a ‘streaming’ manner
- Sort processes sample initial data and determines the split
- Input data is sorted and bucketed based on the split
- Merge sort on each split, and final write back to global filesystem



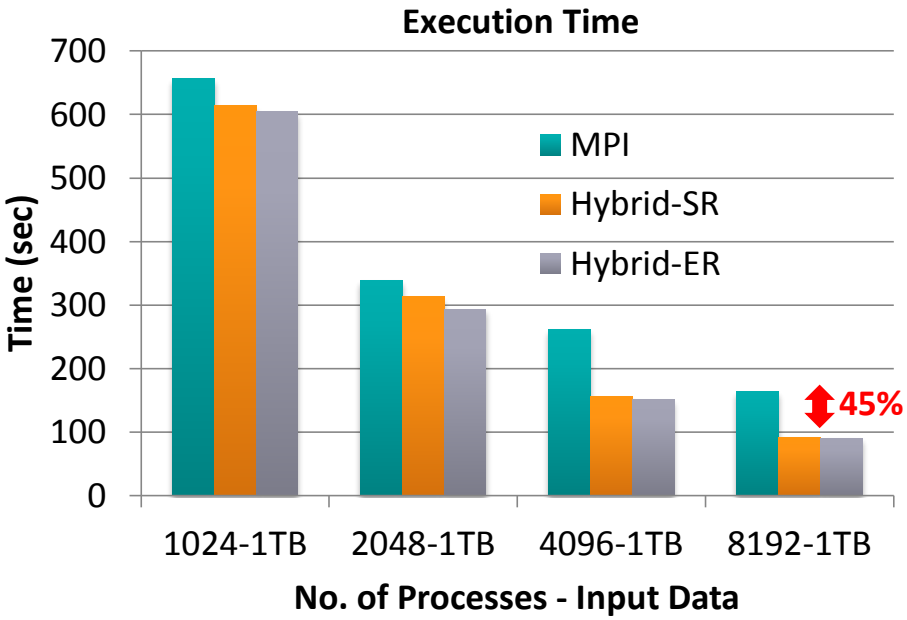
Overheads in Existing Design

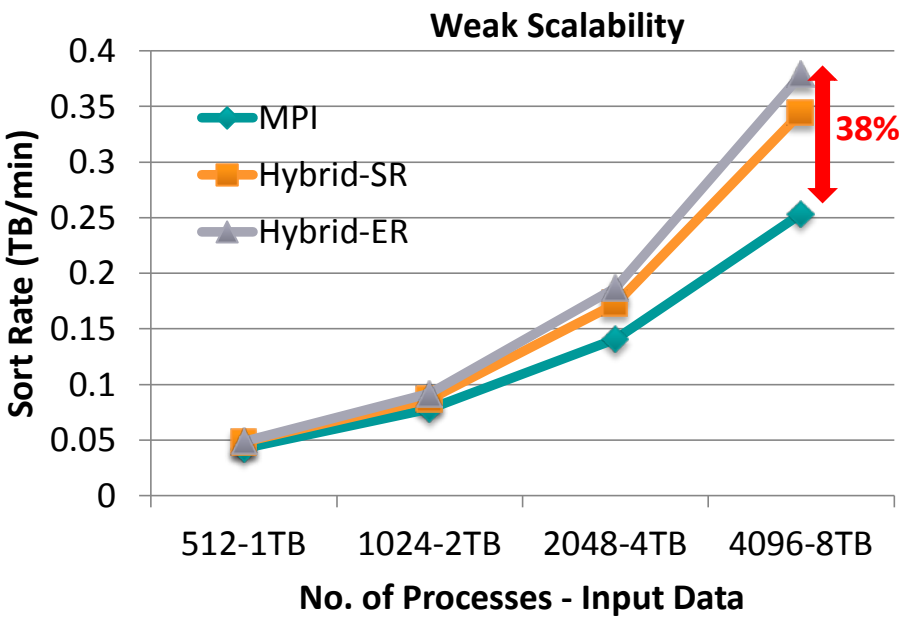
- Poor resource utilization and overlap
  - Dedicated Receiver Task limits the compute resources available for sorting
  - Multiple BIN_COMMs expected to provide high overlap
  - Profiling Data using HPC-Toolkit indicates nearly 30% time spent in waiting for input data
- Book-keeping and Synchronization overheads
  - Reader tasks continuously participate in MPI_Gather/MPI_Scatter destination assignment

Hybrid MPI+OpenSHMEM Out-of-Core Design

- Data Transfer using OpenSHMEM one-sided communication
- Atomic Counter based destination selection
- Remote buffer co-ordination using compare-swap
- Non-blocking put+notify for data delivery and synchronization
- Buffer structure for efficient synchronization
- Custom memory allocator using OpenSHMEM shared heap

Hybrid MPI+OpenSHMEM Sort Application



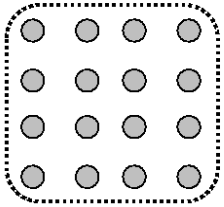


- Performance of Hybrid (MPI+OpenSHMEM) Sort Application
  - Execution Time
    - 1TB Input size at 8,192 cores: MPI – 164 seconds, Hybrid-SR (Simple Read) – 92.5 seconds, Hybrid-ER (Eager Read) - 90.36 seconds
    - **45%** improvement over MPI-based design
  - Weak Scalability **(c**onfiguration: input size of 1TB per 512 cores)
    - **At 4,096 cores:** MPI – 0.25 TB/min, Hybrid-SR – 0.34 TB/min, **Hybrid-SR –0.38 TB/min**
    - **38%** improvement over MPI based design

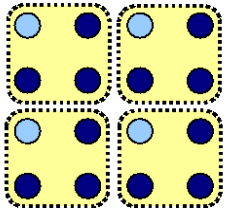
Barnes Hut

- N-body cosmological simulation algorithm
- Simulate motion of n bodies over T time steps
- At each step, calculate the gravitational interaction of each body with all others to find the net force
- Approximate the interaction between distant bodies as an interaction with the center of mass of whole region
- Represents sparse volume of 3-dimensional space using a large shared oct-tree (each node is split in half along each dimension, resulting 8 children)

Barnes Hut – using hybrid MPI+UPC



UPC Only (baseline)



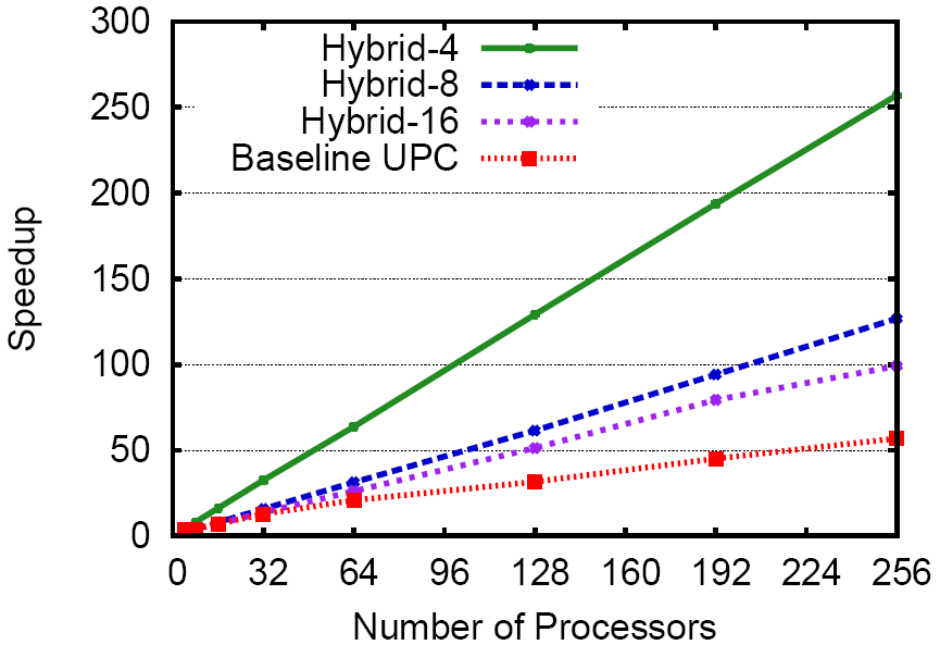
Nested-Funneled Hybrid UPC+MPI

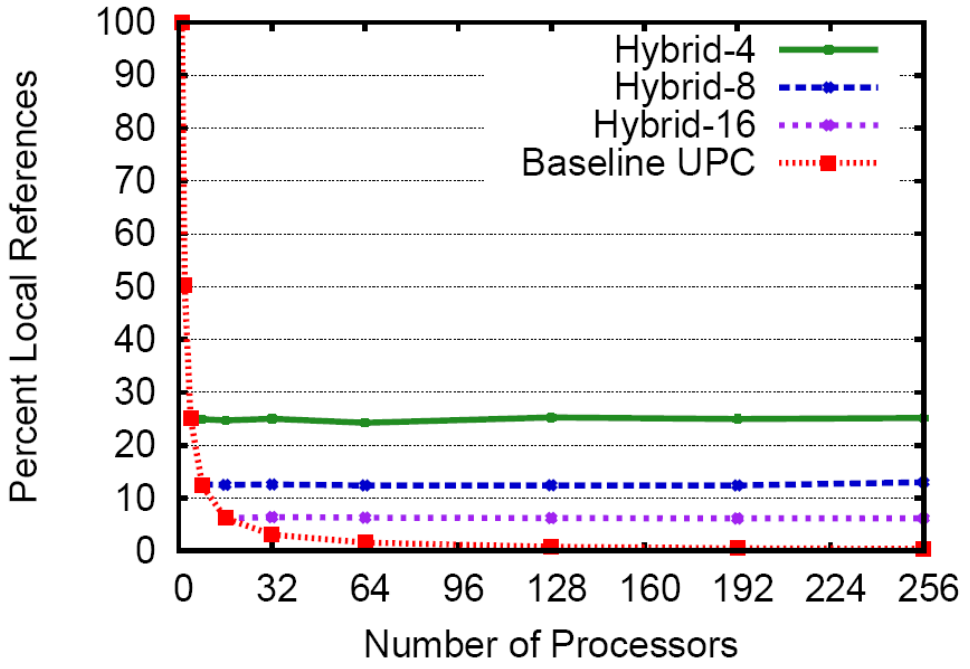
```
for i in 1..t_max
  t <- new octree()
  forall b in bodies
    insert(t, b)
  summarize_subtrees(t)



  forall b in bodies
    compute_forces(b, t)
  forall b in bodies
    advance(b)
```

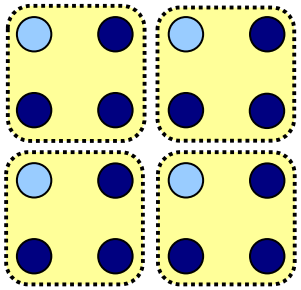
```
for i in 1..t_max
  t <- new octree()
  forall b in bodies
    insert(t, b)
  summarize_subtrees(t)
  our_bodies <-
    partion(group id, bodies)
  forall b in our_bodies
    compute_forces(b, t)
  forall b in our_bodies
    advance(b)
  Allgather(our_bodies, bodies)
```

- Nested-Funneled: One MPI rank per group, UPC threads for communicating within the group, and MPI for communicating across groups

Hybrid MPI+UPC Barnes-Hut







Hybrid-4

- Nested-funneled model
  - Tree is replicated across UPC groups
- 51 new lines of code (2% increase)
  - Distribute work and collect results
- 55X speedup at 256 processes

J. Dinan, P. Balaji, E. Lusk, P. Sadayappan and R. Thakur, Hybrid Parallel Programming with MPI and Unified Parallel C, ACM Computing Frontiers, 2010

Hands-on Exercises

PDF version of slides available from

http://www.cse.ohio-state.edu/~panda/vssce14/dk_karen_day2_exercises.pdf

Compiling Programs with MVAPICH2-X

- Compile MPI programs using mpicc
  - $ mpicc -o helloworld_mpi helloworld_mpi.c
- Compile UPC programs using upcc
  - $ upcc -o helloworld_upc helloworld_upc.c
- Compile OpenSHMEM programs using oshcc
  - $ oshcc -o helloworld_oshm helloworld_oshm.c
- Compile Hybrid MPI+UPC programs using upcc
  - $ upcc -o hybrid_mpi_upc hybrid_mpi_upc.c
- Compile Hybrid MPI+OpenSHMEM programs using oshcc
  - $ oshcc -o hybrid_mpi_oshm hybrid_mpi_oshm.c

Running Programs with MVAPICH2-X

- MVAPICH2-X programs can be run using
  - mpirun_rsh and mpiexec.hydra (MPI, UPC, OpenSHMEM and hybrid)
  - upcrun (UPC)
  - oshrun (OpenSHMEM)
- Running using mpirun_rsh/mpiexec.hydra
  - $ mpirun rsh -np 4 -hostfile hosts ./test
  - $ mpiexec -f hosts -n 2 ./test
- Running using upcrun
  - $ export MPIRUN CMD="<path-to-MVAPICH2-X-install>/bin/mpirun rsh -np %N -hostfile hosts %P %A"
  - $ upcrun -n 2 ./test
- Running using oshrun
  - $ oshrun -f hosts -np 2 ./test

OpenSHMEM

1. **Circular Shift - OpenSHMEM**: Write a simple program using OpenSHMEM, with eight processes. Every process shall have two ints (a1, a2). Do a circular left-shift on a1's globally to get the output on a2. An example for circular shift implemented using static int is given in /nfs/02/w557091/mpi-pgas-exercises/openshmem/excercise1-ref.c. Re-write this code to use dynamic memory allocation

2. **PI Calculation**: Write a program with multiple processes for calculating PI. Process0 shall read the number of iterations from stdin, and broadcast to all processes. The PI calculation work is shared among all the PEs, where each PE calculates locally, and then summed up to generate the global solution (Reference MPI code: http://www.mcs.anl.gov/research/projects/mpi/usingmpi/examples/simplempi/cpi_c.htm)

UPC

- **Circular Shift - UPC**: Write the circular-shift program described in previous slide using UPC. Here, the arrays shall be declared as UPC shared array, and the set operations shall be done using assignment operations
- **Work Sharing:** Write the PI calculation code explained in previous slide in UPC, by utilizing the ‘upc_forall’ work-sharing construct

Hybrid MPI+PGAS

- **Simple Hybrid MPI+OpenSHMEM Code**: Write a hybrid MPI+OpenSHMEM code, in which every process does an OpenSHMEM fetch-and-add of the rank to a variable (sum) at process 0. Following the fetch-and-add, every process shall do a barrier using MPI_Barrier, and then Process0 broadcasts sum to all processes using MPI_Bcast. Finally, all processes print the variable sum
- **Simple Hybrid MPI+UPC Code**: Write the similar code using hybrid MPI+UPC. Here, every UPC thread set its rank to one element of a global shared memory array (A) such that A[MYTHREAD] has affinity with the UPC thread who set the value of it. Then an MPI Allreduce operation is made to sum-up the ranks, and then every UPC thread does an MPI barrier, and all processes print the sum

Solutions and Guidelines

- Solutions for these exercises available at:
  /nfs/02/w557091/mpi-pgas-exercises/

- See README file inside the above folder for Build and Run instructions

Plans for Thursday

- Tuesday, May 6 – MPI-3 Additions to the MPI Spec
  - Updates to the MPI One-Sided Communication Model (RMA)
  - Non-Blocking Collectives
  - MPI Tools Interface
- Wednesday, May 7 – MPI/PGAS Hybrid Programming
  - MVAPICH2-X: Unified runtime for MPI+PGAS
  - MPI+OpenSHMEM
  - MPI+UPC
- Thursday, May 8 – MPI for many-core processor
  - MVAPICH2-GPU: CUDA-aware MPI for NVidia GPU
  - MVAPICH2-MIC Design for Clusters with InfiniBand and Intel Xeon Phi