PGAS and Hybrid MPI+PGAS Programming Models on Modern HPC Clusters with Accelerators
Parallel Programming Models Overview

- 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.
- Additionally, OpenMP can be used to parallelize computation within the node
- Each model has strengths and drawbacks - suite different problems or applications
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
    • UPC++
    • 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
Presentation Overview

• Introduction to PGAS Programming Models
• Runtime Designs for PGAS and Hybrid MPI+PGAS models
• Application Level Case Studies and Evaluation
• High-Performance OpenSHMEM Runtimes for GPU Clusters
• Conclusions and Final Q&A
Can High-Performance Interconnects, Protocols and Accelerators Benefit MPI, PGAS and Hybrid MPI+PGAS Models?

- Can MPI, PGAS and Hybrid MPI+PGAS models take advantage of high-performance interconnects, protocols and accelerators?
- What are the challenges?
- Where do the bottlenecks lie?
- Can these bottlenecks be alleviated with new designs?
- What are the impacts of such designs on performance?
Presentation Overview

• **Introduction to PGAS Programming Models**
  – **PGAS Libraries: OpenSHMEM**
  – **PGAS Languages: Unified Parallel C (UPC)**
  – **Hybrid MPI+PGAS Programming Models and Benefits**

• **Runtime Designs for PGAS and Hybrid MPI+PGAS models**

• **Application Level Case Studies and Evaluation**

• **High-Performance OpenSHMEM Runtimes for GPU Clusters**

• **Conclusions and Final Q&A**
OpenSHMEM

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

<table>
<thead>
<tr>
<th></th>
<th>SGI SHMEM</th>
<th>Quadrics SHMEM</th>
<th>Cray SHMEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>start_pes(0)</td>
<td>shmem_init</td>
<td>start_pes</td>
</tr>
<tr>
<td>Process ID</td>
<td>_my_pe</td>
<td>my_pe</td>
<td>shmem_my_pe</td>
</tr>
</tbody>
</table>

- Made application 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
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: 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_putSIZE` – elements of SIZE: 32/64/128
  - `shmem_putmem` - bytes
  - Similar get operations

```c
int *b;
b = (int *) shmalloc (10*sizeof(int));
if ((_my_pe() == 0) {
    shmem_int_put (b, b, 5, 1);
}
```
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
A Sample code: Circular Shift

```c
#include <shmem.h>
int aaa, bbb;
int main (int argc, char *argv[]) {
    int target_pe;
    start_pes(0);
    target_pe = (_my_pe() + 1)% _num_pes();
    bbb = _my_pe() + 1
    shmem_barrier_all();
    shmem_int_get (&aaa, &bbb, 1, target_pe);
    shmem_barrier_all();
}
```
Presentation Overview

• Introduction to PGAS Programming Models
  – PGAS Libraries: OpenSHMEM
  – **PGAS Languages: Unified Parallel C (UPC)**
    – Hybrid MPI+PGAS Programming Models and Benefits

• Runtime Designs for MPI, PGAS and Hybrid MPI+PGAS models

• Application Level Case Studies and Evaluation

• High-Performance MPI and OpenSHMEM Runtimes for GPU Clusters

• Conclusions and Final Q&A
UPC: Execution Model

• A UPC program is translated under a static or dynamic THREADS environment:
  – THREADS: number of threads working independently in 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:
  
```c
#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: holds 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]
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

Right is the pointer type and left is the space type.
UPC: Dynamic Memory Management

- shared void *upc_all_alloc (size_t nbblocks, 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 nbblocks, 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 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.
A Sample code: Circular Shift in UPC

```c
#include <upc.h>

shared [1] int A[THREADS];
shared [1] int B[THREADS];

int main (int argc, char * argv[])
{
    A[MYTHREAD]=MYTHREAD+1;
    upc_barrier;
    B[MYTHREAD] = A[(MYTHREAD+1)%THREADS];
    upc_barrier;
}
```
Presentation Overview

• Introduction to PGAS Programming Models
  – PGAS Libraries: OpenSHMEM
  – PGAS Languages: Unified Parallel C (UPC)
  – Hybrid MPI+PGAS Programming Models and Benefits

• Runtime Designs for PGAS and Hybrid MPI+PGAS models
• Application Level Case Studies and Evaluation
• High-Performance OpenSHMEM Runtimes for GPU Clusters
• Conclusions and Final Q&A
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

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
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
Template for simple Hybrid MPI + OpenSHMEM examples:

```c
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

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

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

Hybrid MPI+OpenSHMEM version

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

- Introduction to PGAS Programming Models
- Runtime Designs for PGAS and Hybrid MPI+PGAS models
- Application Level Case Studies and Evaluation
- High-Performance OpenSHMEM Runtimes for GPU Clusters
- Conclusions and Final Q&A
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
OpenSHMEM Data Movement in MVAPICH2-X

• Data Transfer Routines (put/get)
  – Implemented using RDMA transfers
  – Strided operations require multiple RDMA transfers
  – IB requires remote registration information for RDMA - expensive

• Remote Registration Cache
  – Registration request sent over “Active Message”
  – Remote process registers and responds with the key
  – Key is cached at local and remote sides
  – Hides registration costs
OpenSHMEM Data Movement: Performance

- **shmem_putmem**
  - UH-SHMEM
  - MV2X-SHMEM

- **shmem_getmem**
  - UH-SHMEM
  - MV2X-SHMEM

- OSU OpenSHMEM micro-benchmarks
  - [http://mvapich.cse.ohio-state.edu/benchmarks/](http://mvapich.cse.ohio-state.edu/benchmarks/)

- Slightly better performance for putmem and getmem with MVAPICH2-X

- MVAPICH2-X 2.2 RC1, Broadway CPU, EDR Interconnect
OpenSHMEM Atomic Operations: Performance

- OSU OpenSHMEM micro-benchmarks (OMB v5.3)
- MV2-X SHMEM performs up to 22% better compared to UH-SHMEM
OpenSHMEM 1.3 Support: Data Movement

- Highly optimized for IB networks
- Higher message rate with NBI interface
- Perfect overlap of computation/Communication
- Extension of OMB with OpenSHMEM benchmarks
- Will be available with next releases of MV2-X

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
J. Jose, K. Hamidouche, J. Zhang, A. Venkatesh, and D. K. Panda, Optimizing Collective Communication in UPC (HiPS’14, in association with IPDPS’14)
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 dedicated progress threads
  - Bad performance!!
- Similar issues for MPI + UPC applications over individual runtimes

```c
shmem_int_fadd (data at p1);
/* operate on data */
MPI_Barrier(comm);
```

```c
/*
local
computation
*/
MPI_Barrier(comm);
```
• Unified communication runtime for MPI, UPC, OpenSHMEM, CAF, UPC++ available with MVAPICH2-X 1.9 onwards! (since 2012)
  – http://mvapich.cse.ohio-state.edu
• Feature Highlights
  – Supports MPI(+OpenMP), OpenSHMEM, UPC, CAF, 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 1.3), CAF 2008 standard (OpenUH), UPC++
  – Scalable Inter-node and intra-node communication – point-to-point and collectives
Hybrid MPI+UPC NAS-FT

- Modified NAS FT UPC all-to-all pattern using MPI_Alltoall
- Truly hybrid program
- For FT (Class C, 128 processes)
  - **34%** improvement over UPC-GASNet
  - **30%** improvement over UPC-OSU

Hybrid MPI + UPC Support
Available since
MVAPICH2-X 1.9 (2012)

J. Jose, M. Luo, S. Sur and D. K. Panda, Unifying UPC and MPI Runtimes: Experience with MVAPICH, Fourth Conference on Partitioned Global Address Space Programming Model (PGAS ’10), October 2010
Performance Evaluations for CAF model

- Micro-benchmark improvement (MV2X vs. GASNet-IBV, UH CAF test-suite)
  - Put bandwidth: 3.5X improvement on 4KB; Put latency: reduce 29% on 4B
- Application performance improvement (NAS-CAF one-sided implementation)
  - Reduce the execution time by 12% (SP.D.256), 18% (BT.D.256)

**UPC++ Collectives Performance**

- Full and native support for hybrid MPI + UPC++ applications
- Better performance compared to IBV and MPI conduits
- OSU Micro-benchmarks (OMB) support for UPC++
- Available since MVAPICH2-X 2.2RC1

**Inter-node Broadcast (64 nodes 1:ppn)**

![Graph showing time vs message size for GASNet_MPI, GASNET_IBV, and MV2-X](image)

Presentation Overview

• Introduction to PGAS Programming Models
• Runtime Designs for PGAS and Hybrid MPI+PGAS models
• **Application Level Case Studies and Evaluation**
  – Graph500
  – Out-of-Core Sort
• High-Performance OpenSHMEM Runtimes for GPU Clusters
• Conclusions and Final Q&A
Incremental Approach to Exploit One-sided Operations

- Identify the communication critical section (mpiP, HPCToolkit)
- 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
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 *newqueue* elements
- Major Bottlenecks:
  - Overhead in send-receive 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
Pseudo Code For Both MPI and Hybrid Versions

**Algorithm 1: EXISTING MPI SEND/RECV**

```plaintext
while true do
    while CurrQueue != NULL do
        for vertex u in CurrQueue do
            HandleReceive()
            u ← Dequeue(CurrQueue)
            Send(u, v) to owner
        end
    Send empty messages to all others
    while all_done != N − 1 do
        HandleReceive()
    end
    // Procedure: HandleReceive
    if rcv_count = 0 then
        all_done ← all_done + 1
    else
        update (NewQueue, v)
    end
end
```

**Algorithm 2: HYBRID VERSION**

```plaintext
while true do
    while CurrQueue 6= NULL do
        for vertex u in CurrQueue do
            u ← Dequeue(CurrQueue)
            to the adjacent points to u do
            Shmem_fadd(owner, size,recv_index)
            shmem_put(owner, size,recv_buf)
        end
    end
    if recv_buf[size] = done then
        Set ← 1
    end
end
```
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

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
Presentation Overview

• Introduction to PGAS Programming Models

• Runtime Designs for PGAS and Hybrid MPI+PGAS models

• **Application Level Case Studies and Evaluation**
  
  – Graph500
  
  – Out-of-Core Sort

• High-Performance OpenSHMEM Runtimes for GPU Clusters

• Conclusions and Final Q&A
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
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
Remote Buffer Co-ordination and Data-delivery

- Sender does compare-swap on the status buffer to know the status of remote buffer chunk.
- If buffer is ‘empty’ (0), data is transferred using non-blocking put+notify.
- Notify writes ‘full’ (1) on remote status buffer.
- If buffer was full, cswap operation writes rank of sender at remote status buffer.
- Sender polls local buffer until signaled by remote process.

![Diagram of remote buffer coordination and data-delivery](image)
Hybrid MPI+OpenSHMEM Sort Application

**Execution Time**

<table>
<thead>
<tr>
<th>No. of Processes - Input Data</th>
<th>MPI</th>
<th>Hybrid-SR</th>
<th>Hybrid-ER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024-1TB</td>
<td>200</td>
<td>450</td>
<td>700</td>
</tr>
<tr>
<td>2048-1TB</td>
<td>400</td>
<td>800</td>
<td>1400</td>
</tr>
<tr>
<td>4096-1TB</td>
<td>600</td>
<td>1200</td>
<td>2100</td>
</tr>
<tr>
<td>8192-1TB</td>
<td>800</td>
<td>1600</td>
<td>2800</td>
</tr>
</tbody>
</table>

**Weak Scalability**

- **Performance of Hybrid (MPI+OpenSHMEM) Sort Application**
  - **Execution Time (seconds)**
    - 1TB Input size at 8,192 cores: MPI – 164, Hybrid-SR (Simple Read) – 92.5, Hybrid-ER (Eager Read) - 90.36
    - **45%** improvement over MPI-based design
  - **Weak Scalability** (configuration: 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

J. Jose, S. Potluri, H. Subramoni, X. Lu, K. Hamidouche, K. Schulz, H. Sundar and D. Panda Designing Scalable Out-of-core Sorting with Hybrid MPI+PGAS Programming Models, PGAS’14
Presentation Overview

- Trends in Modern HPC Clusters and Interconnects
- Introduction to MPI and PGAS Programming Models
- Runtime Designs for MPI, PGAS and Hybrid MPI+PGAS models
- Application Level Case Studies and Evaluation
- **High-Performance OpenSHMEM Runtimes for GPU Clusters**
- Conclusions and Final Q&A
Limitations of OpenSHMEM for GPU Computing

- OpenSHMEM memory model does not support disjoint memory address spaces - case with GPU clusters

Existing OpenSHMEM Model with CUDA

PE 0

```
host_buf = shmalloc (...)  
cudaMemcpy (host_buf, dev_buf, ...)  
shmem_putmem (host_buf, host_buf, size, pe)  
shmem_barrier (...)  
```

PE 1

```
shmem_barrier (...)  
cudaMemcpy (dev_buf, host_buf, size, ...)  
host_buf = shmalloc (...)  
```

- Copies severely limit the performance
- Synchronization negates the benefits of one-sided communication
- Similar issues with UPC
Global Address Space with Host and Device Memory

- Extended APIs:
  - heap_on_device/heap_on_host
  - a way to indicate location of heap
  - host_buf = shmalloc (sizeof(int), 0);
  - dev_buf = shmalloc (sizeof(int), 1);

CUDA-Aware OpenSHMEM
Same design for UPC
PE 0
dev_buf = shmalloc (size, 1);
shmem_putmem (dev_buf, dev_buf, size, pe)

PE 1
dev_buf = shmalloc (size, 1);

CUDA-aware OpenSHMEM and UPC runtimes

• After device memory becomes part of the global shared space:
  - Accessible through standard UPC/OpenSHMEM communication APIs
  - Data movement transparently handled by the runtime
  - Preserves one-sided semantics at the application level

• Efficient designs to handle communication
  - Inter-node transfers use host-staged transfers with pipelining
  - Intra-node transfers use CUDA IPC
  - Possibility to take advantage of GPUDirect RDMA (GDR)

• Goal: Enabling High performance one-sided communications semantics with GPU devices
Inter-Node Communication

• Pipelined data transfers through host memory - overlap between CUDA copies and IB transfers
• Done transparently by the runtime
• Designs with GPUDirect RDMA can help considerably improve performance
• Hybrid design:
  – GPUDirect RDMA
  – Pipeline 1 Hop
  – Proxy-based design
Exploiting GDR: OpenSHMEM: Inter-node Evaluation

- GDR for small/medium message sizes
- Host-staging for large message to avoid PCIe bottlenecks
- Hybrid design brings best of both
- 3.13 us Put latency for 4B (7X improvement) and 4.7 us latency for 4KB
- 9X improvement for Get latency of 4B
GDR for small and medium message sizes
IPC for large message to avoid PCIe bottlenecks
Hybrid design brings best of both

2.42 us Put D-H latency for 4 Bytes (2.6X improvement) and 3.92 us latency for 4 KBytes
3.6X improvement for Get operation
Similar results with other configurations (D-D, H-D and D-H)
Application Evaluation: GPULBM and 2DStencil

**Weak Scaling**

<table>
<thead>
<tr>
<th>Number of GPU Nodes</th>
<th>Host-Pipeline</th>
<th>Enhanced-GDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>100</td>
<td>45%</td>
</tr>
<tr>
<td>16</td>
<td>100</td>
<td>45%</td>
</tr>
<tr>
<td>32</td>
<td>100</td>
<td>45%</td>
</tr>
<tr>
<td>64</td>
<td>100</td>
<td>45%</td>
</tr>
</tbody>
</table>

GPULBM: 64x64x64

- Redesign the application
  - CUDA-Aware MPI: `Send/Recv` => hybrid CUDA-Aware MPI+OpenSHMEM
  - `cudaMalloc` => `shmalloc(size,1)`
  - `MPI_Send/recv` => `shmem_put + fence`
  - 53% and 45%
  - Degradation is due to small Input size

2DStencil 2Kx2K

- Platform: **Wilkes** (Intel Ivy Bridge + NVIDIA Tesla K20c + Mellanox Connect-IB)
- New designs achieve 20% and 19% improvements on 32 and 64 GPU nodes

Presentation Overview

- Introduction to PGAS Programming Models
- Runtime Designs for PGAS and Hybrid MPI+PGAS models
- Application Level Case Studies and Evaluation
- High-Performance OpenSHMEM Runtimes for GPU Clusters
- Conclusions and Final Q&A
Concluding Remarks

- Presented an overview of PGAS models and Hybrid MPI+PGAS models
- Outlined research challenges in designing efficient runtimes for these models on clusters with InfiniBand and GPGPUs
- Demonstrated the benefits of Hybrid MPI+PGAS models for a set of applications
- Hybrid MPI+PGAS model is an emerging paradigm which can lead to high-performance and scalable implementation of applications on exascale computing systems