Lessons Learned from OSHMPI

Yanfei Guo

Argonne National Laboratory

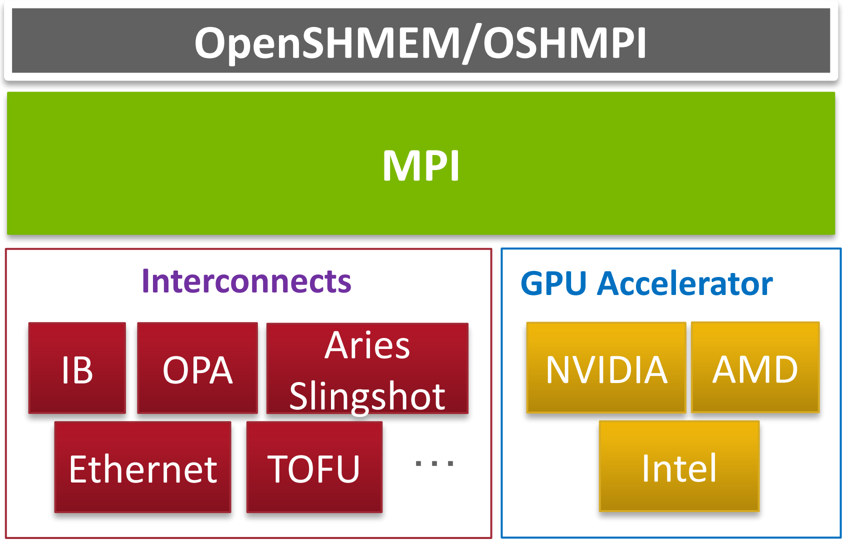
yguo@anl.gov

Background of MPI and OpenSHMEM

- MPI and OpenSHMEM are both programming models for large-scale distributed memory systems (a.k.a most supercomputers today)
- OpenSHMEM employs a Partitioned Global Address Space (PGAS) abstract which provides a "global view of shared memory" among compute processes/nodes/blades. It allows user to use PUT/GET operation (similar to memory read/write semantic) to access this globally shared memory. Under the hood, these accesses will be performed through communication between compute nodes.
- MPI is traditionally focus on high-performance communication between processes/nodes, and has been well optimized to run on different architectures and hardware (e.g. CPUs and GPUs)

Overview of OpenSHMEM over MPI

- OSHMPI: an OpenSHMEM library built on top of MPI
  - A ***portable*** OpenSHMEM implementation
    - If there is MPI on the machine, the user can run OpenSHMEM application and expect good performance
  - A ***performant*** OpenSHMEM implementation
    - Leveraging the high-performance MPI library
  - A ***GPU-aware*** OpenSHMEM implementation
    - *Support CPU-initiated GPU communication*
    - *Leverage highly-optimized GPU-aware MPI implementations*



Major Technical Challenges of the Project

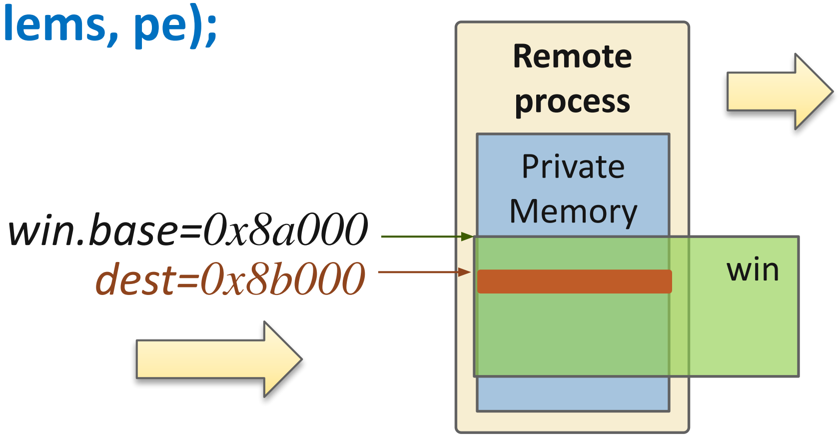
- OpenSHMEM has always focused on high levels of performance
  - Any overhead is too much overhead
  - Encouraged "native" down-to-the-metal implementations
  - Intent is to get to close to zero instructions from the application to the network hardware
- Why does OpenSHMEM over MPI not perform well?
  - Challenge 1: MPI implementations have not been traditionally optimized for PUT/GET operations
    - Many MPI implementations focus their effort on optimizing SEND/RECV and Collective operations (e.g. reduce, broadcast, etc.)
  - Challenge 2: MPI standard is too generic compared with OpenSHMEM
    - E.g., MPI allows PUT/GET communication with weird unstructured data structures
      - Even if OpenSHMEM/MPI does not need this functionality, MPI implementations still have to check

SHEMEM RMA and MPI RMA Semantics

- OpenSHMEM
  - Symmetric heap or global variable are remote accessible
  - Use absolute virtual address of remote buffer
  - Define separate function for each data type (e.g., shmem_int_put)
  - Support only basic datatype
- MPI
  - Expose a remote accessible memory region as ***window***
  - *A window object is associated with a* ***communicator*** (group of processes)
  - Use ***displacement*** of remote buffer
  - Use generic MPI ***datatype*** object
  - Support both basic and user-defined datatypes

shmem_int_put_nbi(dest=0x8b00, source, nelems, pe);

MPI_PUT(source, src_count=nelems,
src_dtype=MPI_INT,
pe, dest_disp=0x1000, **dest_count**=nelems,
dest_type=MPI_INT, **win);**



MPI Internal:

- Translate pe in win->comm to network addr;
- Decode src_dtype and dest_dtype;
- Translate dest_disp to absolute virtual addr;

...

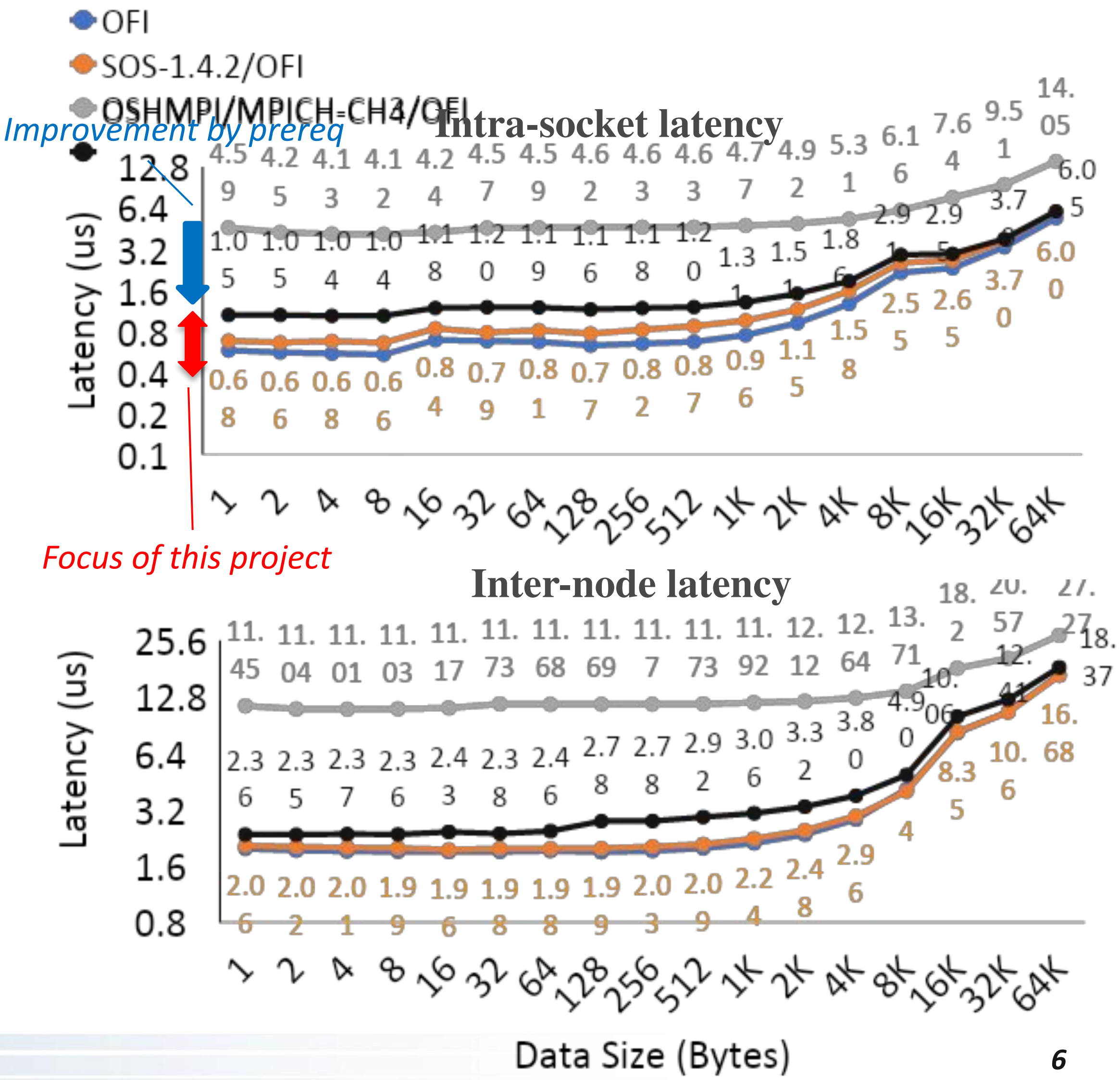
SHMEM PUT latency Evaluation

- Measured latency between two processes

```
shmem_putmem(dest, source, nelems, pe);
shmem_quiet();
```

- Targeted implementations:
  - **OFI**: ideal performance
  - **SOS**: native OpenSHMEM implementation
  - **OSHMPI/MPICH-CH3/OFI**: original implementation before this project
  - **OSHMPI/MPICH-CH4/OFI**: with optimized MPI RMA

shmem_putmem + quiet latency on Argonne Bebop (Intel Broadwell, Omni-Path)



Breakdown of SHMEM_PUTMEM Implementation

OSHMPI
MPI
OFI

shmem_putmem(dest, source, nelems, pe);

- Obtain dest's **win** and **disp** (relative displacement);
- **MPI_Put(source, nelems, src_dtype=MPI_BYTE,**
  pe, disp, nelems, dest_dtype=MPI_BYTE, win);

```
Obtain window object (win_ptr);
Translate pe in win_ptr->comm to network addr(av);
if(!is_local(av)) {
   Decode src_dtype and dest_dtype (i.e., obtain size, is_contig);
    if ( is_contig ) {
       Obtain OFI parameters;
       Issue OFI write message;
    }
}
```

- **MPI_Win_flush_local(pe, win); /* ensure local completion */**

```
Obtain window object (win_ptr);
Wait OFI message completion;
Make full MPI progress (e.g., point-to-point, coll,
    internal active msg, network, shared memory);
```

SOS
OFI

shmem_putmem(...);

- Obtain OFI parameters;
  Issue OFI write message;
- **Wait OFI message local Completion (skipped for small messages);**

Breakdown of SHMEM_QUIET Implementation

OSHMPI | **MPI** | OFI

shmem_quiet();

```
/* ensure remote completion */
MPI_Win_flush_all(win=symm_heap_win);
  Obtain window object (win_ptr);
    Wait OFI message completion;
    Make full MPI progress (e.g., point-to-point, coll,
        internal active msg);
MPI_Win_flush_all(win=symm_data_win);
    Obtain window object (win_ptr);
    Wait OFI message completion;
    Make full MPI progress;
/* ensure memory updates */
MPI_Win_sync(win=symm_heap_win);
    Obtain window object (win_ptr);
    memory barrier;
MPI_Win_sync(win=symm_data_win);
    Obtain window object (win_ptr);
    memory barrier;
```

SOS | OFI

shmem_quiet();

```
└─ Wait OFI message completion;
```

Simple Optimizations

- Datatype decoding
  - Datatype is a constant in each SHMEM op but becomes a variable when passing down to MPI
    - Compiler cannot optimize, result in **14 additional instructions** at PUT fast path
  - **Optimization**: leverage **compiler IPO** (already provided by mainstream compilers) to optimize code across OSHMPI and MPI libraries at link-time
    - All instructions can be eliminated by compiler
  - **Optimization**: If both datatypes are the same, only decode once in MPICH. (OpenSHMEM have the same src dest datatype)
  - **Optimization**: Typed MPI_PUT to eliminate dtype decoding. i.e. MPIX_Put_int (Also possible with link-time optimization/inlining)
- Window metadata access
  - MPI internal win obj stores metadata, e.g., **comm (MPI-specific)**, network ep, remote mr_rkey...
    - Access to MPI **win->comm's attributes** causes expensive pointer dereferences at RMA /AMO fast-path
  - **Optimization**: Identify win with COMM_WORLD at win creation and avoid win->comm dereferences at OSHMPI RMA fast path (All OSHMPI windows use dup of COMM_WORLD)

RMA Optimizations - Avoid Virtual Address Translation (1)

- Unlike OpenSHMEM, MPI defines generic relative offset (displacement) to describe remote address in order to support various network
  - One requires relative offset such as OFI/psm2
  - But another may require absolute vaddr such as OFI/uGNI and UCX
- Causes extra virtual address translation in PUT/GET fast path
- Can we get rid of the extra address translation?

Dest virtual address translation in OSHMPI/MPICH path

```
shmem_putmem(dest=0x8000, source, nelems, pe);

shmem_putmem @ OSHMPI:
dest_disp = dest – symm_heap.base; /* abs vaddr to offset */
MPI_PUT(source, nelems,  src_dtype,
            pe, dest_disp, nelems,  dest_type, win);

MPI_PUT @ MPI:
/* offset to abs vaddr */
[OFI/uGNI, UCX] dest_abs_addr = dest_disp * win->disp_unit + win.winfo[pe].base
[OFI/psm2]      dest_abs_addr = dest_disp * win->disp_unit
```

RMA Optimizations - Avoid Virtual Address Translation (2)

- **Optimization:** MPIX_PUT|GET_ABS (extension of MPI standard):
  - Allows the user to directly specify absolute virtual address as dest
  - *Pros*: Directly benefits networks that require physical remote address (e.g., OFI/uGNI, UCX)
  - *Cons*: For networks that prefer relative offset (e.g., OFI/psm2), extra translation is required
    - Can be avoided by passing an info hint at window creation that only XXX_ABS will be used, thus the window base can be registered as MPI_BOTTOM
    - Instruction "relative offset = dest - MPI_BOTTOM" can be eliminated by compiler, eventually becomes the same as "relative offset = dest"

Using PUT_ABS in OSHMPI/MPICH path

shmem_putmem(dest=0x8000, source, nelems, pe);

shmem_putmem @ OSHMPI:
MPIX_PUT_ABS(source, nelems, **src_dtype**,
pe, dest, nelems, **dest_type**, win);

MPI_PUT @ MPI:
[OFI/uGNI, UCX] dest_abs_addr = dest

RMA Optimizations - Eliminate MPI Full-Progression (1)

- MPI makes expensive "full progress"
  - To ensure prompt progress for all MPI communication types (i.e., point-to-point, collectives, active-message based routines).
  - Expensive network progress polling functions. E.g., fi_cq_read, ucp_worker_progress
- We may not need the "full progress" in the OSHMPI/MPI context?

Progression in shmem_putmen	#Instr
Flush_local: wait network completion • Ensure remote completion of outstanding PUT • OFI: waiting on OFI counters by calling **fi_cntr_read** • UCX: **ucp_ep_flush**	38
Flush_local: MPI full progress • Ensure MPI-layer progression for send/receive, collectives, active message based RMA • OFI: polling incoming CQE by calling **fi_cq_read** • UCX: ucp_tag_probe_nb + **ucp_worker_progress**	24

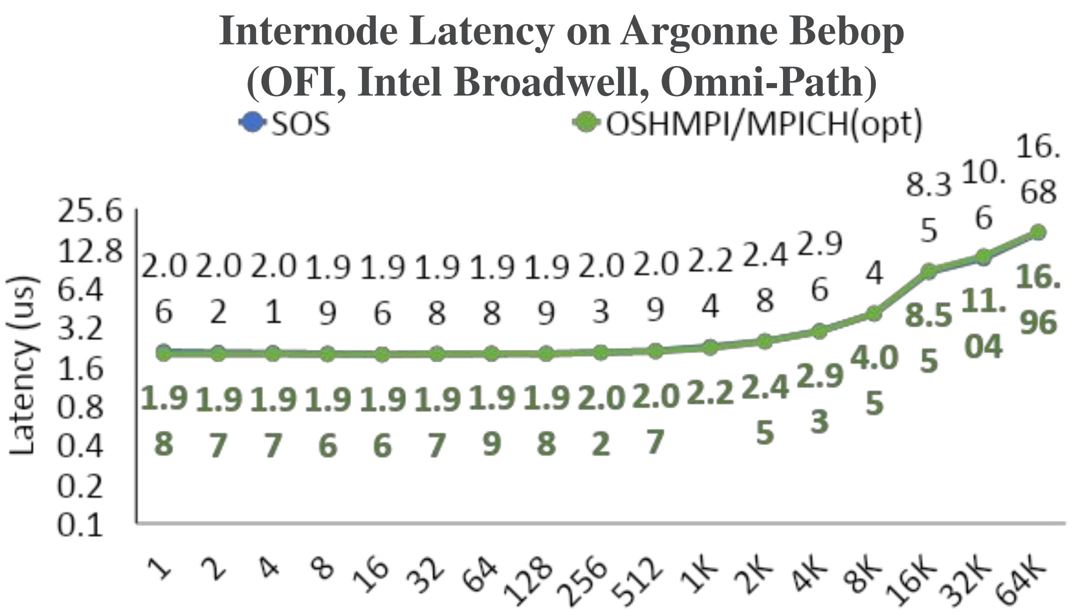
Progression in shmem_quiet	#Instr
Flush_all: wait network completion	14
Flush_all: MPI full progress	24

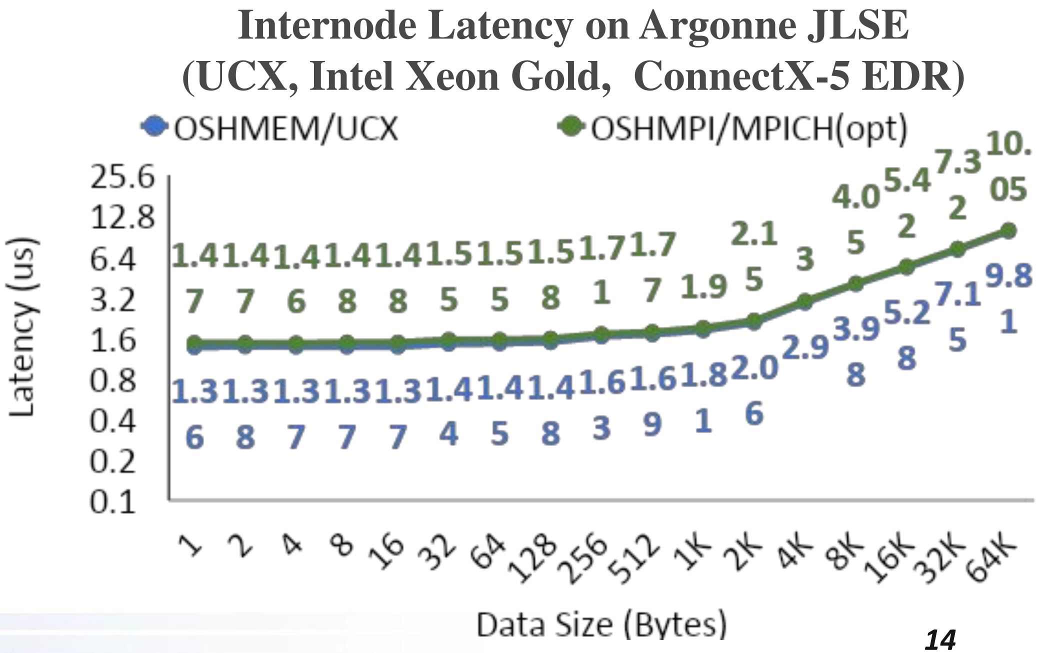
RMA Optimizations – Eliminate MPI Full-Progression (2)

- Breakdown of required progression:
  1. **Progress for MPI coll/pt2pt:** Full progress will always be triggered at the context of collectives and pt2pt (e.g., in MPI_Wait) to ensure their progression
  2. **Progress for MPI RMA AM:** If we know all RMA/AMO operations are directly offloaded to the network hardware (or SW emulation), MPI AM progression can be safely eliminated
  3. **Progress for OFI/UCX internal AM (for network SW emulation):** Can still be triggered by waiting on network completion in flush (e.g., fn_cntr_read in OFI, ucp_ep_flush in UCX)
- **Optimization (generic approach to avoid progression-2):**
  - **User hints:** specify extended info ”accumulate_op_types” and “rma_op_types” at window creation to describe the used AMO/RMA op, each with all datatypes and max element counts
    - E.g., “accumulate_op_types:cswap” = “int:1,long:1,longlong:1,uint:1,ulong:1,ulonglong:1,...” in OSHMPI
  - **Network capability:** query network the supported atomics/RMA with all possible datatypes and count limit at MPI_Init
  - MPI full progress can be safely skipped at win_flush{local, all} if all user-required AMO/RMA are supported by the network

Performance After Optimizations

- OSU benchmark osu_oshm_put
- Over OFI/Intel Omni-Path:
  - Optimized OSHMPI/MPICH delivers similar results as that of SOS in internode latency
  - No visible gap in internode message rate (graph omitted)
- Over UCX/Mellanox ConnectX-5:
  - OSHMPI/MPICH delivers only ~5% additional overhead compared to OSHMEM in internode latency
  - No visible gap in internode message rate (graph omitted)





OSHMPI AMO: Mismatched Atomic Semantics in MPI (1)

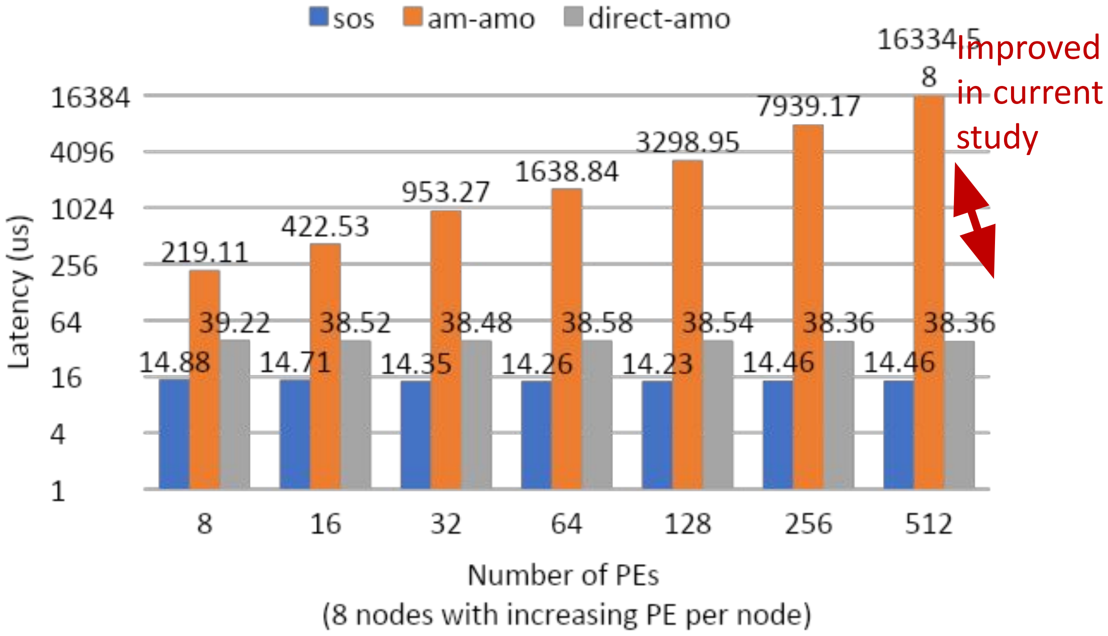
- Problem statement: Mismatched atomics semantics in OpenSHMEM and MPI
  - OpenSHMEM ensures atomicity between two different atomic operations (e.g., finc and fset)
  - MPI accumulate operations guarantee atomicity only between “same_op” operation or “same_op_no_op” operation (e.g., only when every PE performs finc, or finc + fetch)
  - **Cannot use MPI accumulates to directly implement OSHMPI atomics**
- Workaround in OSHMPI: implement active messages based AMO via MPI PT2PT
  - While ensuring correctness, performance is suboptimal due to additional costs such as manual progress polling (or enabling additional async thread) in OSHMPI
  - Support AMO with GPU memory heap becomes complex, i.e., OSHMPI layer has to handle vendor-specific GPU atomic reduce operations

OSHMPI AMO: Mismatched Atomic Semantics in MPI (2)

- **Solution**: Proposals to MPI standard (implementation is already available in OSHMPI/MPICH/OFI)
  - Replace original "accumulate_ops" with more specific "which_accumulate_ops=sum,replace..."
  - Default value is "all", which excepts MPI to guarantee atomicity between all atomic operations, same to OpenSHMEM spec.
    - Allow OSHMPI to directly use MPI accumulates in AMO
    - **Performance impact:** no network supports MPI atomic PROD, MINLOC, MAXLOC. Thus, the default "all" disallows utilization of network hardware atomics
      - Not a problem for OSHMPI, because we need only a subset of MPI ops (cswap,sum,no_op,replace,band,bor,bxor), which are likely supported by major networks
- **Further investigations**
  - Remote atomics are also limited by datatypes, number of elements, and required ordering of each op (e.g., some MPI <op,dataype> may not be supported by OFI provider, no war|waw|raw ordering support)
    - MPI MUST ensure all remote atomics to the same memory location can use hardware atomics
    - Involves additional checks before issuing network atomics, may increase software overhead

OSHMPI AMO: Performance Evaluation

- Platform: Argonne Theta
  - KNL nodes with Cray Aries network
- Evaluation with modified **osu_oshm_atomics**
  - All-to-one pattern**:** Let every PE issue AMO to PE 0
    - Mimic use of AMOs in real applications
  - SOS/OFI-gni: native implementation
  - OSHMPI/AMO-AM: MPI PT2PT based AM
  - OSHMPI/AMO-direct: Directly use MPI accumulates and enable network atomics
  - **Result summary**: Obviously direct AMO is more scalable than AM-based AMO; maintain low overhead with increasing number of PEs



Performance on Haswell (NERSC Cori): SOS 2.5us, OSHMPI/direct-amo 4.21us

* Summary of used MPI hints in AMO-direct:

- Required MPI hints to enable direct use of MPI accumulates: which_accumulate_ops="cswap,sum,no_op,replace,band,bor,bxor"
- Additional hints to enable network atomics: disable_shm_accumulate=true, accumulate_ordering=none,

Memory Space API with Memory Kinds

- Memory space proposal under investigation by the spec committee
  - https://github.com/openshmem-org/specification/wiki/Memory-Spaces
  - Goal: supporting different kinds of memory for symmetric heap
  - Led by Naveen N Ravichandrasekaran@HPE
- Memory space prototype in OSHMPI (subset of the entire proposal)
  - Omit teams in this prototype, but flexible to extend

```
typedef enum {
   SHMEMX_SPACE_HOST,
   SHMEMX_SPACE_CUDA,
   /* Other GPU kinds will be added */
} shmemx_space_memkind_t;
typedef struct {
   size_t sheap_size;
   int num_contexts;
   shmemx_space_memkind_t memkind;
} shmemx_space_config_t;
```

```
void shmemx_space_create(shmemx_space_config_t space_config, shmemx_space_t * space);
void shmemx_space_destroy(shmemx_space_t space);
void shmemx_space_attach(shmemx_space_t space);
void shmemx_space_detach(shmemx_space_t space);
int shmemx_space_create_ctx(shmemx_space_t space, long options, shmem_ctx_t * ctx);
void *shmemx_space_malloc(shmemx_space_t space, size_t size);
void *shmemx_space_calloc(shmemx_space_t space, size_t count, size_t size);
void *shmemx_space_align(shmemx_space_t space, size_t alignment, size_t size);
```

Key Semantics of Memory Space in OSHMPI

- shmemx_space_create(space_config, *space)
  - Allocate space heap based on space_config. sheap_size and memkind
  - PE local operation
- shmemx_space_attach(space)
  - Attach the space to all PEs (i.e., TEAM_WORLD), and prepare necessary communication resources (e.g., register the space heap to existing network endpoint, creating private endpoint for the space if space_config.num_contexts > 0)
  - Collective operation across all PEs
- shmemx_space_create_ctx(space, options, * ctx)
  - Return a private communication context (e.g., network endpoint) dedicated to the space
  - Allowing fast RMA/AMO: (1) skip dest-based lookup from all spaces (2) can wait for completion only on the dedicated context in quiet and fence
  - PE local operation

Implementation of Memory Space Communication in OSHMPI (1)

- Memory space can support two types of communication schemes:
  - **Scheme-1:** AMO/RMA with a space context
    - The operation can access only to the corresponding space heap. SHMEM_QUIETE will only complete operations issued to the space heap
  - **Scheme-2:** AMO/RMA without specific context (CTX_DEFAULT)
    - The operation can access to any location of the global symmetric data, symmetric heap, and space heap. SHMEM_QUIET ensures completion of all outstanding AMO/RMA.
- Communication support for **scheme-1** is relatively straightforward
  - At collective space_attach call: create a dedicated internal window for each space context (number of wins == space_config. num_contexts)
  - At each AMO/RMA operation: directly use the dedicated window in MPI operation

Implementation of Memory Space Communication in OSHMPI (2)

- Communication support for **scheme-2** is more challenging
  - ***Approach 1***: Similar to existing OSHMPI implementation, create separate window for each space by using MPI_Win_create
    - At each RMA/AMO operation, lookup the corresponding window based on the dest address
    - At quiet/fence, flush the default symm_heap|data windows as well as all space windows.
    - *Cons*: **Not scalable when number of spaces (=number of MPI windows) becomes large**
      - Lookup overhead at each OSHMPI RMA/AMO – can be avoided by using space context
      - Consume expensive MPI internal resources required by large number of wins (each win contains a dup of COMM_WORLD to ensure separate communication environment)
        - » Limited number of available communicator->context_id (e.g., **2045** in MPICH, **32768** in Intel MPI, **4095** in IBM Spectrum MPI)
        - » Memory usage per window per PE is about 6KBytes

Implementation of Memory Space Communication in OSHMPI (3)

- ***Approach 2***: Create a single dynamic window and attach all heaps to this window
  - **Challenge: Dynamic window is not well optimized by most MPI implementations**
    - MPI_Win_attach is a **local** operation. MPI cannot exchange the info of attached memory regions (e.g., address, MR key) among processes, thus losing optimization opportunities (e.g., using network RDMA)
  - Can we optimize dynamic window RMA in the OSHMPI context?
    - In OSHMPI, symmetric heap and global data are **always collectively attached** at shmem_init. space_attach is a collective call, thus space heap is also collectively attached.
    - Extended MPI info for collective attach:
      - Specify info "**coll_attach**" at win_create_dynamic. If it is TRUE, means all attach calls are **collective**
      - MPI implementation can safely exchange memory region info at win_attach, thus leverage RDMA
    - *Pros*: Eliminate non-scalable resource consumption (i.e., all spaces share the same window and thus the same window internal communicator)
    - *Special concern*: May require remote MR key lookup at each OP inside MPI on some networks (e.g., OFI/uGNI, UCX)
      - If dedicated communication resource is needed (e.g., no MR key lookup), create space context !

Performance Analysis of Memory Space Approaches

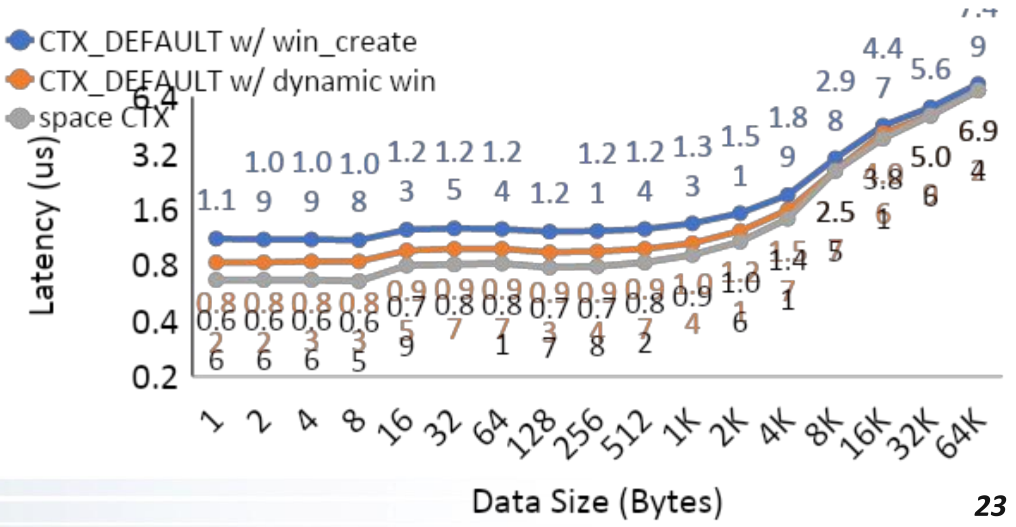
- Memory usage analysis (per space per PE)
  - $M_{space} = M_{metadata} + M_{winobjs} + M_{mrkey}$
    - $M_{metadata}$: is constant per space
    - M_{mrkey}: is same as that consumed by native OpenSHMEM (required by network)
    - $M_{winobjs}$: Each object takes about 6KB in MPICH
      - Additional usage in OSHMPI/MPICH
        - » Win_create windows: $6KB * N_{spaces}$
        - » Dynamic window: $6KB$

Total additional memory usage in OSHMPI/MPICH compared to native OpenSHMEM on 1024 processes (in MB)

Num of spaces	Win_create	Dynamic win
1	6.1	6.1
16	97.6	6.1
256	1562.0	6.1
4096	24992.0	6.1

- Latency analysis for memory space with host memory
  - With CTX_DEFAULT, dynamic win reduces up to 20% overhead compared to win_create
    - Flush only a single window at QUIET
  - Space CTX further reduces the overhead by skipping space traversal at QUIET and space lookup at PUT

Intra-socket Latency with 100 spaces on Argonne Bebop



Other Things

- Support for GPU buffers
- GPU-Initiated/-Triggered Operations
- OpenSHMEM 1.5 New Features
  - Teams
  - Team-based Collectives
  - Nonblocking AMO
- OSHMPI inside MPICH

Thank you!

Yanfei Guo

Argonne National Laboratory

yguo@anl.gov