Application-Level Optimization of On-Node Communication in OpenSHMEM

Md. Wasi-ur- Rahman, David Ozog, and James Dinan

Legal Disclaimers

Outline

- Introduction and Motivation
- Challenges
- Design of an on-node data sharing library, **`shnode`**
- Performance Evaluation
- Future Work and Conclusion

Introduction and Motivation

- PGAS programming models provide seamless ability to perform one-sided remote memory operations
  - No explicit participation from the remote Processing Element (PE)
- For on-node PEs, such operations can result in performance overheads due to
  - Multiple copies of the same data within a shared memory address space
  - Synchronization mechanisms associated with distributed memory access patterns
  - Memory replication cost of single process multiple data programming
- OpenSHMEM community is actively investigating library extensions to better support node-level optimizations

Introduction and Motivation

- OpenSHMEM Context extension provides thread-safety by isolating communication streams
  - Optimizes communication performance through overlap with each other
- Paris OpenSHMEM
  - High performance communication engine based on the Boost library
- Application developers may still opt for evolutionary approach for data sharing
  - Most OpenSHMEM implementations support a query function that provide a direct pointer to the remotely accessible memory of a remote PE

Problem Statement

How can application developers avoid/minimize on-node communications with the information provided by the built-in OpenSHMEM routines?

Outline

- Introduction and Motivation
- Challenges
- Design of an on-node data sharing library, **`shnode`**
- Performance Evaluation
- Future Work and Conclusion

Challenges

- Current implementations of `shmem_ptr` provide limited information
- Most OpenSHMEM implementations do not provide an easy way to identify the local PEs and the data object references
- Leader based collective implementation can be beneficial for many applications, for which no abstraction is present
- Tuning to identify the optimum number of local leaders for each collective operation needs to be investigated

Existing Works

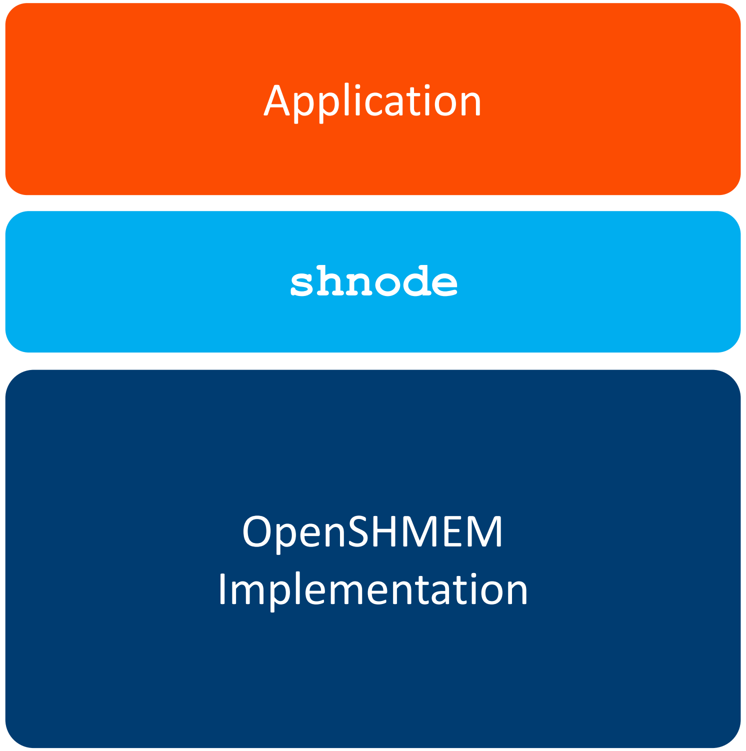
- Welch et al. introduced teams and spaces concepts for gathering on-node groups of PEs
- Hoefler et al. introduced MPI+MPI to enable inter-process communication through shared memory windows
- Cray-SHMEM provides APIs to discover local PEs and building a team through `shmem_local_ptr` and `shmem_team_translate_pe`

Outline

- Introduction and Motivation
- Challenges
- Design of an on-node data sharing library, **`shnode`**
- Performance Evaluation
- Future Work and Conclusion

Design of **`shnode`**

- Utilizing the built-in `shmem_ptr` routine, we design **`shnode`**
- Purpose is to provide application developers a way to minimize on-node communication
- Stores the data object references for other PEs located on the same node
- Subsequent remote memory operations can be substituted with direct load and stores
- For each on-node team, the lowest rank PE is assigned as the leader



Proposed fundamental APIs for **shnode**

```
int shnode_init();/*initialization*/
int shnode_create_team (void *data);            /*team creation*/
int shnode_add_data (void *data);
                                   /*addition of data objects*/
int shnode_is_team_member (int rem_pe);         /*member check*/
void *shnode_get_member_remote_addr (int rem_pe, void *data);
/*retrieval of memory address*/
int shnode_am_team_leader();                    /*leader check*/
int shnode_finalize();                              /*destroy*/
```

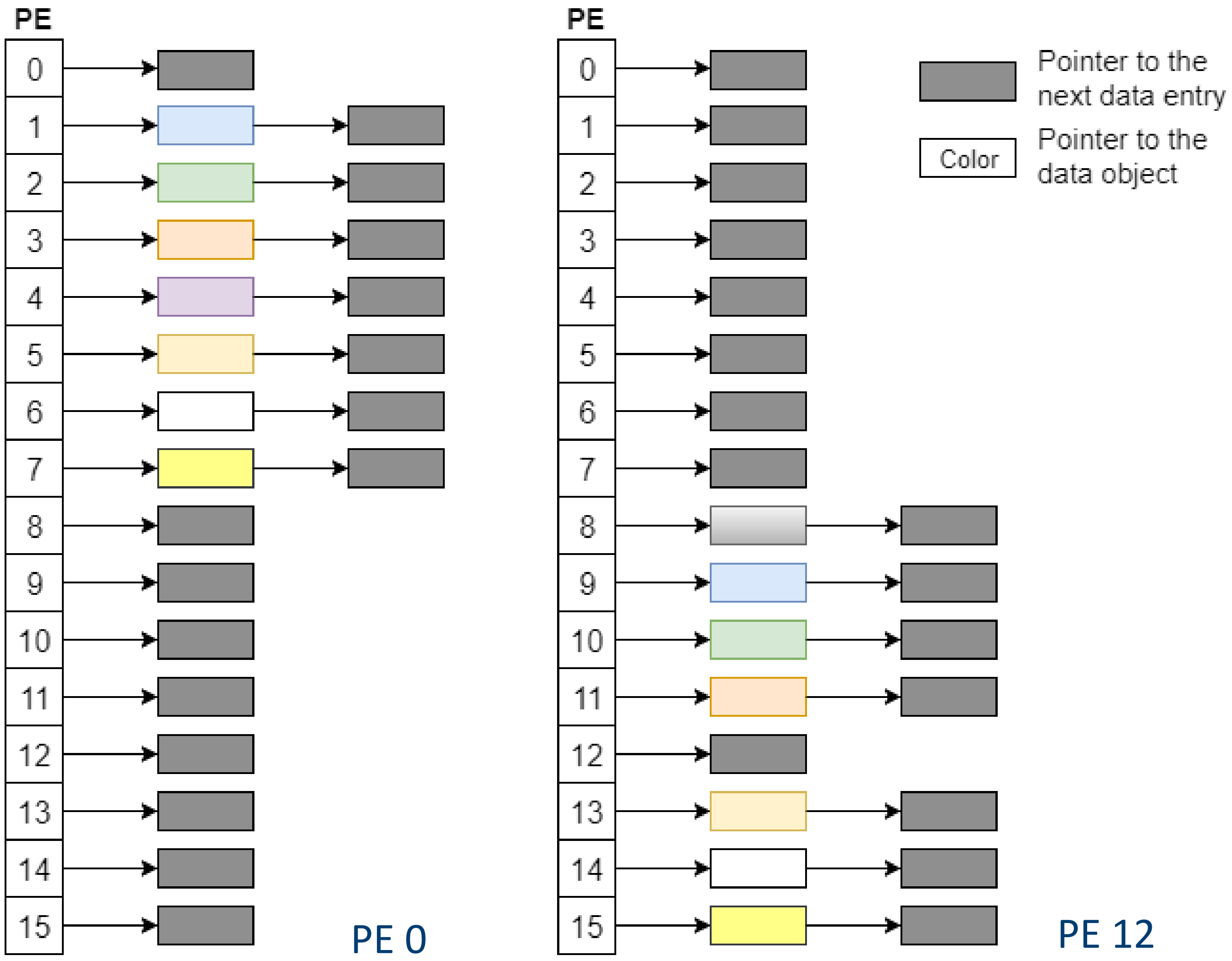
Used with shmem_ptr to populate the on_node PEs' table

Additional data can be added to the table using shmem_ptr for the team PEs

Retrieves the address from the stored team table created by shnode_create_team

Data Structure for **`shnode`**

- Simple data structure mapping each PE to a list of data object references
- A second list is maintained that maps the desired object to the location it is stored on the first map list
- References to PEs on other nodes and self are kept empty
- This data structure is populated through `shnode_create_team` which is invoked only once at the beginning



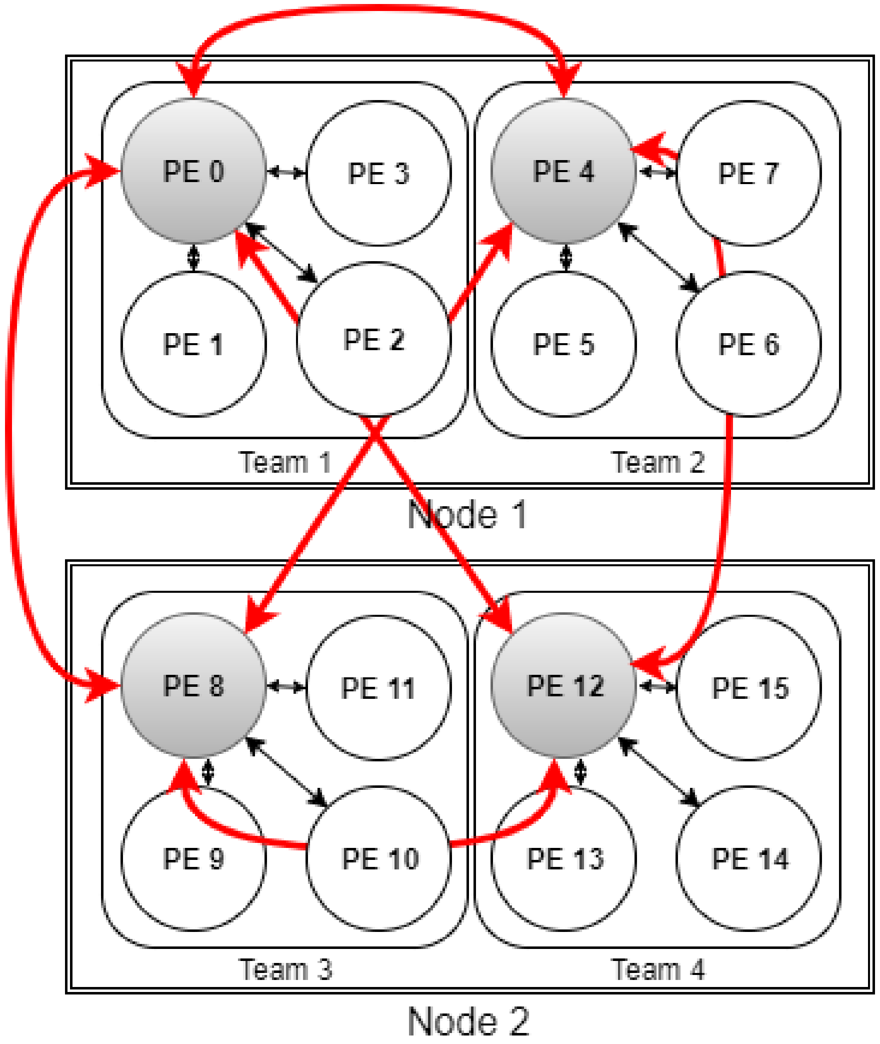
Example all-to-all program utilizing **shnode**

```
shmem_init();
…

data1 = shmem_malloc (size1);

…

for (rem_pe = 0; rem_pe < npes;
rem_pe++) {
    shmem_put(dest, src,
nelems, rem_pe);
}

…

shmem_finalize();
```

```
shmem_init();
shnode_init();
data1 = shmem_malloc (size1);
shnode_create_team (data1);
for (rem_pe = 0; rem_pe < npes;
rem_pe++) {
     if (shnode_is_team_member
     (rem_pe)) {
          void *ptr = shnode_get
          _member_remote_addr(rem_pe,
          data1);
          memcpy(…);
     } else {
          shmem_put (dest, src,
     nelems, rem_pe);
     }
}
shnode_finalize();
shmem_finalize();
```

Designing helper routines for Collectives

- Each collective operation can be divided into three sub-tasks based on the teams formed by **`shnode`**
  - The local PEs transmit their data to the corresponding leaders
  - A collective operation strided over all the leaders across the nodes takes place
  - Leaders transmit the collected value to the corresponding local PEs
- A power-of-two number of processes per node is assumed to be launched for the current implementation



Example `int_sum_to_all` utilizing **shnode**

```
shmem_init();
…

//shmem_int_sum_to_all (…);
shnode_int_sum_to_all (…);

…

shmem_finalize();
```

```
If (leader == ME) {

     for (all members in team) {
          //copy data from src_ptr to the
          self source
     }

     shmem_barrier(…);    //strided over
     shmem_int_sum_to_all(…); //all the
                              //leaders

     for (all members in team) {
          //copy data from self destination
          to dst_ptr
     }
}
shmem_barrier_all();
```

Designing better overlapping with communication

- **`shnode`** provides the opportunity to the application developers to replace the remote memory operations with direct load/store
- If applicable, pointer swapping can bring further benefits as it removes all memory to memory data transfer
- Another alternative is to schedule the inter-node and intra-node data transfers separately
- Scheduling the intra-node memory operations at the end may ensure better overlapping between computation and inter-node transfers
- Re-structuring of the communication and computation can be beneficial with **`shnode`**

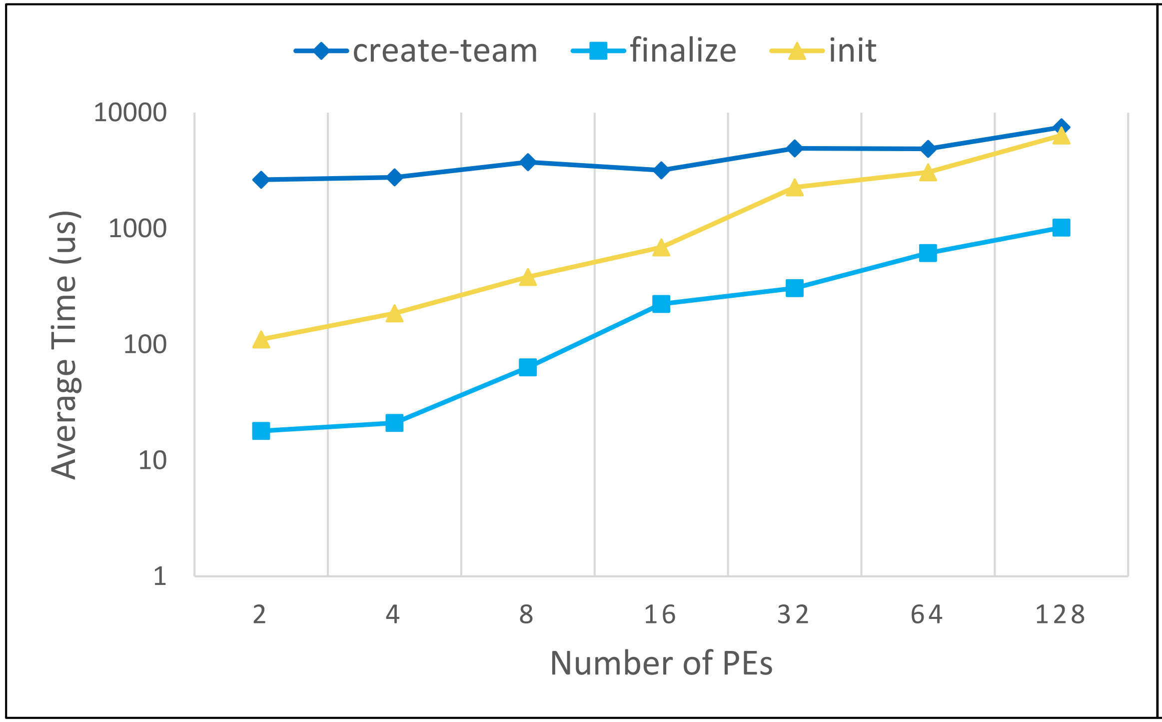
Outline

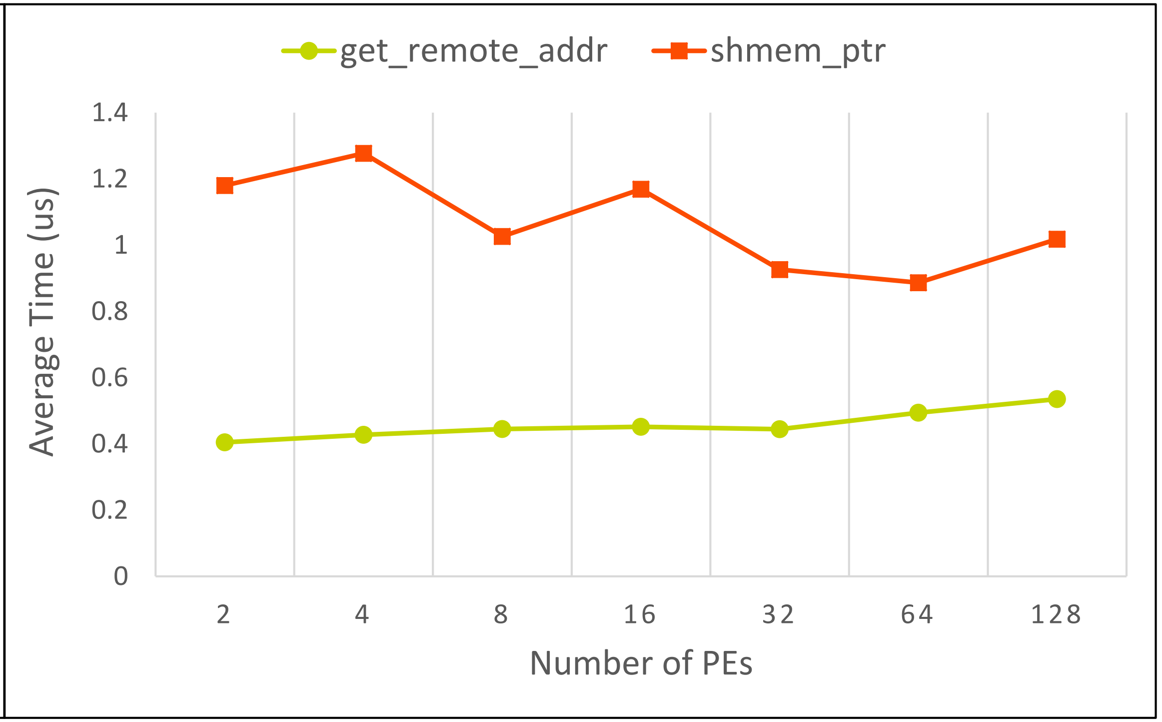
- Introduction and Motivation
- Challenges
- Design of an on-node data sharing library, **`shnode`**
- Performance Evaluation
- Future Work and Conclusion

Experimental Setup

- NERSC system, Cori
  - Cray XC40
  - Intel® Xeon Phi™ 7250 (Knights Landing), 68 cores/node @ 1.4 GHz
  - 96 GB DDR4 memory
- **`shnode`** was implemented on top of Cray SHMEM v7.5.5

Profiling **shnode** APIs

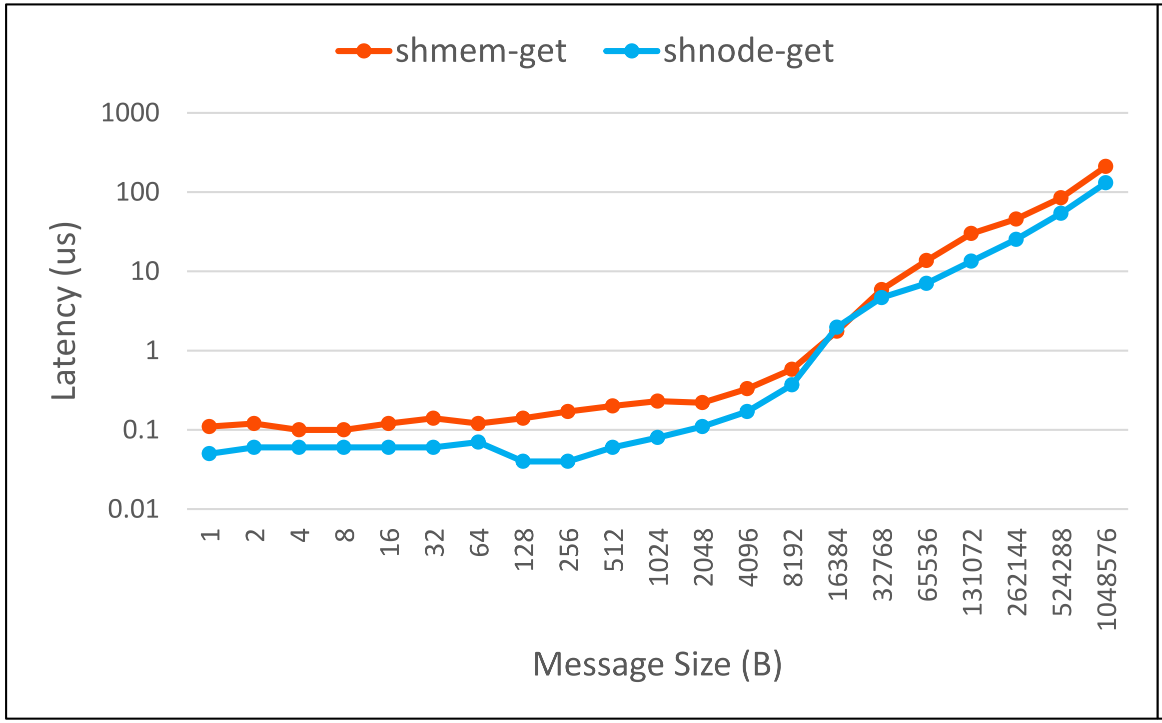


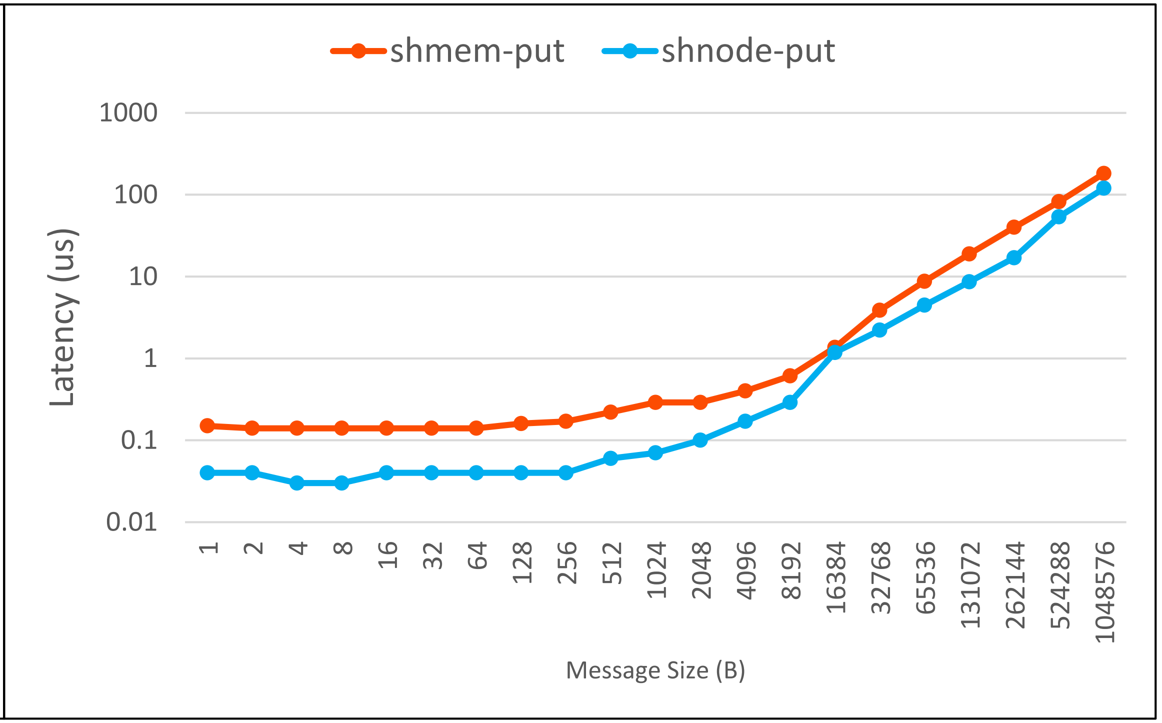


- Profiling 4 APIs on Two KNL nodes
- PEs per node is varied from 1 to 64
- Average execution time is taken across all PEs

- `init`, `create_team`, and `finalize` take less than 0.1 seconds for 128 PEs
- **`shnode`** implementation can reduce the query operation cost by 50% compared to the default approach

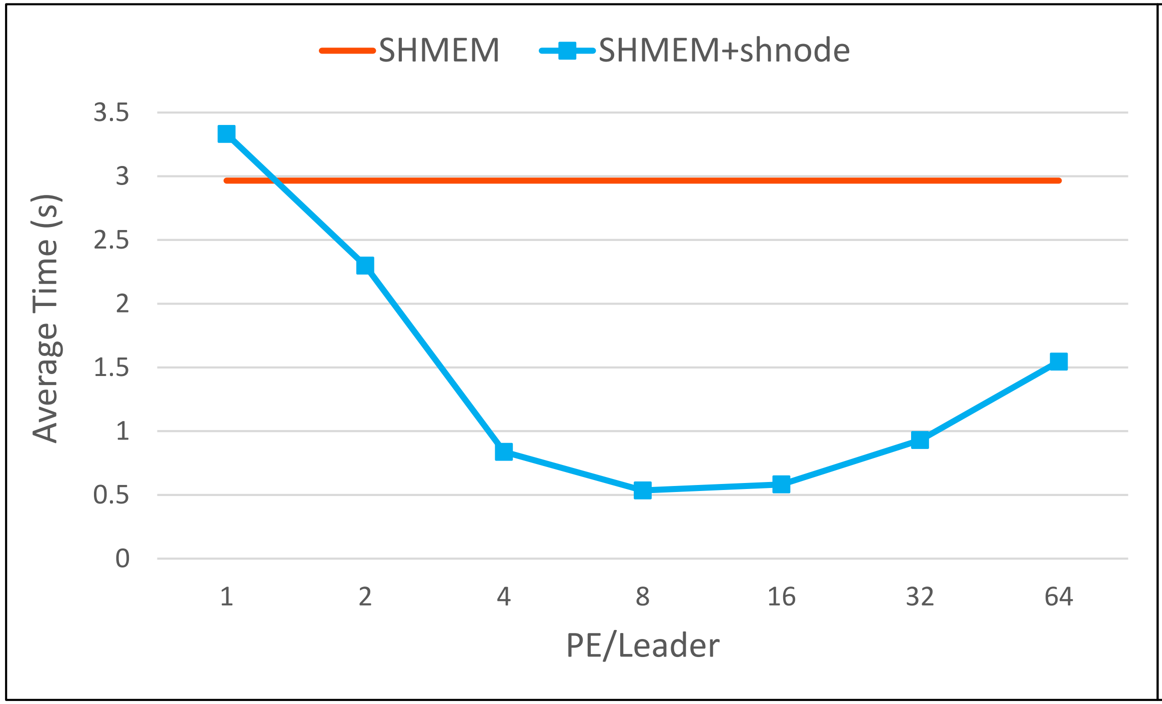
Evaluation with OSU micro-benchmark

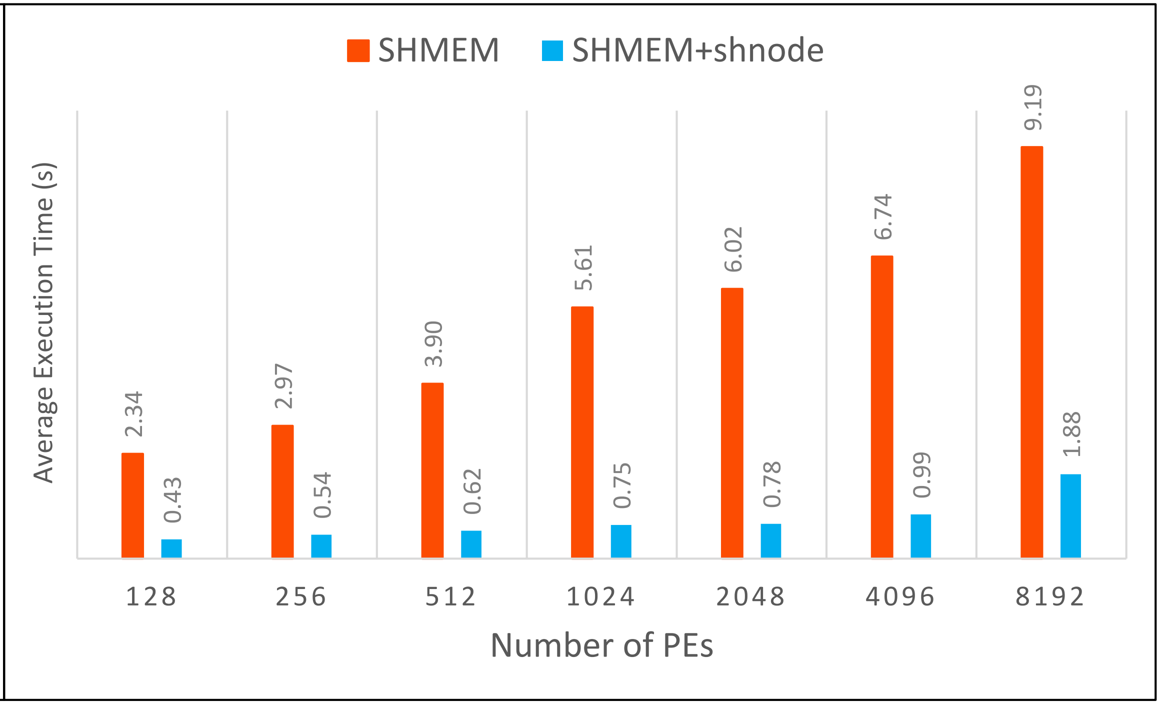




- Put and Get performance evaluation with 2 PEs
- Modified OSU benchmarks to evaluate with **`shnode`**
- For small message sizes, both shnode-put and shnode-get can perform 3-4.6x better; for large message sizes, benefit is around 1.5-2.35x compared to the default

Evaluation of Collectives (`int_sum_to_all`)

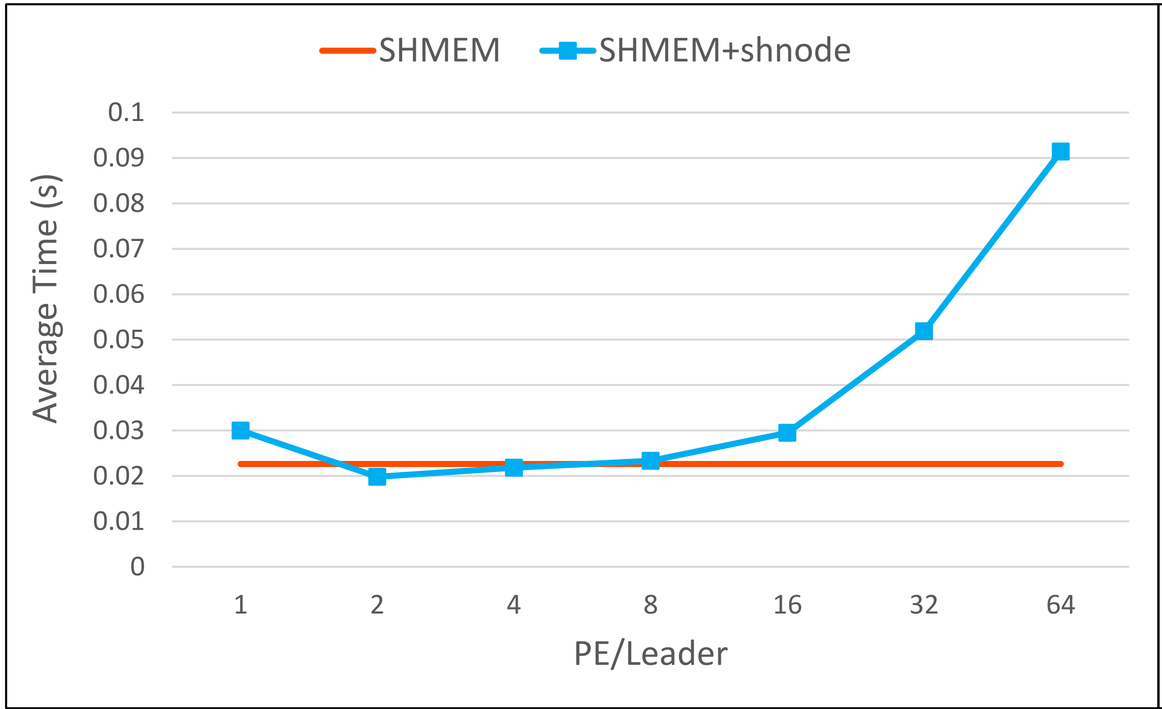


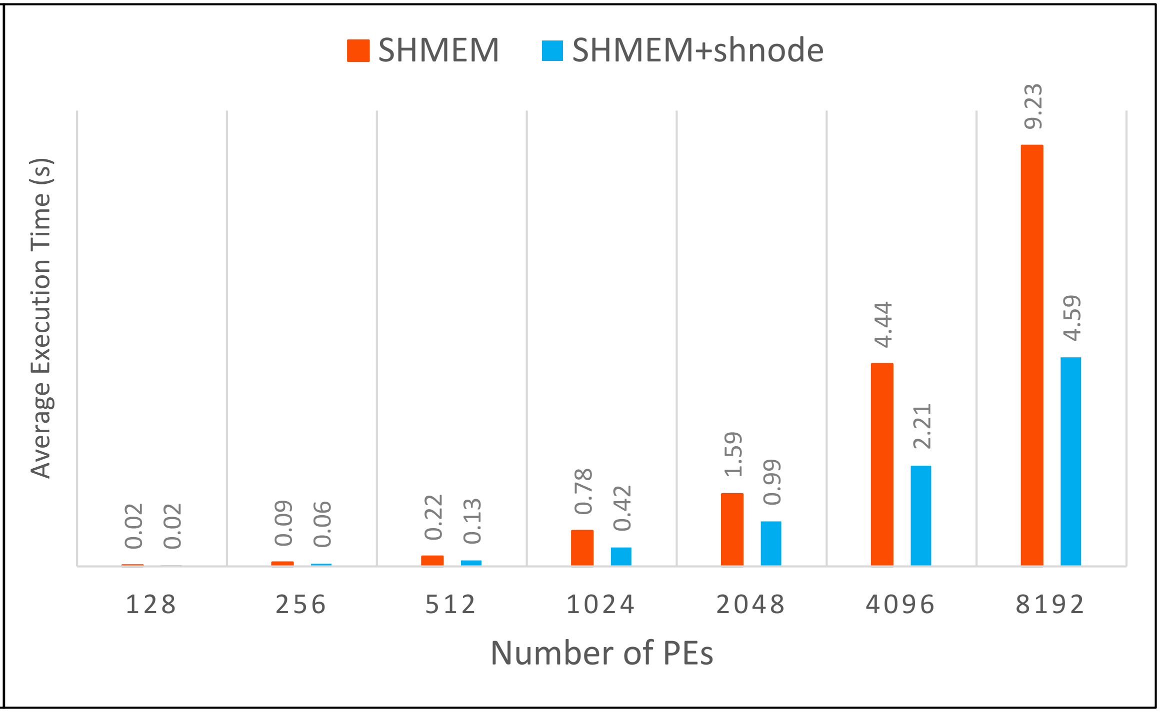


- Analyze the impact of multiple leaders on 4 nodes with PEs per leader from 1 to 64
- Evaluated with 128 (2 nodes) to 8,192 (128 nodes) PEs
- 10MB buffer used as the source data; averaged over 10 iterations

- Multiple leaders/node achieve better performance compared to the default; 8 PEs for each leader provides optimum
- **`shnode`** provides 4.87x benefit compared to the default implementation for 8,192 PEs

Evaluation of Collectives (`fcollect`)

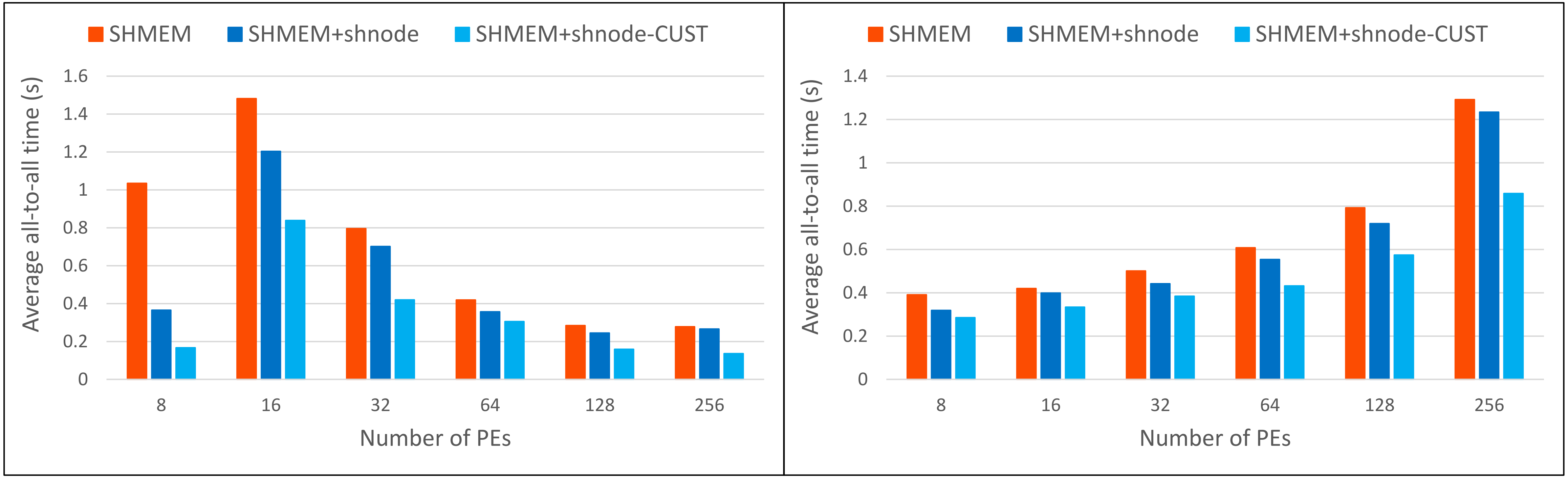




- Analyze the impact of multiple leaders on 4 nodes with PEs per leader from 1 to 64
- Evaluated with 128 (2 nodes) to 8,192 (128 nodes) PEs
- 10MB buffer used as the source data; averaged over 10 iterations

- 2 PEs for each leader provides optimum; more PEs per leader introduces overhead
- **`shnode`** provides 2x benefit compared to the default implementation for 8,192 PEs

Evaluation of ISx



- Strong and weak scale experiments for ISx on 4 nodes with PEs varying from 8 to 256
- 1.5 billion items to sort for strong scale; for weak scale, the number of items per PE is 33 million
- Node-local transfers are separated for the SHMEM+shnode-CUST version of ISx
- Without customized communication, **`shnode`** provides little benefit (~5%) compared to the SHMEM version
- With customized communication pattern, **`shnode`** provides 1.5 - 2x benefit for both strong and weak scaling

Outline

- Introduction and Motivation
- Challenges
- Design of an on-node data sharing library, **`shnode`**
- Performance Evaluation
- Future Work and Conclusion

Future Work

- Collective operations with any number of PEs per node
- Enabling/Disabling **`shnode`** features through configuration variables within OpenSHMEM
  - Collectives
  - RMA functions
  - Other helper query routines
- Exploring other applications to extract benefits from **`shnode`**
  - Re-ordering to achieve communication avoidance for stencil algorithm
- Exploring performance improvement potential for MapReduce applications
  - For shuffle and reduce sensitive applications, **`shnode`** may provide further benefits based on the data transmission and reduce function characteristics

Conclusion

- **shnode** supports the formation of node-local teams within which applications can do shared memory operations
- We present a set of APIs for **shnode** that can be used to create teams as well as nominating single/multiple leader processes
- Number of leaders has a significant impact on collective performance
- **shnode** APIs has less overhead compared to the default available APIs
- For `int_sum_to_all`, **shnode** can bring 4.87x benefit compared to the default approach by using multiple leaders
- By re-ordering the computation and communication phases, ISx can be improved by 1.5x using **shnode**

Questions?