Lightweight Instrumentation and Analysis using OpenSHMEM Performance Counters

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

Legal Disclaimers

Outline

- Introduction and Motivation
- Existing Approaches
- Performance Counter APIs
- Design and Implementation of a Collector
- Experimental Analysis
- Conclusion and Future Work

Introduction and Motivation

- Effective performance profiling and analysis tools for PGAS applications have been challenging
  - One-sided high-throughput usage model
  - Scale of parallel applications
  - Rate and volume of communication operations generated
- Tracing is the most common approach
  - Captures a log of each operation for offline analysis
  - Instrumentation introduce overhead and impact dynamic behavior of applications
- Can an alternative lightweight instrumentation approach be devised that skip library interposition, yet achieve detailed profiling for communication performance?

Outline

- Introduction and Motivation
- Existing Approaches
- Performance Counter APIs
- Design and Implementation of a Collector
- Experimental Analysis
- Conclusion and Future Work

Existing Approaches

- A number of tools provide communication tracing and analysis for OpenSHMEM applications
  - Collect detailed information
  - Plug-in/out capabilities
  - User-friendly interfaces
- Can generate per-operation overhead
- Requires library interposition
- Using hardware performance counters, e.g. PAPI is challenging for process-level application performance analysis

CrayPAT

HPCToolkit

KOJAK

PAPI

Score-P
Scalable performance measurement infrastructure for parallel codes

scalasca

Our Approach

- Performance profiling using network and library counters through well-defined SHMEM APIs
- Associate performance information to process level as well as contexts within a process
- Simplest design of collector that impose low overhead to the application runtime
- Profiling analysis and optimization strategies proposed in this work can be applicable for other PGAS models

Outline

- Introduction and Motivation
- Existing Approaches
- Performance Counter APIs
- Design and Implementation of a Collector
- Experimental Analysis
- Conclusion and Future Work

OpenSHMEM Performance Counters

- Unsigned 64-bit integers
- Follow C language rules for unsigned integer arithmetic
- Monotonically increasing over the duration of program execution
- Incremented 0 or more times by SHMEM operations
  - One single large put operation can be fragmented to several smaller writes
  - Operation performed through shared memory rather than fabric

Implementation in Sandia OpenSHMEM

- Different design choices available for APIs based on operation type, input arguments, and return values
- Two class API for per-context counters
  - Operations reading data from a symmetric object (get, fetch AMO)
  - Operations writing data to a symmetric object (put, non-fetch AMO)
- Each context utilizes
  - Middleware level counters for issued operations
  - Fabric level event counters for completed operations
- Tracks number of fabric operations that have completed in the local memory
  - Associated with local process instead of a particular context

Proposed APIs

```
/* Retrieve write operation counters */
int shmemx_pcntr_get_issued_write(shmem_ctx_t ctx, uint64_t *cntr_value);
int shmemx_pcntr_get_completed_write(shmem_ctx_t ctx, uint64_t *cntr_value);

/* Retrieve read operation counters */
int shmemx_pcntr_get_issued_read(shmem_ctx_t ctx, uint64_t *cntr_value);
int shmemx_pcntr_get_completed_read(shmem_ctx_t ctx, uint64_t *cntr_value);

/* Retrieve target operation counters */
int shmemx_pcntr_get_completed_target(uint64_t *cntr_value);

/* Retrieve all operation counters */
int shmemx_pcntr_get_all(shmem_ctx_t ctx, shmemx_pcntr_t *pcntr);
```

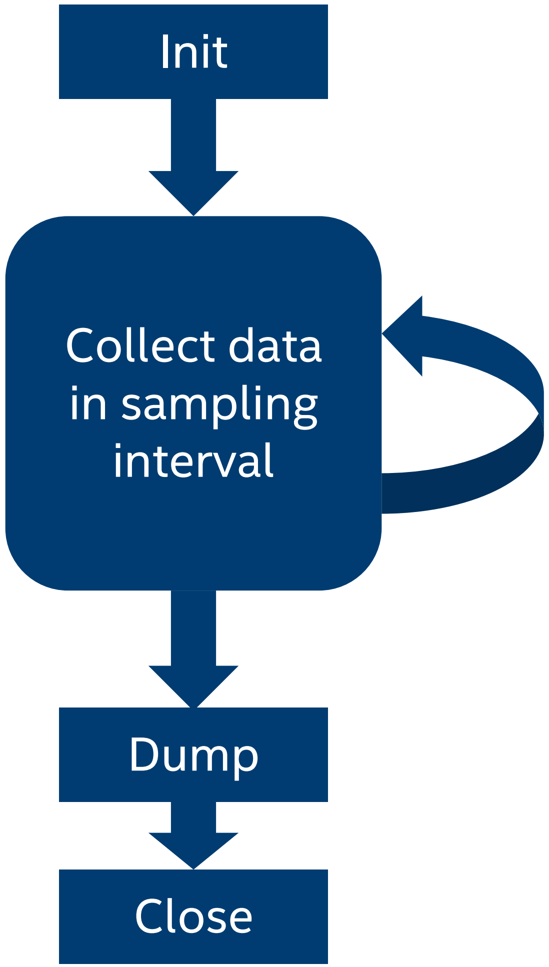
Object to hold all counter values

Outline

- Introduction and Motivation
- Existing Approaches
- Performance Counter APIs
- Design and Implementation of a Collector
- Experimental Analysis
- Conclusion and Future Work

Design of a Collector

- Simple design choices to collect the performance counter data
- Initiated as a thread to run alongside with the calling PE
- Sampling Interval defines the rate at which the data is collected; controlled by a runtime parameter
- Samples are timestamped and stored in memory
- Samples are discarded when there has been no change in the counter values since the last collection
- By default, collects the data for `SHMEM_CTX_DEFAULT`
- Additional contexts can be added and removed during runtime; maximum number of allowable contexts can be controlled via a runtime parameter
- Stored samples are dumped in a simple CSV format



Example program utilizing the Collector

```c
shmem_init();
…

shmem_ctx_create(&ct

for (rem_pe = 0; rem
rem_pe++) {
    shmem_ctx_put(ctx, dest,
src, nelems, rem_pe);
}

…

shmem_finalize();
```

Contexts must not be created with SHMEM_CTX_PRIVATE/ SHMEM_CTX_SERIALIZED

```c
shmem_init_thread(SHMEM_THREAD_
MULTIPLE, &tl);
start_collect();

shmem_ctx_create(&ctx);
register_context(ctx);

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

remove_context(ctx);
stop_collect();
shmem_finalize();
```

Outline

- Introduction and Motivation
- Existing Approaches
- Performance Counter APIs
- Design and Implementation of a Collector
- Experimental Analysis
- Conclusion and Future Work

Experimental Setup

- Cluster with 14 compute nodes
  - Intel® Xeon® E5-2699 V3, 36 cores/node @ 2.3 GHz
  - 64 GB DDR4 memory
  - Intel® Omni-Path Fabric
- Performance counter APIs are implemented on top of Sandia OpenSHMEM (SOS), git `#908682ee`
- Applications
  - Integer Sort (ISx)
  - Stencil from Parallel Research Kernel (PRK)

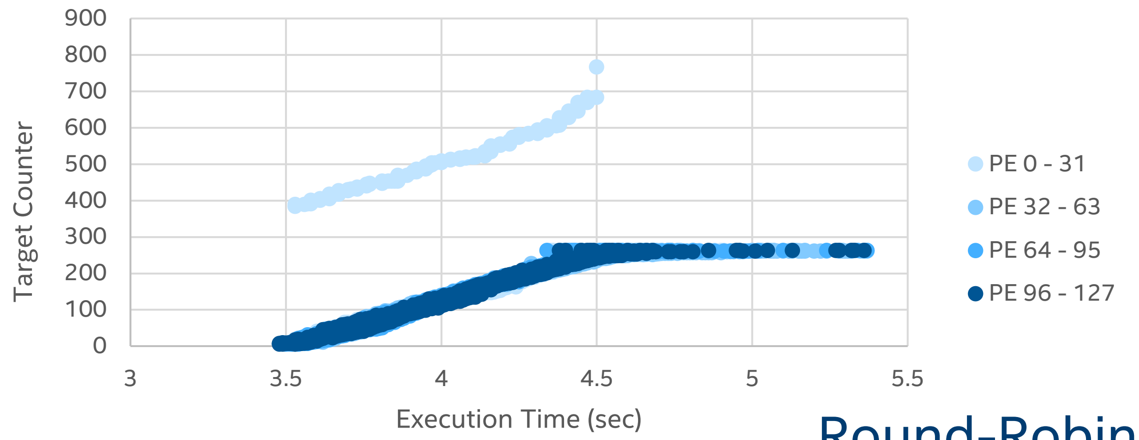
Outline

- Introduction and Motivation
- Existing Approaches
- Performance Counter APIs
- Design and Implementation of a Collector
- Experimental Analysis
  - Communication Schedule
  - Overlap
  - Load Balance
  - Weak Scale Analysis
  - Overhead
- Conclusion and Future Work

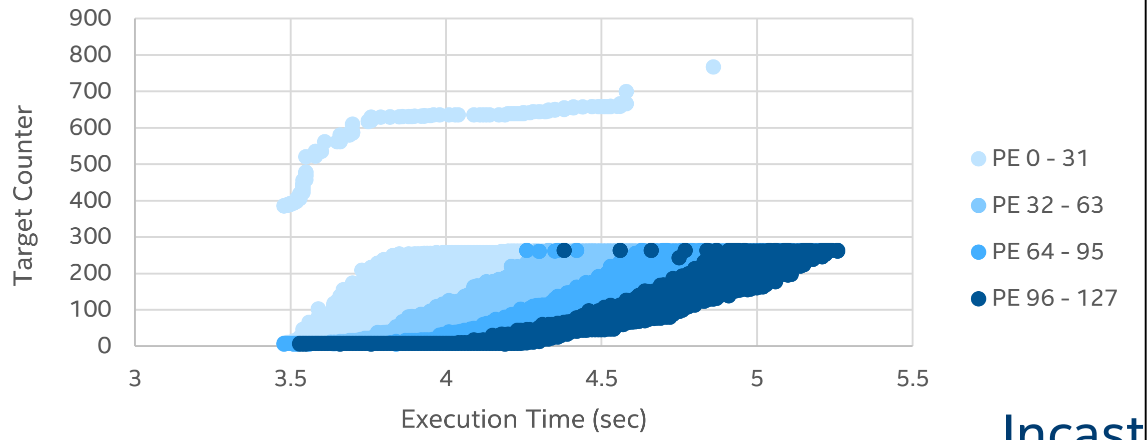
Communication Schedule

- Defines the next target PE in an all-to-all key exchange for each PE
- ISx implements three different communication scheduling pattern
  - Round-Robin (default): Chooses the next PE based on the given PE’s rank and loops over a circular array of PEs
  - Incast: Iterates over an array of PEs from *0, 1, 2, … n*
  - Permute: Iterates over a random array of PEs
- Target counter progression follows different trend for different communication schedule
- Divide the total number of PEs into four groups to highlight the differences in progression

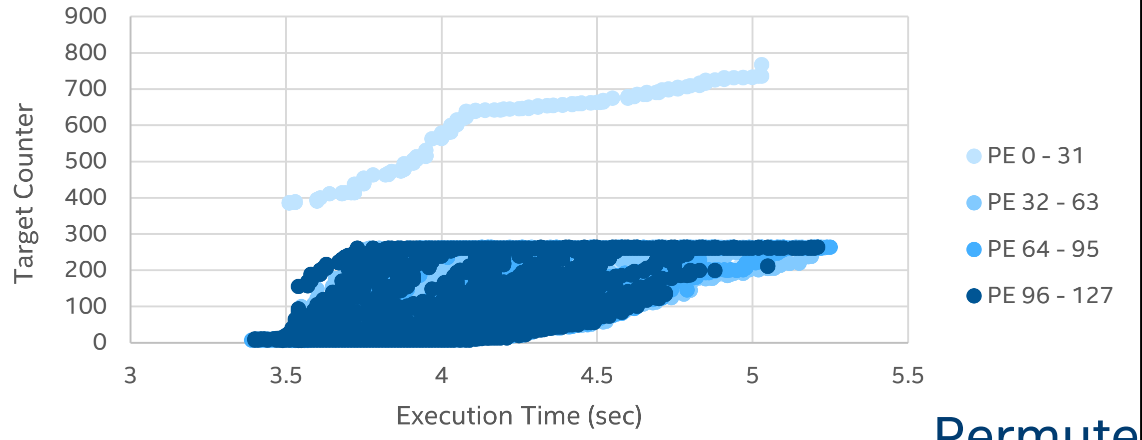
Communication Schedule



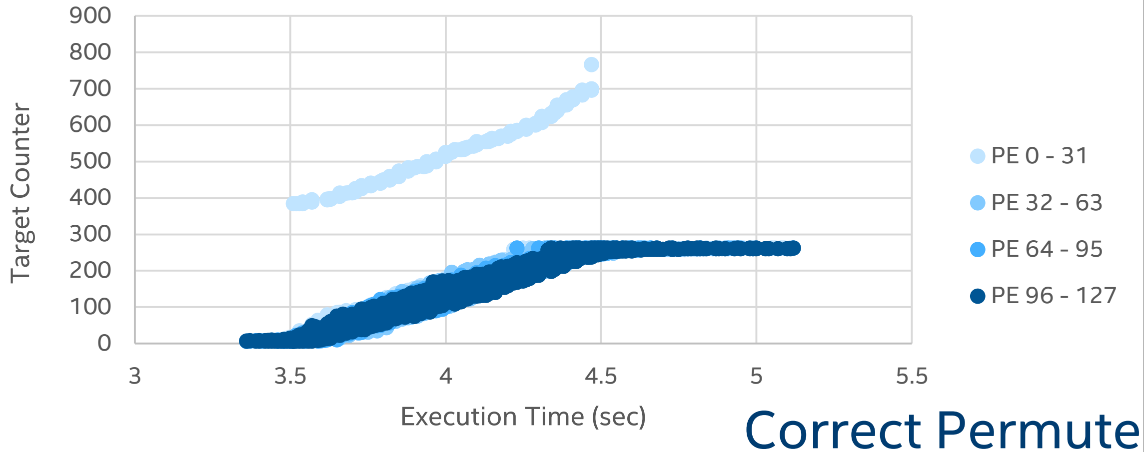
Round-Robin



Incast



Permute



Correct Permute

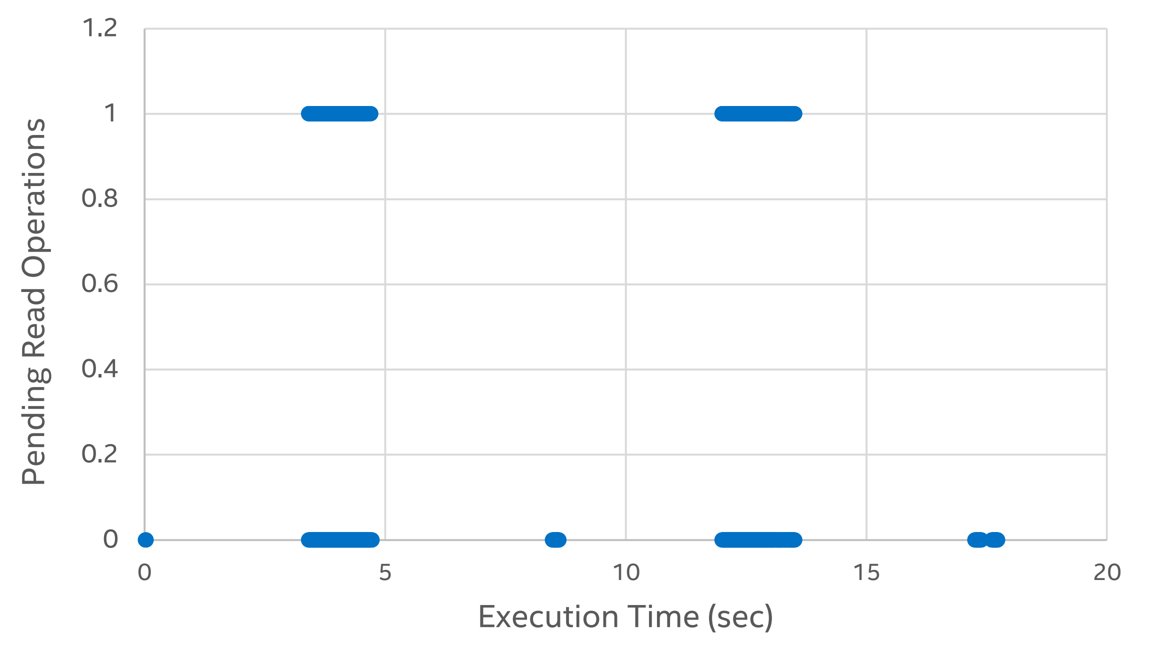
Outline

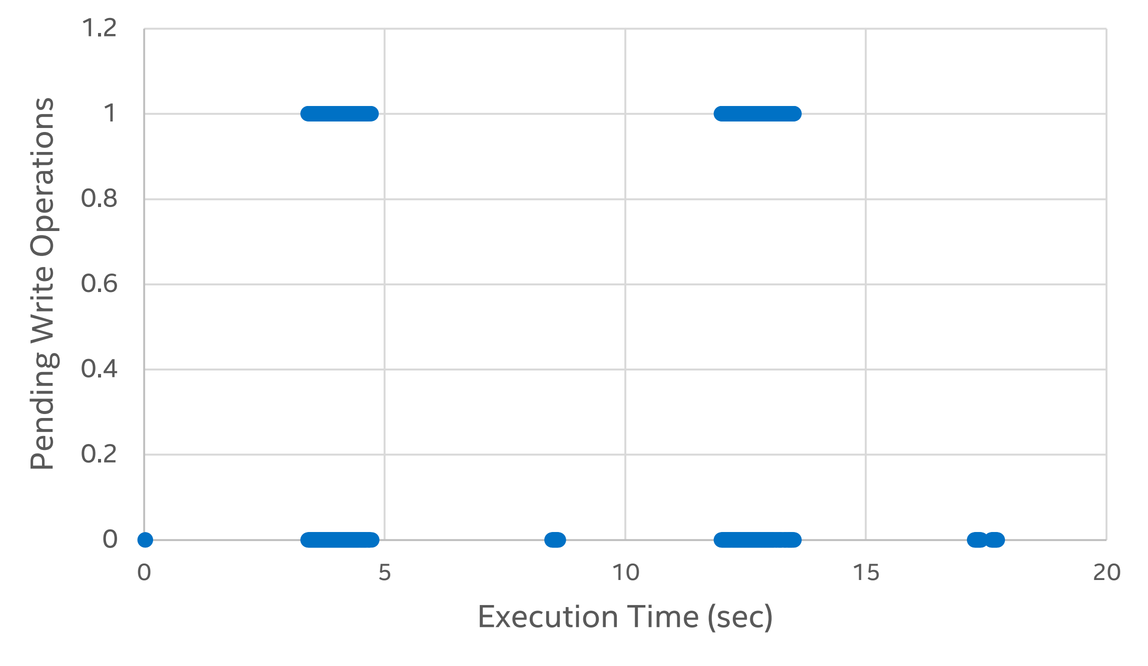
- Introduction and Motivation
- Existing Approaches
- Performance Counter APIs
- Design and Implementation of a Collector
- Experimental Analysis
  - Communication Schedule
  - Overlap
  - Load Balance
  - Weak Scale Analysis
  - Overhead
- Conclusion and Future Work

Overlap

- Observe the dynamic differences between the posted and completed operation counters
- Analyze the opportunities to introduce communication overlap
- Use ISx for this analysis and apply different optimization strategies based on the counter values
- Focus on the counter changes in the key exchange routine through
  - Pending read/write operations
  - Issued write operations w.r.t. completed read operations

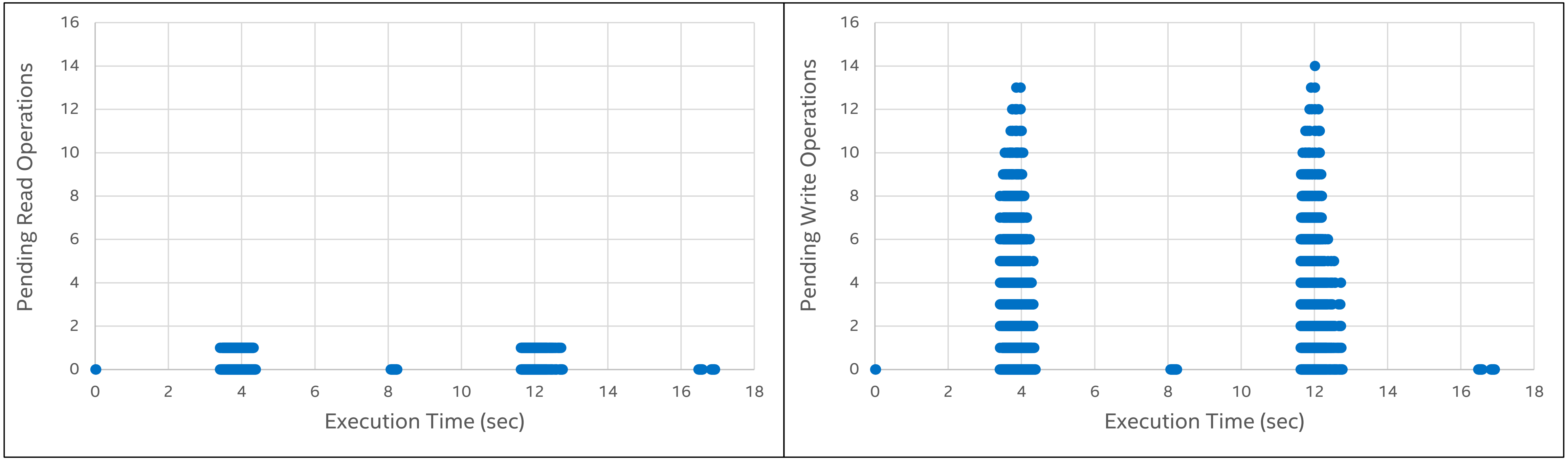
Pending Operations with default ISx





- Pending operation counters (difference between the issued and completed counter values) over execution time
- Both read (left) and write (right) counter values reveal only one pending operation at any given time – presenting the opportunity for the usage of non-blocking APIs

Pending Operations with Non-blocking Put



- Replace Blocking Put with Non-blocking API
- Pending read operations are unchanged as no overlap is introduced; Pending write operations increase to at most 14 during the key exchange execution
- Further overlap is possible through non-blocking read (non-blocking `fadd` AMO)

Non-blocking AMO in ISx key exchange

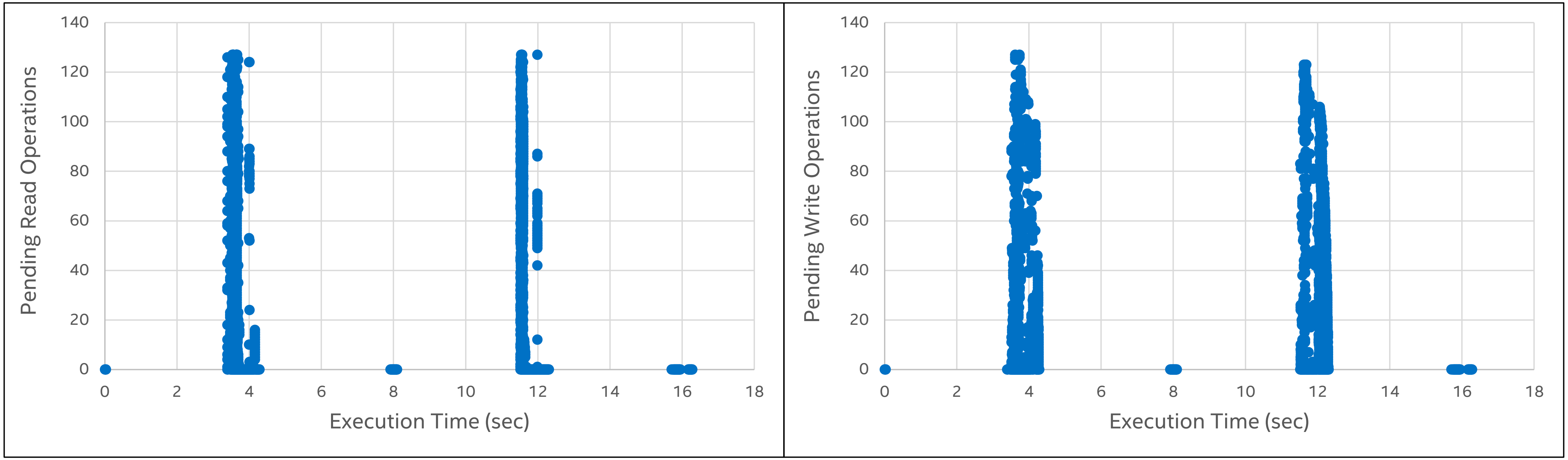
```
for (int i=0; i<shmem_n_pes(); i++) {
  int dest_pe = peers_iter(i);
  long long dest_offset = shmem_longlong_atomic_fetch_add(
                &bucket_offset, bucket_sizes[dest_pe], dest_pe);
  shmem_int_put_nb(&bucket_keys[dest_offset], … , dest_pe);
}
```

Loop fission

```
for (int i=0; i<shmem_n_pes(); i++) {
  int dest_pe = peers_iter(i);
  shmem_longlong_atomic_fetch_add_nbi(&bucket_offset,
         bucket_sizes[dest_pe], &dest_offsets[dest_pe], dest_pe);
}
shmem_quiet();
for (int i=0; i<shmem_n_pes(); i++) {
  int dest_pe = peers_iter(i);
  shmem_int_put_nb(&bucket_keys[dest_offsets[dest_pe]],…,dest_pe);
}
```

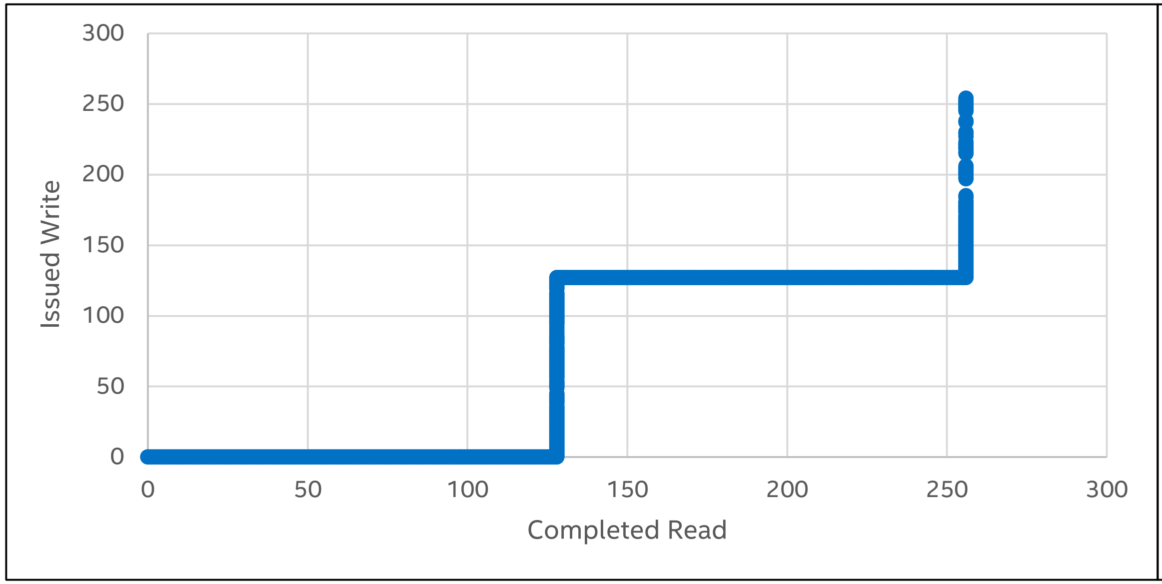
shmem_quiet() ensures the completion of all fetches

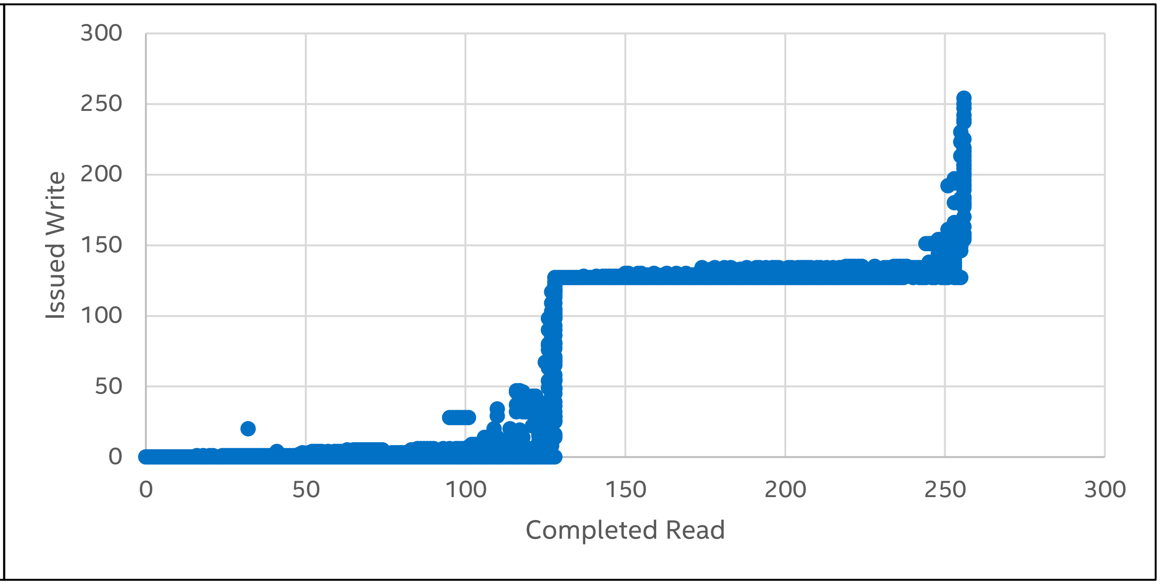
Pending Operations with Non-blocking AMO



- Both pending read and write operations increase to almost 128 (total number of PE)
- Loop fission ensures read and write operations overlap within themselves
- `shmem_quiet` ensures the completion of all fetches, but prevents any overlap between read and write

Overlap between Read and Write





- Plot issued write v/s. completed read to present any overlap between read and write
- In both iterations, write and read progress independently and thus, no overlap

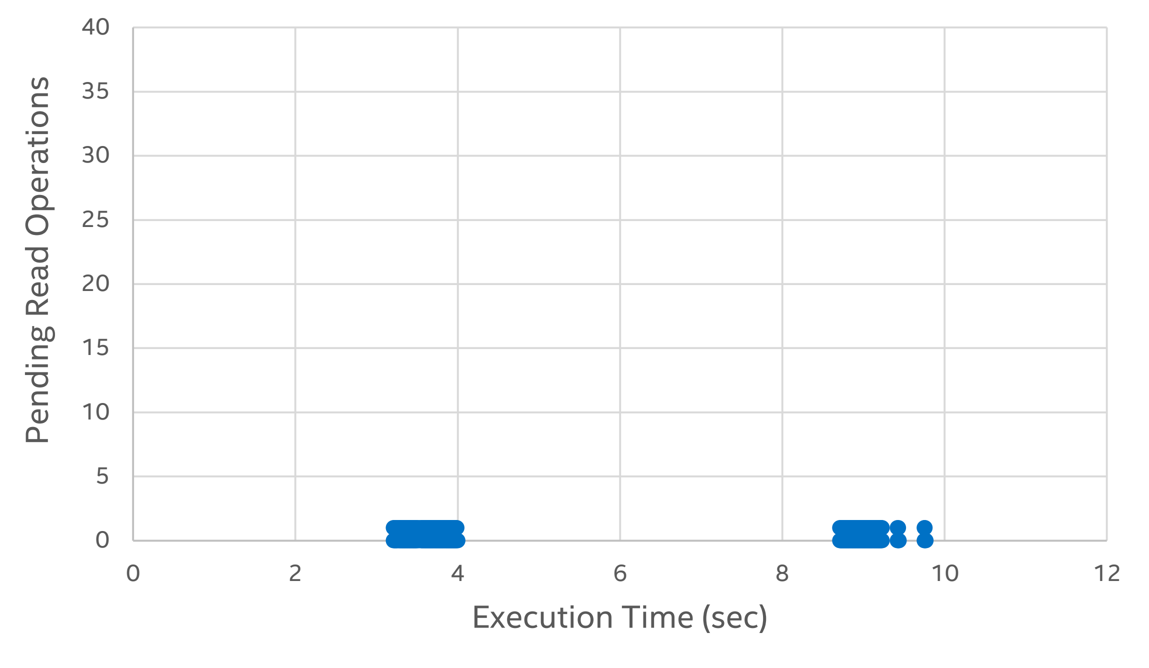
- Replace `shmem_quiet` with individual `wait_until` to wait for each non-blocking `fadd` to complete before invoking the corresponding `shmem_put`
- Both the iterations exhibit overlap between read and write with `wait_until` at the end of the loop execution

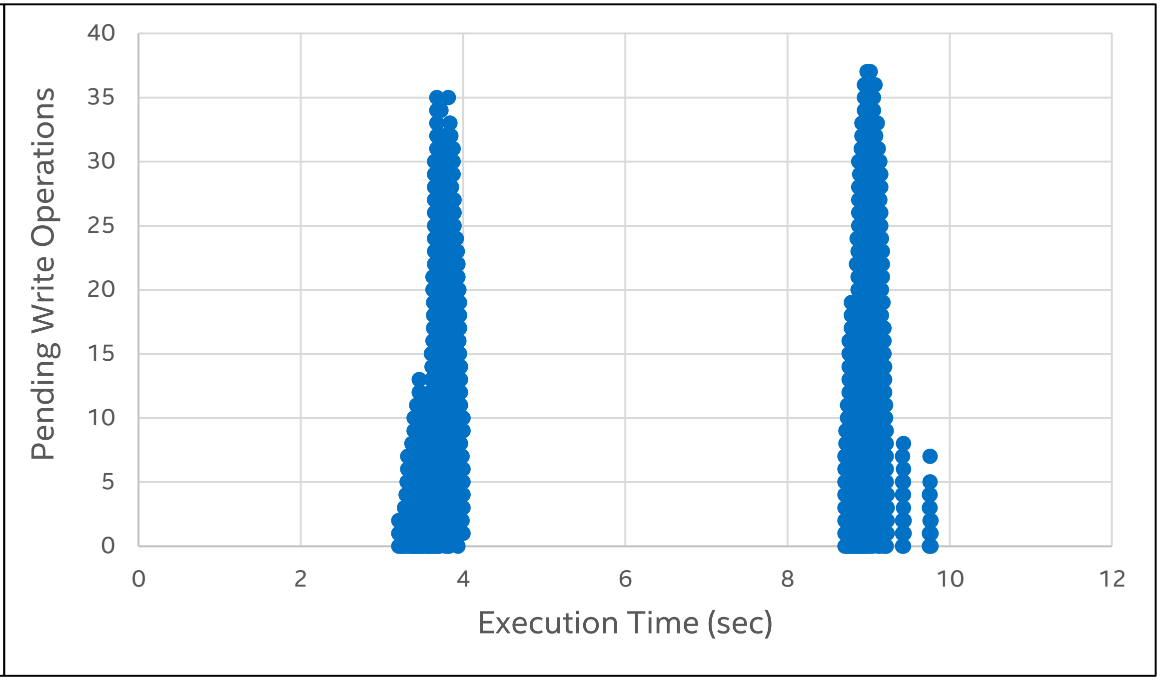
Overlap with Threads

- Alternative approach to pipeline the all-to-all exchange
  - Distribute loop iterations to multiple threads
  - Launch the threads in parallel
- Use OpenMP threads to the key-exchange routine
  - Create a pool of contexts to be used by different threads
  - Each thread utilizes it's own context to invoke SHMEM APIs
- Apply OpenMP threads on both implementations of ISx (with and without loop fission)

```
#pragma omp parallel num_threads(T) {
  int thread_id = omp_get_thread_num();
  int PEs_per_thread = shmem_n_pes()/T;
  for (int i = thread_id * PEs_per_thread; i < (thread_id + 1) * PEs_per_thread; i++) {
    int dest_pe = peers_iter(i);
    long long dest_offset = shmem_longlong_atomic_fetch_add(ctx_pool(thread_id),
      &bucket_offset, bucket_sizes[dest_pe], dest_pe);
    shmem_int_put_nb(ctx_pool[thread_id], &bucket_keys[dest_offset], … , dest_pe);
  }
}
```

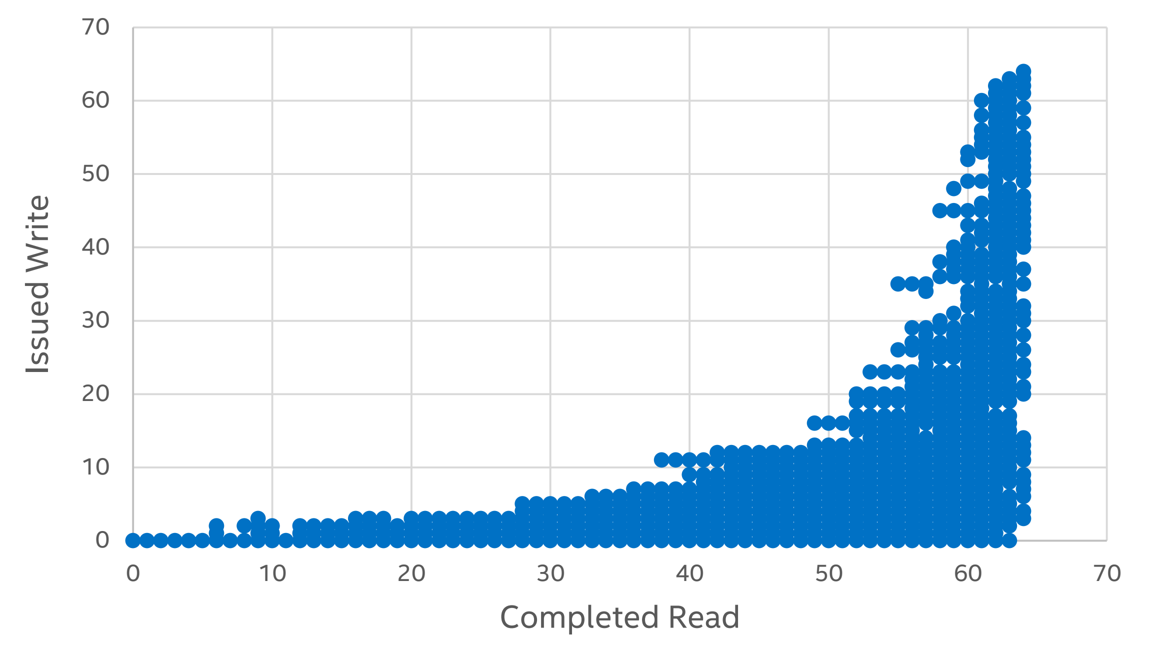
Pending Operations with Non-blocking Put and Two Threads

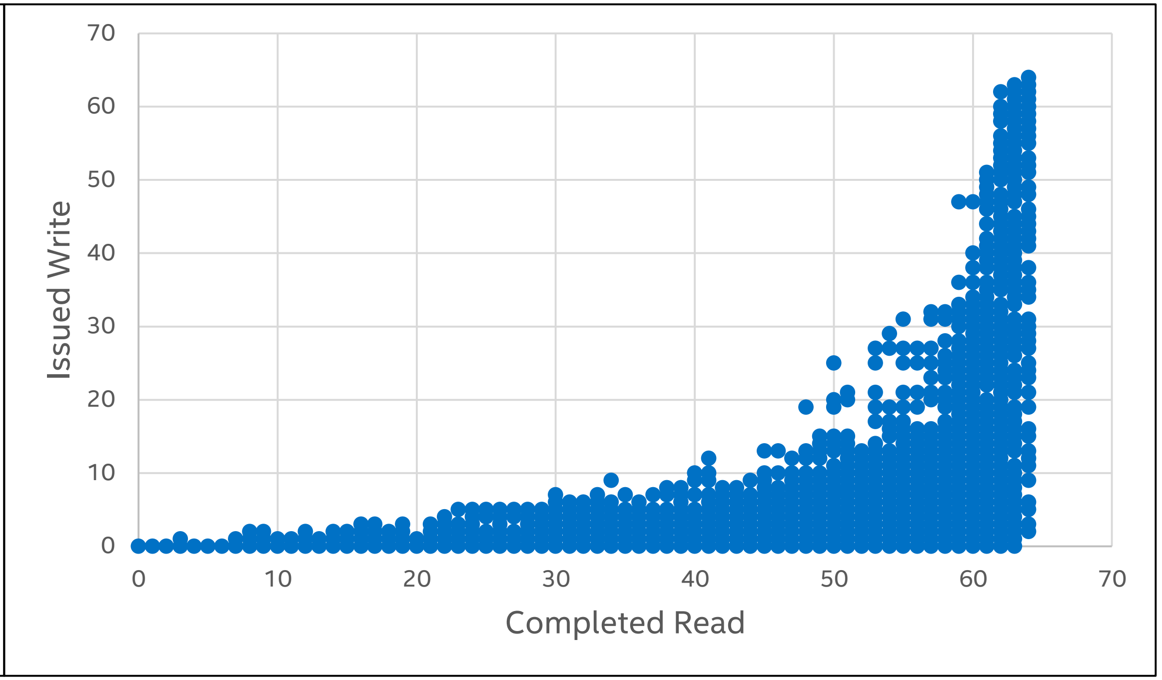




- Pending read operations are similar to the non-threaded implementation as they use the blocking API; per thread, it does not increase more than once
- Pending write operations increase more than that of non-threaded implementation; with multiple threads, overlapping among different write operations can be increased

Overlap with Non-blocking Put, AMO and Two Threads





- Apply OpenMP threads on the two distributed loops of key exchange with wait-until
- Both warm-up and trial iterations exhibit more overlap with multiple threads
- Increased pipelining between read and write operations compared to the non-threaded implementation

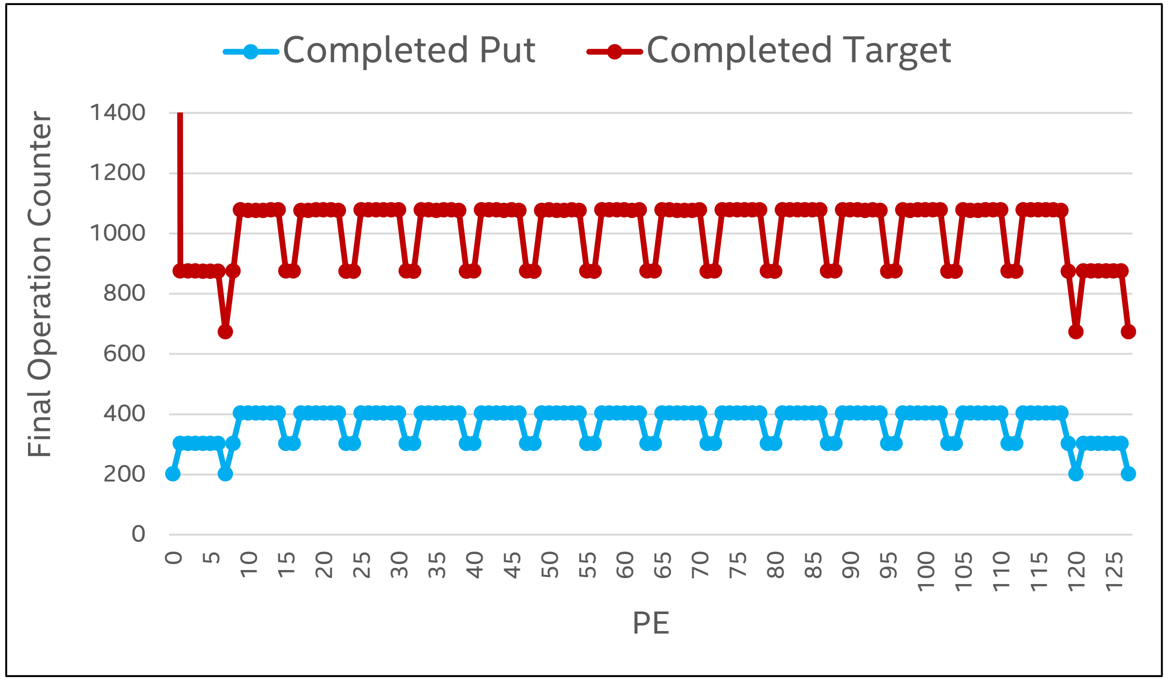
Outline

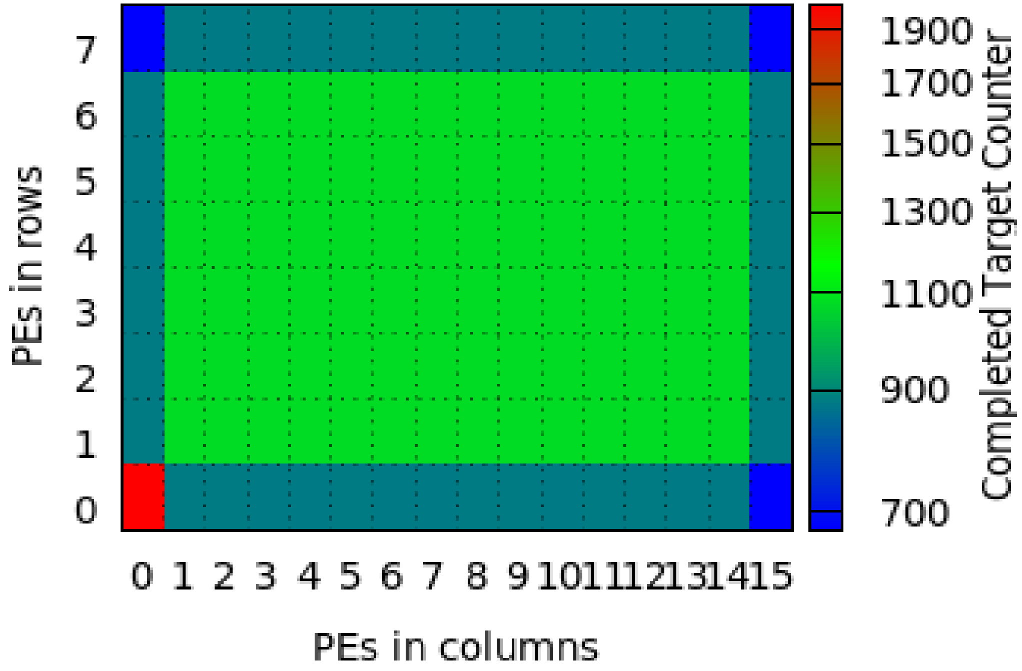
- Introduction and Motivation
- Existing Approaches
- Performance Counter APIs
- Design and Implementation of a Collector
- Experimental Analysis
  - Communication Schedule
  - Overlap
  - Load Balance
  - Weak Scale Analysis
  - Overhead
- Conclusion and Future Work

Load Balance

- Utilize performance counters to detect load balance across all PEs
- Focus on the final operation counter value
- Use Stencil kernel (128 PEs with grid size of 1000 and 100 iterations)
- Observe put and get counters as well as target counter

Load Balance





- Grid of PEs with 8 rows and 16 columns; PEs with less neighbors (edge) have less load compared to the PEs with more neighbors (inner)
- Both Put and Target counter exhibit the load imbalance for stencil
- PE0 has a high target counter value because of collective and synchronize operations

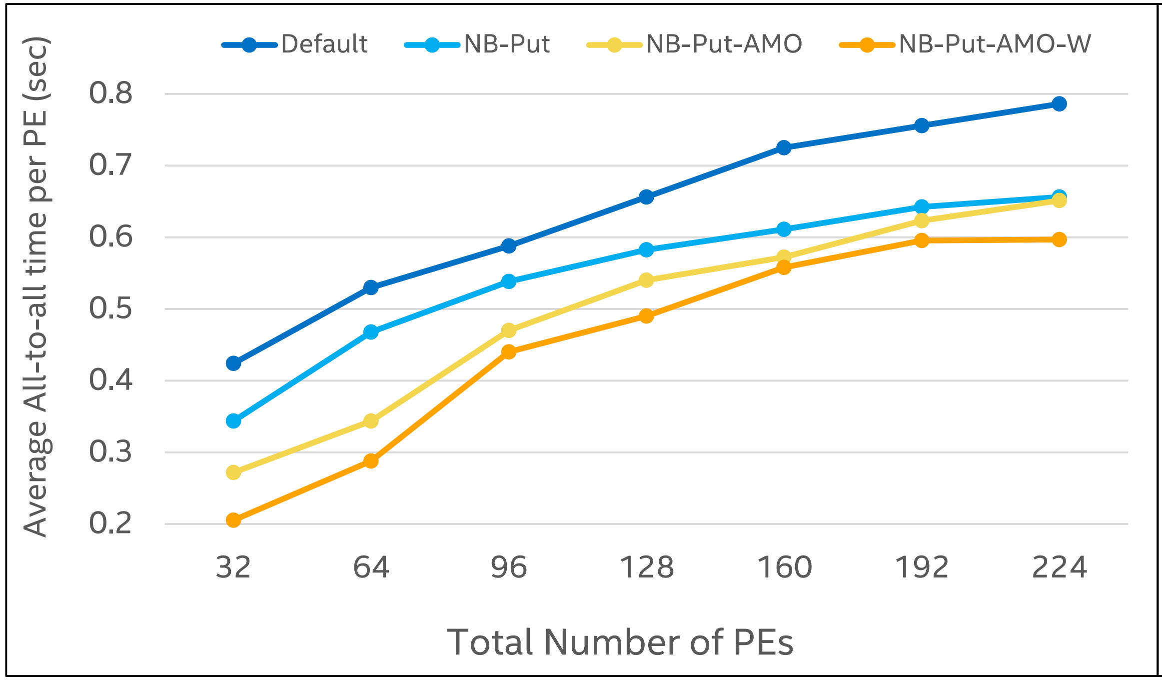
Outline

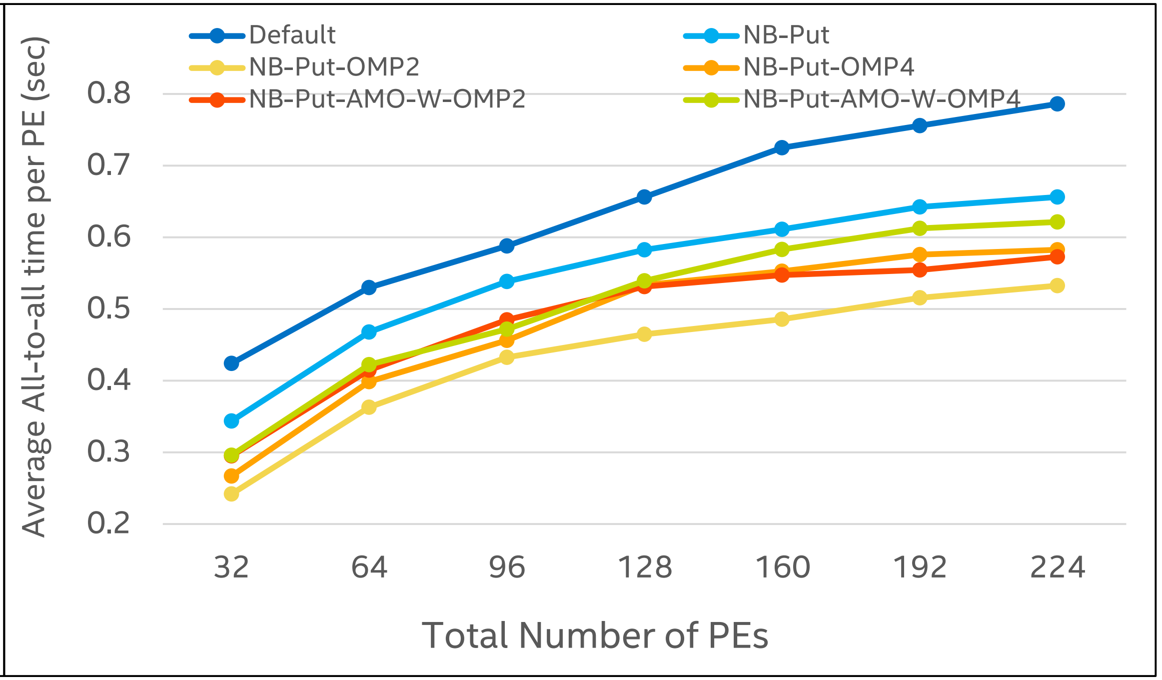
- Introduction and Motivation
- Existing Approaches
- Performance Counter APIs
- Design and Implementation of a Collector
- Experimental Analysis
  - Communication Schedule
  - Overlap
  - Load Balance
  - Weak Scale Analysis
  - Overhead
- Conclusion and Future Work

Different ISx Implementations

	ISx Implementation	Description
Non-threaded	Default	Default implementation with blocking Put and AMO
	NB-Put	Implementation with non-blocking Put
	NB-Put-AMO	Implementation with non-blocking Put and AMO in two distributed loops using `shmem_quiet` in between
	NB-Put-AMO-W	Implementation with non-blocking Put and AMO in two distributed loops using `shmem_wait_until`
Threaded	NB-Put-OMP2	Implementation with non-blocking Put and 2 OpenMP threads
	NB-Put-OMP4	Implementation with non-blocking Put and 4 OpenMP threads
	NB-Put-AMO-W-OMP2	Implementation with non-blocking Put and AMO in two distributed loops using `shmem_wait_until` and 2 OMP threads
	NB-Put-AMO-W-OMP4	Implementation with non-blocking Put and AMO in two distributed loops using `shmem_wait_until` and 4 OMP threads

Weak Scale Analysis





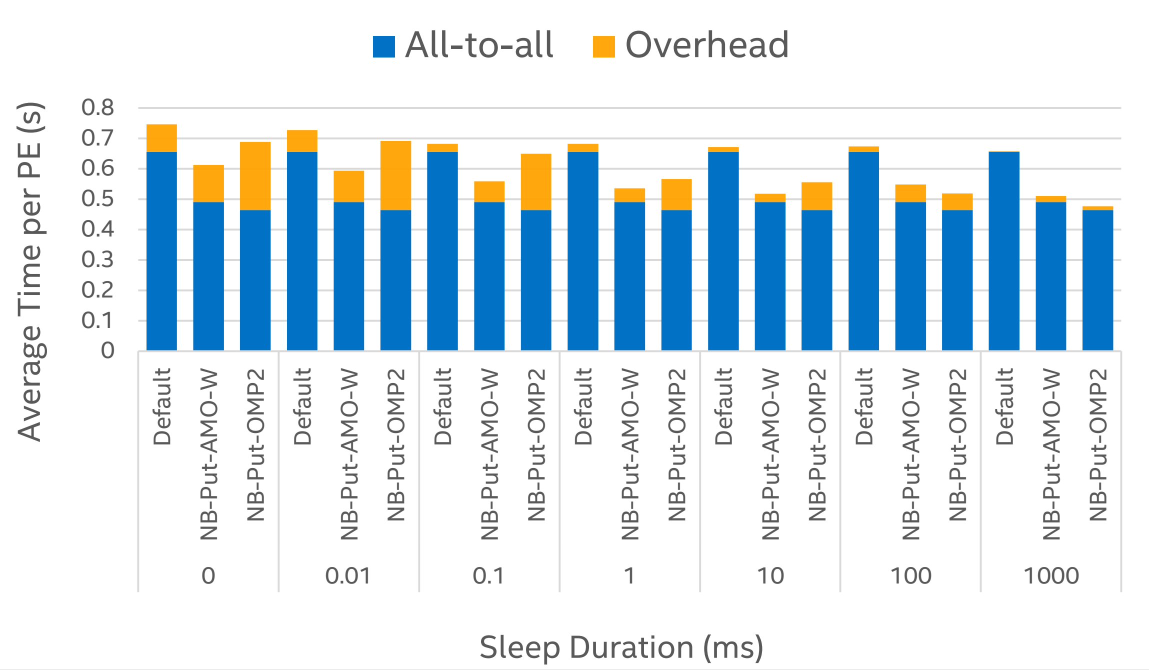
- Comparison of non-threaded implementations in 2 to 14 nodes (16 PEs/node)
- NB-Put achieves 16.5% benefit compared to the Default; NB-Put-AMO-W out-performs NB-Put-AMO by 8.3%

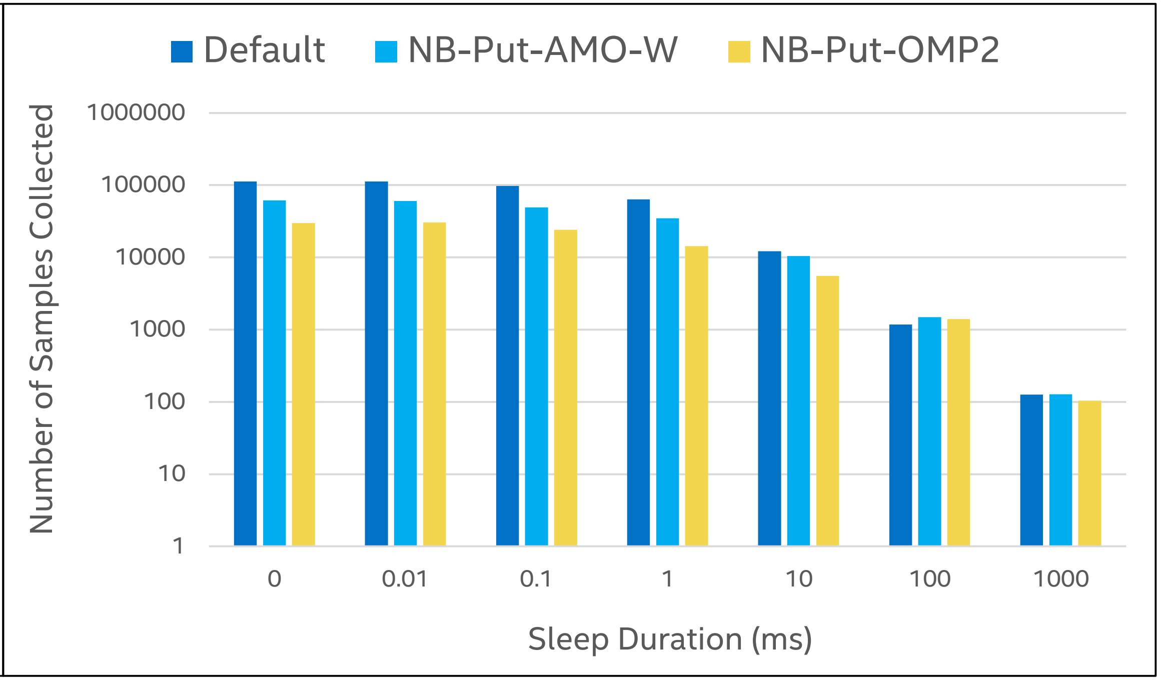
- Comparison of threaded implementations in 2 to 14 nodes (16 PEs/node) with 2, 4 threads
- NB-Put-OMP2 achieves 10% benefit compared to the single-threaded NB-Put-AMO-W; Additional threads degrade performance

Outline

- Introduction and Motivation
- Existing Approaches
- Performance Counter APIs
- Design and Implementation of a Collector
- Experimental Analysis
  - Communication Schedule
  - Overlap
  - Load Balance
  - Weak Scale Analysis
  - Overhead
- Conclusion and Future Work

Collector Overhead





- Analysis on three different implementations based on different optimization choices
- Observe 20-100 ms overhead in average all-to-all time per PE for NB-Put-AMO-W
- Additional overheads for threaded implementation, NB-Put-OMP2
- Can collect reasonable number of samples with a sleep duration of 0.1 ms

Outline

- Introduction and Motivation
- Existing Approaches
- Performance Counter APIs
- Design and Implementation of a Collector
- Experimental Analysis
- Conclusion and Future Work

Conclusion and Future Work

- Proposed a performance counter API extension to OpenSHMEM specification
- Implemented the APIs in Sandia OpenSHMEM library
- Designed and implemented a low-overhead collector to use these APIs
- Analyzed applications with the performance counters to
  - Reveal and fix implementation bug in communication scheduling
  - Characterize load balance
  - Identify opportunities to improve pipelining and overlapping deficiencies
- Proposed approaches improve the average all-to-all time for ISx by 30%

- Investigation on automated methods for analyzing the collected data
- Use performance counter APIs to aid developers of recent and proposed API extensions
- Identify system-level performance optimization opportunities

Questions?