OpenSHMEM Queues for Communication and Data Aggregation

Table of Contents

Design Goals

- Aggregation of both communication and data aggregation
- Aggregation should be agnostic of underlying mechanisms
  - Post list, UMR, SmartNIC-based aggregation, and optimizations to reduce the number of doorbells
- Aggregation should be agnostic of thread posting the operation
  - CPU, SmartNICs, or GPU
- Need to express QoS on the aggregated objects and ability to segregate traffic
  - Short Messages - aggregate and send (increase message rate); separate SL (reduce tail latency)
  - Large messages - optimize for bandwidth

OpenSHMEM Queues

The OpenSHMEM queue is either a communication or data aggregation mechanism. An aggregation queue is an opaque object. OpenSHMEM users can interact with the queue only through the OpenSHMEM queue interfaces.

There are two types of queues: communication queues and data queues. The communication queues are used for aggregating communication operations. The operations that can be queued include shmem_put_nbi, shmem_get_nbi, shmem_atomic operations, and strided communication operations. The collective operations, blocking operations, and ordering operations cannot be queued using this mechanism. The data queues enable aggregating of data before sending the data to the destination PEs. The data aggregation can be achieved by posting buffers to the queue.

After queuing, the library is free to progress the operations or complete the operations. The user can complete all outstanding operations by using a shmem_queue_flush operation or it is automatically flushed after the timeout if it is set.

Data Structures

```
typedef enum {
  SHMEM_QUEUE_EXCLUSIVE = 0,
  SHMEM_QUEUE_SHARED = 1
} shmem_queue_thread_t;

typedef enum {
        SHMEM_QUEUE_COMM = 0,
        SHMEM_QUEUE_DATA = 1
} shmem_queue_type_t;

typedef enum {
      SHMEM_PE_SAME = 0,
      SHMEM_PE_ALL = 1,
} shmem_pe_type_t;

typedef enum {
        SHMEM_OP_PUT = 0,
        SHMEM_OP_GET  = 1,
        SHMEM_OP_ATOMIC_ADD = 2,
        SHMEM_OP_ATOMIC_INC = 3,
} shmem_op_type_t;

typedef struct shmem_queue_config {
```

```
    shmem_queue_type_t qtype;
    shmem_queue_sharing_t sharing_model;

/* Exhaustive optional fields */
    uint64_t max_elems; /* applicable for comm queue */
    uint64_t max_bytes; /* applicable for data queues */
    double timeout_flush;
    shmem_op_type_t op_type;
    shmem_pe_type_t pe_type;
} shmem_queue_config_t;


typedef struct shmem_queue_attrib {
   uint64_t queue_id;
   uint64_t outstanding_elems;
   uint64_t available_elems;
}  shmem_queue_attrib_t
```

Communication: Interfaces

int shmem_queue_comm_create(shmem_queue_t *queue, shmem_queue_config_t *qconfig);
Brief Description: Create a communication queue
Semantics: ● This operation creates the queue and allocates the necessary resources for communication aggregation. On return from the operation, the Queues are ready for communication operation aggregation. ● During creation, the queue can be customized using the shmem_queue_config_t structure. ● The user is responsible for destroying the queues by calling shmem_queue_comm_destroy interface. ● The create operation returns an error if the queue cannot be created or necessary resources cannot be allocated. ● When the SHMEM_QUEUE_SHARED option is passed, the queue created can be accessed via multiple threads. ○ TBD: Should we tie it to the SHMEM_THREAD_* model?
Return values: Zero on success and nonzero otherwise

void shmem_queue_comm_destroy(shmem_queue_t queue);

Brief Description: Destroy the communication queue.

Semantics:

- Before this routine is called, all operations that are posted to the queue should be completed.
- When this routine is called, the library deallocates the resources associated with the queue. On return, no new communication operations can be posted.
- TBD: Multithread behavior

Return values: Zero on success and nonzero otherwise

int shmem_query_attrib(shmem_queue_t queue, shmem_queue_attrib_t *attr);

Brief Description: The routine queries the attributes of the queue

Semantics:

- Returns the attributes of the queue as defined shmem_queue_attrib_t

int shmem_query_size(shmem_queue_t queue, size_t *size);

Brief Description: Queries the size

Semantics:

- Returns the number of outstanding communication operations.

Return values: Zero on success and nonzero otherwise

C11:

int shmem_queue_comm_push(shmem_queue_t queue, TYPE *dest, TYPE *src, size_t nelems, int PE, uint64_t op);

Brief Description: Posts operation for the aggregation

Semantics:

- The push is a local operation on the calling PE.
- The operation is non-blocking and returns immediately after posting the operation to the queue.
- The operations posted to the queue could be completed but are not guaranteed to be completed until the flush operation is called.
- The src buffer is ignored for the atomic increment operation.
- When the number of outstanding ops is greater than max elements of the queue, the push operation returns an error.
- TBD: How to support other arguments for atomic operations, strided operations? Args?

Return values: Zero on success and nonzero otherwise

int shmem_queue_progress(shmem_queue_t queue);

Brief Description: Advances the operations posted to the queue

Semantics:

- It might progress or complete some of the communication operations.
- If no operations are outstanding it might return immediately.
- It returns the number of outstanding operations.

Implementation Notes:

The queue can be progressed either by the user calling the progress interface or asynchronously such as the library's progress thread. Though not required, an efficient implementation of queues should strive to achieve asynchronous progress.

Return values: Number of outstanding operations.

int shmem_queue_local_flush(shmem_queue_t);

Brief Description: Completes all operations posted to the queue

Semantics:

- This routine is a blocking operation.i.e., does not complete until all operations are completed.

• After the flush, all operations are completed and results are visible to all PEs. • TBD: Should we add a non-blocking version of local flush? Note: • The semantics are similar to the shmem_queue
Return values: Zero on success and nonzero otherwise

OpenSHMEM Data Queues

Data Queues: Interfaces

int shmem_queue_data_create(shmem_queue_t *queue, shmem_queue_config_t *qconfig, shmem_team_t team);
Brief Description: Create a data queue on all PEs participating in the create operation.
Semantics: • The data queue creation is a collective operation on the team, i.e., all PEs that are a part of the team should participate in the create operation • The queue options are defined in the structure shmem_queue_config_t • On success, a queue object is created. The OpenSHMEM programs can enqueue “max_bytes” of incoming buffers and “max_bytes” of outgoing buffers for each PE in the team. • Like other collective routines, this routine may not be called simultaneously by multiple threads.
Return values: Zero on success and nonzero otherwise

int shmem_queue_data_push(shmem_queue_t queue, TYPE *src_buffer, size_t nelems, int PE);
Brief Description: The routine pushes data elements to the queue
Semantics:

- The push is a local operation and does not require participation from all PEs in the team.
- The source buffer is a symmetric object.
- After this operation, the “nelems * sizeof (TYPE)” bytes of queue buffer is occupied
- The operation returns an error when the outgoing queue buffers for the target PE are full. Typically, a flush operation can empty the buffers. Also, popping the incoming queue on the target PE can empty the buffers.
- TDB: What data types to support?

Return values: Zero on success and nonzero otherwise

int shmem_queue_data_pop(shmem_queue_t queue, TYPE *dest_buffer, size_t nelems, int PE);

Brief Description: The routine moves data elements from queue to the dest buffer

Semantics:

- The pop is a local operation and does not require participation from all PEs in the team.
- The dest buffer is a symmetric object
- After this operation, the “nelems * sizeof(TYPE)” bytes of queue buffer is available for enqueuing the data.

Return values: Zero on success and nonzero otherwise

int shmem_queue_global_flush(shmem_queue_t queue);

Brief Description: The routine pushes all data from the outgoing queues to the incoming queues

Semantics:

- The flush is a collective operation .i.e., all PEs that are part of the queue when created need to be part of the flush operations
- After this operation, all outgoing buffers are available for enqueueing
- The routine will return an error if any of the PEs cannot push the data from the outgoing queues to the target PE’s incoming queues.
- TBD: Should errors indicate local status rather than global statuses?

Notes:

In OpenSHMEM programs, it is ideal to use this routine, when the outgoing buffers are full and incoming buffers are empty on all PEs.
Return values: Zero on success and nonzero otherwise

int shmem_query_data_size(shmem_queue_t queue, size_t *incoming, size_t *outgoing, int PE);
Brief Description: The routine queries and outputs the number of outgoing bytes and incoming bytes which are outstanding in the queue for a given PE.
Semantics: ● Returns the number of outstanding data bytes. ● Returns the number of incoming data bytes. ● TBD: Datatype of data elements
Return values: Zero on success and nonzero otherwise

int shmem_queue_data_progress(shmem_queue_t queue, shmemt_team_t team);
Brief Description: Progress the queues
Semantics: ● TBD: Do we need this routine?

void shmem_queue_data_destroy(shmem_queue_t *queue, shmem_team_t team);
Brief Description: The routine destroys the queue
Semantics: ● The destroy is a collective operation on the team .i.e., all PEs in the team call the team ● The user is responsible for flushing all the data from the outgoing queues and popping all data from the queue. ● Destroying the queue without emptying the queue can result in lost data elements

Optimizations:

- Postlist for comm operations:

    - Post the list of operations to the same PE using ibv_postlist
- UMR:
    - Post UMR when queues are created. Post a new UMR when queues are popped and queue becomes empty.
    - When Flush is called
- Leverage DPAs for Asynchronous progress:

Comparison with other APIs

Chained operations: The application indicates the start of the session and the end of the session.

1. Couples with thread calling the operation - semantics, and ordering.
2. Not conducive to offloading to the network hardware. i.e., hardware acceleration for aggregation.
3. Aggregation/Chaining is limited to a given thread
4. Doesn't provide an estimate on the number of operations posted (max_elems posted) and limiting optimization space
5. How to expand to collective semantics?
6. How to expand to GPU/DPU based hardware?
7. How to expand to data aggregation?

DMAPP `dmapp_bput_nbi`

- Series of puts followed by put with the flag to indicate the end of the puts
- All puts in the session should be to the same target

Other API Choices for OpenSHMEM Queues

1. Orthogonal to Contexts (the current proposal)
    a. Pros: There is no complexity of mixing context and team semantics with the queues
    b. Cons: Increase in API surface area to support a wide variety of types and operations
2. Attach Queues to OpenSHMEM contexts
    c. Pros: Does not increase the API surface area.
    d. Cons:
        i. Ties to context semantics such as private/shared
        ii. Ties to the calling thread and aggregation can be limited

Future Extensions

1. Collective Queues
    a. Initialize all queues during team Init and destroy all during destroy
        i. shmem_queue_collective_create();

ii. shmem_queue_collective_flush();
iii. shmem_queue_collective_destroy();

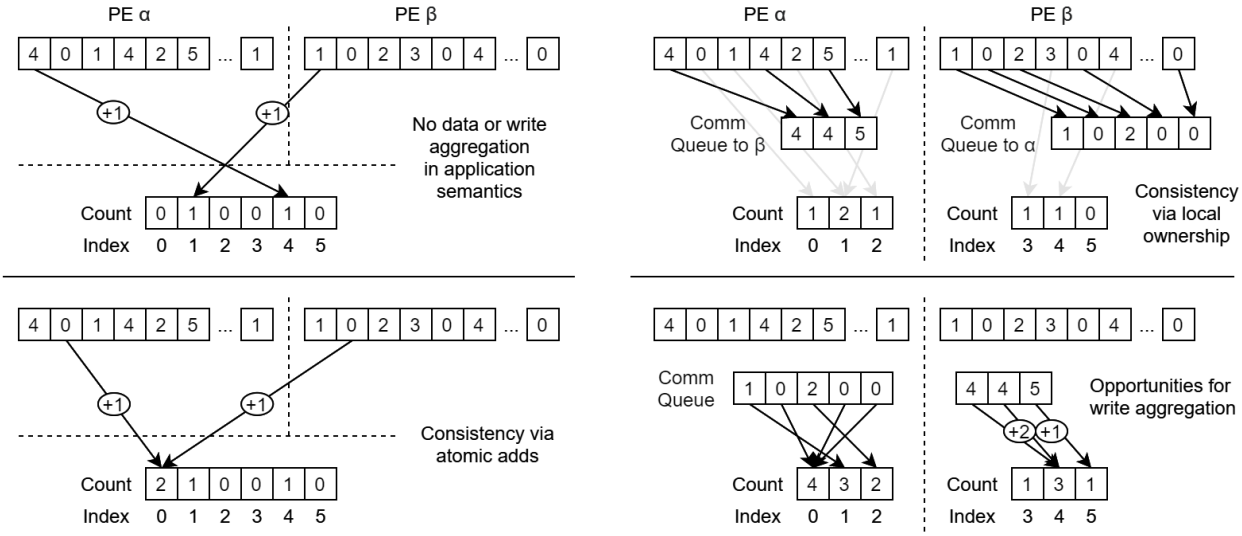
b. Add collective operations for aggregation (Motivation ?)

Examples

Histogram

```
#include "histo.h"

/*!
 * \brief This routine implements the shmem_queue variant of histogram.
 * \param data the histo_t struct that carries all the parameters for the implementations
 * \param buf_cnt defines the shmem_queue_size
 * \return average run time
 *
 */
double histo_shmem_queue(histo_t * data, const int64_t buf_cnt, shmem_team_t team) {
  int ret;
  double tm;
  int64_t pe, col, idx, *idxp;
  minavgmaxD_t stat[1];
  int64_t my_pe = shmem_my_pe();

  shmem_queue_config_t data_config = { .max_elems = buf_cnt,
          .data_elem_size = sizeof(int64_t),
          .timeout_flush = SHMEM_QUEUE_MAX_TIMEOUT,
          .op_type = SHMEM_OP_ATOMIC_ADD,
          //.PE
          .qtype = SHMEM_QUEUE_COMM,
          .thread_model = SHMEM_QUEUE_EXCLUSIVE };

  shmem_queue_t data_queue;
  shmem_queue_comm_create(&comm_queue, &comm_config, team);

  shmem_barrier_all();
  tm = wall_seconds();

  int64_t i = 0;
  int64_t one = 1;
  for(; i < data->l_num_ups; i++){
     col = data->pckindx[i] >> 20L;
     pe  = data->pckindx[i] & 0xfffff;
     assert(pe < THREADS);

     shmem_queue_comm_push(comm_queue, &(data->lcounts[col]), &one, pe, SHMEM_OP_ATOMIC_ADD);

     shmem_queue_progress(comm_queue);
  }

  shmem_queue_local_flush(comm_queue);

  shmem_barrier_all();
```

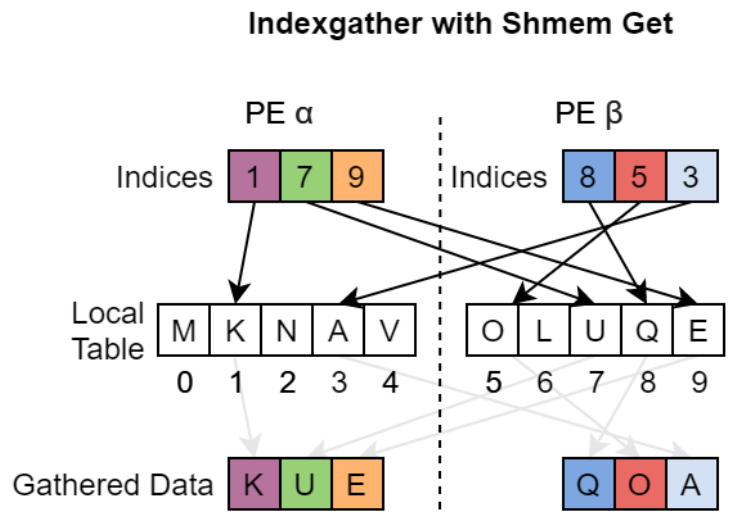
```
  tm = wall_seconds() - tm;
  lgp_min_avg_max_d( stat, tm, THREADS );

  shmem_queue_comm_destroy(comm_queue);

  return( stat->avg );
}
```



Indexgather

```
/* shmem indexgather with no queues */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

#include <shmem.h>

int
main(void)
{
   int32_t me, npes;

   shmem_init();
   me = shmem_my_pe();
   npes = shmem_n_pes();


   const uint32_t local_table_count = 10000;
   int32_t* local_table = shmem_malloc(local_table_count*sizeof(int32_t));


   for(uint32_t i = 0; i<local_table_count; i++) {
      local_table[i] = rand();
   }

   const uint32_t indicies_to_gather_count = 100;
   uint32_t indicies_to_gather[indicies_to_gather_count];

   for(uint32_t i = 0; i<indicies_to_gather_count; i++) {
```

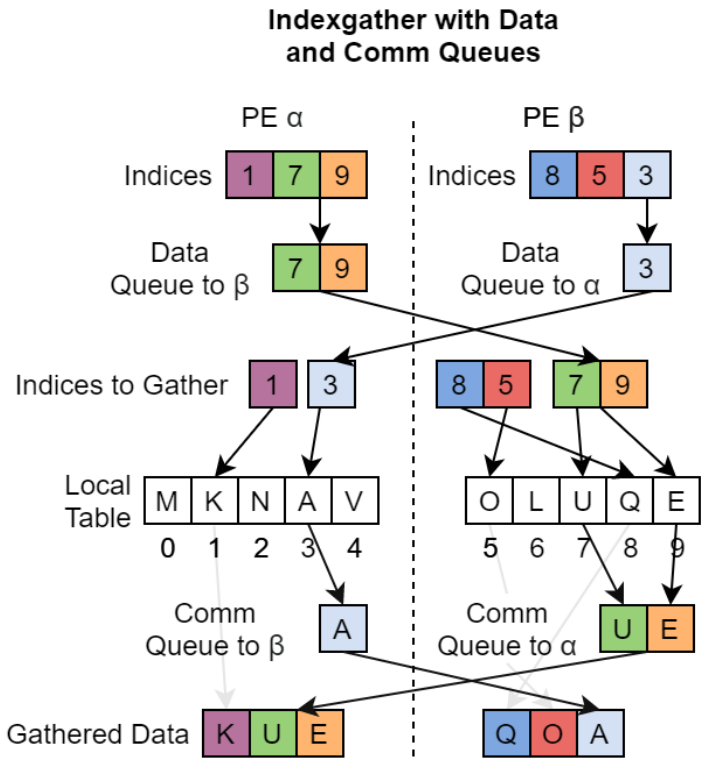
```
        indicies_to_gather[i] = rand() % (10000*npes);
    }

    const uint32_t gathered_data_count = indicies_to_gather_count;
    int32_t gathered_data[gathered_data_count];

    shmem_barrier_all();

    for(uint32_t i = 0; i<indicies_to_gather_count; i++) {
        uint32_t global_offset = indicies_to_gather[i];
        uint32_t local_offset = global_offset % local_table_count;
        uint32_t pe_requested = global_offset / local_table_count;

        shmem_get(gathered_data+i, local_table+local_offset, 1, pe_requested);
    }

    shmem_barrier_all();
    shmem_free(local_table);

    return 0;
}
```




```
/* shmem indexgather with data and comm queues */


#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

#include <shmem.h>

int
main(void)
{
    int32_t me, npes;

    shmem_init();
    me = shmem_my_pe();
    npes = shmem_n_pes();
```

```
    shmem_queue_config_t data_config = {
        .max_elems = buf_cnt,
        .data_elem_size = sizeof(int64_t),
        .timeout_flush = SHMEM_QUEUE_MAX_TIMEOUT,
        //.op_type
        //.PE
        .qtype = SHMEM_QUEUE_DATA,
        .thread_model = SHMEM_QUEUE_EXCLUSIVE
    };

    shmem_queue_t data_queue;
    shmem_queue_data_create(&data_queue, &data_config, SHMEM_TEAM_WORLD);

    shmem_queue_config_t comm_config = {
        .max_elems = buf_cnt,
        .data_elem_size = sizeof(int64_t),
        .timeout_flush = SHMEM_QUEUE_MAX_TIMEOUT,
        .op_type = SHMEM_OP_PUT,
        //.PE
        .qtype = SHMEM_QUEUE_COMM,
        .thread_model = SHMEM_QUEUE_EXCLUSIVE
    };

    shmem_queue_t comm_queue;
    shmem_queue_comm_create(&comm_queue, &comm_config);

    const uint32_t local_table_count = 10000;
    int32_t local_table[local_table_count];

    for(uint32_t i = 0; i<local_table_count; i++) {
        local_table[i] = rand();
    }

    const uint32_t indicies_to_gather_count = 100;
    uint32_t indicies_to_gather[indicies_to_gather_count];

    for(uint32_t i = 0; i<indicies_to_gather_count; i++) {
        indicies_to_gather[i] = rand() % (local_table_count*npes);
    }

    const uint32_t gathered_data_count = indicies_to_gather_count;
    int32_t* gathered_data = shmem_malloc(gathered_data_count*sizeof(int32_t));

    shmem_barrier_all();

    uint32_t i = 0;
    while(i != indicies_to_gather_count) {
        while(i < indicies_to_gather_count) {
            uint64_t packed_indices_and_return_pe = (((uint64_t)me) << 48) | ((uint64_t)(indicies_to_gather[i] % local_table_count) <<
32) | i;
            uint32_t requested_pe = indicies_to_gather[i] / local_table_count;

            if(!shmem_queue_data_push(data_queue, &packed_index_and_return_pe, 1, requested_pe)) {
                break;
            }

            shmem_queue_progress(data_queue);
            i++;
        }
```

```
        shmem_queue_progress(data_queue);

        uint64_t indices_to_lookup[indicies_to_gather_count];

        size_t data_incoming;
        size_t data_outgoing
        shmem_query_data_size(data_queue, &data_incoming, &data_outgoing, my_pe);

        shmem_queue_data_pop(data_queue, &indices_to_lookup, data_incoming, my_pe);

        for(uint64_t lookup_i = 0; lookup_i<data_incoming; lookup_i++) {
            uint64_t packed_indices_and_return_pe = indicies_to_lookup[lookup_i];
            uint32_t return_pe = (packed_indices_and_return_pe & 0xFF000000) >> 48;
            uint32_t gathered_index = (uint32_t)(local_table[(packed_indicies_and_return_pe & 0x00FF) >> 32]);
            uint32_t result_offset = packed_indicies_and_return_pe & 0xFFFF;

            shmem_queue_comm_push(comm_queue, gathered_data+result_offset, &gathered_index, 1, return_pe,
SHMEM_OP_PUT);
        }

        shmem_queue_progress(comm_queue);
    }

    shmem_queue_global_flush(data_queue);
    shmem_queue_local_flush(comm_queue);

    shmem_barrier_all();
    shmem_free(gathered_data);
    shmem_queue_data_destroy(&data_queue, SHMEM_TEAM_WORLD);
    shmem_queue_comm_destroy(&comm_queue);


    return 0;
}
```


Indexgather with Data and Comm Queues
PE α
PE β
Indices
1 7 9
Indices
8 5 3
Data Queue to β
7 9
Data Queue to α
3
Indices to Gather
1 3
8 5
7 9
Local Table
M K N A V
0 1 2 3 4
O L U Q E
5 6 7 8 9
Comm Queue to β
A
Comm Queue to α
U E
Gathered Data
K U E
Q O A