Early Experience in Supporting OpenSHMEM on HPE Slingshot NIC (Slingshot 11)

Naveen Namashivayam
Hewlett Packard Enterprise, USA
naveen.ravi@hpe.com

Bob Cernohous
Hewlett Packard Enterprise, USA

Nathan Wichmann
Hewlett Packard Enterprise, USA

Mark Pagel
Hewlett Packard Enterprise, USA

HPE Slingshot NIC (Slingshot 11) is a new HPE proprietary NIC that is planned to power the three announced US exascale systems. OpenSHMEM is a Partitioned Global Address Space (PGAS) library interface specification. Cray OpenSHMEMX is a HPE proprietary software implementation of the OpenSHMEM standards specification. In this work, we provide an overview of the features supported by Slingshot 11 NIC and our early experience in supporting Cray OpenSHMEMX on Slingshot 11 NIC. We provide high-level implementation details and detailed performance analysis in supporting different standard OpenSHMEM features using microbenchmarks and application kernels. As part of this work, we propose non-standard new implementation specific features to extract best performance from Slingshot 11 NIC.
***Index Terms*—PGAS, OpenSHMEM, RMA, AMO, One-sided Communication, HPE Slingshot, Slingshot 11**

I. Introduction

Evolutionary changes over multiple generations of High-performance Computing (HPC) and data-center networking architectures converged towards remote direct memory access (RDMA). It is a process to enable direct read and write operations on a remote process's memory without the operating system involvement. RDMA provides high-throughput and low-latency networking that are essential for programming large-scale parallel systems.

Partitioned Global Address Space (PGAS) [10] is a style of parallel programming model that employs light-weight one-sided communication primitives based on RDMA mechanisms. Languages and libraries based on PGAS style of programming are broadly split into two categories: (1) language-based models, where the PGAS features are added as an integral part of the base language, and (2) library-based models with PGAS features provided as an application programming interface (API). Parallel Fortran using Coarray (CAF) [21] and Unified Parallel C (UPC) [12] are examples for language-based PGAS models, while OpenSHMEM [13] and Global Arrays [20] are the examples for library-based PGAS models.

To achieve scalable performance on applications with highly irregular communication patterns [11] (like sparse solvers, FFT, and Integer Sort) and scale data parallel deep convolutional neural networks [24, 23, 22], PGAS is explored as an effective alternative to the message passing based programming model.

OpenSHMEM [13] is an example of PGAS library interface specification. It is a culmination of a standardization effort among many implementers and users of the SHMEM programming model [8]. Cray OpenSHMEMX [19] is a HPE proprietary software implementation of the OpenSHMEM standards specification. It is released as part of HPE Cray Programming Environment software package[3]. It is the HPE vendor supported OpenSHMEM implementation on various HPE system architectures. Specifically, it is supported on all HPE Cray EX supercomputer systems with HPE Slingshot interconnect [5, 9, 14].

HPE Slingshot interconnect consists of network switches and network interface cards (NICs) to deliver a performant and scalable network to address the most challenging HPC and Scale-Out Ethernet applications. At present, HPE Slingshot interconnect supports two types of NICs – (1) Industry Standard NIC (Slingshot 10), and (2) HPE Slingshot NIC (Slingshot 11). HPE Slingshot NIC (Slingshot 11) is a new HPE proprietary NIC that is planned to power the three announced US exascale systems is an example for network interface cards that supports RDMA efficiently.

The major contribution of this work is to introduce the new Slingshot 11 NIC specific features that impacts the performance of OpenSHMEM operations and our early experiences in evaluating these operations. To the best of our knowledge, this is the first work to evaluate the performance of OpenSHMEM programming model over Slingshot 11. The following Slingshot 11 NIC specific features are discussed with respect to OpenSHMEM programming requirements:

1) Event completion semantics;
2) Communication protocol usage;
3) Impact of memory registration;
4) Deferred work execution;
5) Bundling events; and
6) AMO reliability.

By introducing these new features available in Slingshot 11 NIC, we expose SHMEM users to different options provided by the Cray OpenSHMEMX library to effectively tune their applications on HPE Slingshot network with Slingshot 11 NIC.

Based on understanding the general OpenSHMEM use-cases and the above mentioned Slingshot 11 features, we propose different Cray OpenSHMEMX implementation specific extensions to extract best performance from Slingshot 11 NIC.

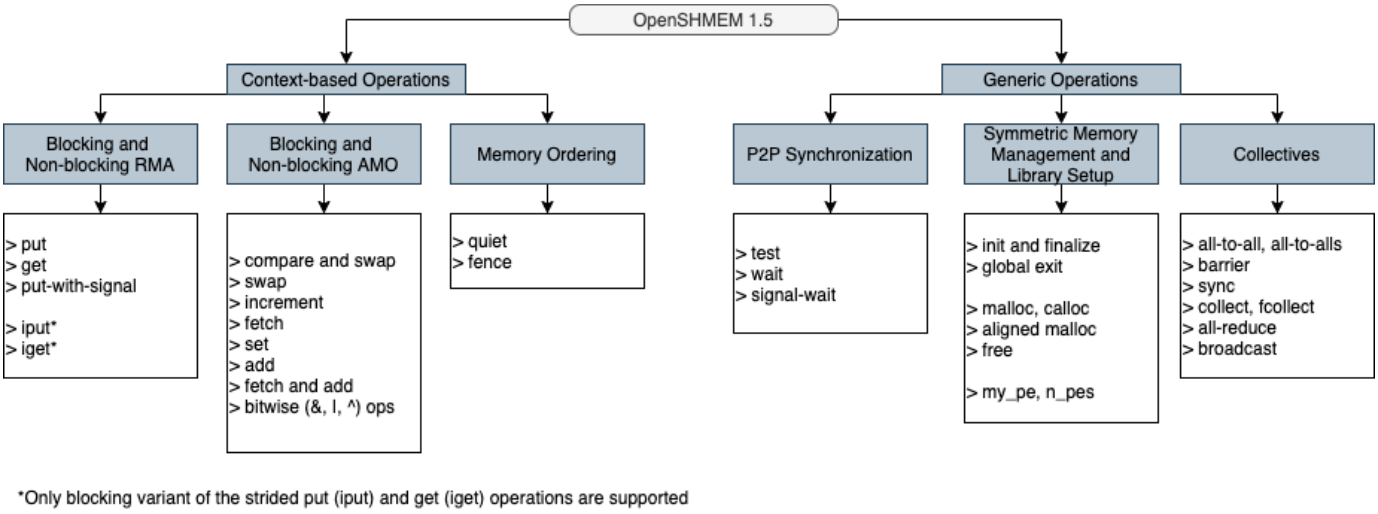

Fig. 1. Overview of features available in OpenSHMEM specification version 1.5. Operations are grouped based on Communication Contexts [15] usage.

The following are the new extensions that are specifically introduced in Cray OpenSHMEMX:

1) Cray OpenSHMEMX sessions;
2) Cray OpenSHMEMX effective signaling; and
3) Cray OpenSHMEMX fine-grain memory ordering.

These proposed new features are exposed to the users using *SHMEMX* prefixed API extensions. The proposed extensions can also benefit users and implementers of other programming models like MPI [16] to explore options for exploiting similar Slingshot 11 NIC features.

A. Contributions of this Work

The following are the major contributions of this work.

1) Use Cray OpenSHMEMX as an exemplar for evaluating the new one-sided RDMA features in Slingshot 11 NIC;
2) Introduce and report on different Slingshot 11 features impacting the performance of an OpenSHMEM application;
3) Propose different non-standard implementation specific extensions to exploit best performance from Slingshot 11 NIC; and
4) Show the performance benefits in using Slingshot 11 for communication patterns involving different irregular parallel applications.

II. Background

In this section, we provide a brief overview of different features supported in the OpenSHMEM specification [8].

A. OpenSHMEM Overview

OpenSHMEM is a library interface specification that offers support for many features enabling PGAS style of programming. The general objective of OpenSHMEM is to expose explicit control of the one-sided data transfer semantics using light-wight asynchronous RMA operations. Fig. 1 shows the rich feature set of the OpenSHMEM programming model, which includes support for blocking and non-blocking remote read and write operations, atomic memory operations, symmetric data object management, fine-grain point-to-point process synchronization, memory ordering, multithreading, and collectives. Communication contexts [15] enables creation of multiple isolated streams of operations within an application, allowing each such stream to be mapped to a separate POSIX [18] or user-threads.

1) Symmetric Data Object: OpenSHMEM programming model follows *Single Program Multiple Data* (SPMD) style of PGAS programming. It consists of multiple processes (*PEs*) sharing data among themselves. Its memory model consists of two types of data objects: (1) private, and (2) remotely accessible shared data objects. *Private data objects* are stored in the local memory of each PE and can only be accessed by the PE itself; these data objects cannot be accessed by other PEs using OpenSHMEM data movement routines. *Remotely accessible objects*, also referred as *Symmetric Data Objects (SDO)*, is accessible by remote PEs using OpenSHMEM routines. Different PEs can asynchronously read or write into the SDOs without the source PEs involvement and the SDOs are managed (allocate and released) using special OpenSHMEM memory management routines like `shmem_malloc` and `shmem_free`.

2) Remote Memory Access Operations: Asynchronous light-weight RMA data transfers (`shmem_put` and `shmem_get`) are the essential core features of PGAS models. Put and get operations are respectively those where the initiator PE writes to and reads from the target PE's remote memory without its involvement. When the RMA data transfer happens on a contiguous block of memory, either a contiguous local data object is written into or a contiguous remote data object is read from a target process.

A strided variant of the remote read and write operations are also supported by the OpenSHMEM specification using `shmem_iput` and `shmem_iget` operations.

The `shmem_put_signal` routines specific RMA operations that provide a method for copying data from a contiguous local data object to a data object on a specified PE and subsequently updating a remote flag to signal completion.

3) Atomic Memory Operations: AMOs are one-sided communication operations which combines memory read, modify, write, or update (RMW) operations with atomicity guarantees as a single operation. The supported blocking and non-blocking atomic memory operations are shown in Fig. 1.

4) Memory Ordering Operations: OpenSHMEM memory ordering routines provide mechanisms to ensure ordering and/or delivery of completion on memory store, blocking, and non-blocking OpenSHMEM routines. While `shmem_fence` operation ensures ordering of delivery of operations on symmetric data objects, `shmem_quiet` waits for completion of all outstanding operations on SDOs issued by a PE.

5) P2P Synchronization Operations: OpenSHMEM point-to-point synchronization (P2P Sync) operations provides a mechanism for synchronization between two PEs based on the value of a symmetric data object. The P2P synchronization routines can be used to portably ensure that memory access operations observe remote updates in the order enforced by the initiator PE using the put-with-signal, fence and quiet routines. `shmem_wait_until` and `shmem_test` operation wait and tests for a variable on the local PE to change respectively.

B. Cray OpenSHMEMX Overview

Cray OpenSHMEMX [19] is proprietary modular implementation of the OpenSHMEM programming model. It supports all OpenSHMEM standard specific features along with extensions specifically tuned for HPE supported architectures. Fig. 2 shows the different features supported by the Cray OpenSHMEMX implementation.

Cray OpenSHMEMX supports OpenSHMEM programming model on different HPE systems like HPE Cray XC and HPE Cray EX system architectures using its different transport modules. For example, Libfabric [17] and DMAPP [25, 6] transport module allows supporting the library on HPE Cray EX and HPE Cray XC systems with HPE-developed Aries and Slingshot 11 NICs respectively. The library also allows the usage of XPMEM [7] as an independent transport module for intra-node jobs. The *SMP* transport module as shown in Fig. 2 allows selection of a combination of different transport modules for inter-node and intra-node data movements separately.

DSMML and PMI are external libraries that are used by Cray OpenSHMEMX SW stack for its symmetric memory management and process management respectively. Namashivayam et al. provides detailed implementation overview of the Cray OpenSHMEMX SW stack.

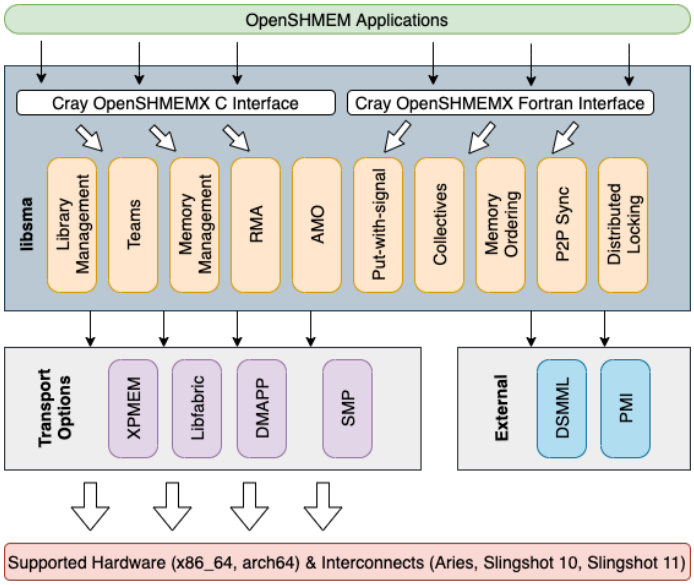

Fig. 2. Overview of different features available in Cray OpenSHMEMX.

III. Implementation

In this section, we have identified and provided detailed description of different Slingshot 11 NIC specific features that impacts the performance of OpenSHMEM operations. Slingshot 11 features discussed as part of this section are exposed to the implementers through the Slingshot 11 provider available in the Libfabric [17] library.

A. Completion semantics

In Slingshot 11, all events are expected to specify an expected completion semantics. Any AMO or RMA operation is considered as an event in Slingshot 11. In brief, Slingshot 11 supports two types of completions: (1) completion with respect to local process, and (2) completion with respect to target process.

Completion with respect to local process guarantees that the local source buffer associated with the event is ready for reuse. But, it does no guarantee that the transmitted data with respect to the event has reached the target process memory. While completion with respect to target process guarantees that the local source buffer is ready for reuse as well as the transmitted data has reached the target process memory.

Fig. 3 and Fig. 4 denotes the above mentioned completion semantics. Fig. 3 represents the completion with respect to local process, the ack for the transmitted event is generated as soon as the data from the event has reached a state on the network where it is guaranteed that it will eventually reach the target buffer, and no re-transmission from the source buffer is required. This denotes that the local source buffer is ready for reuse by the source PE.

Fig. 4 represents the completion semantics with respect to the target process. Here the ack for the event transmitted by the source PE, is shown to be generated only after the transmission is reliably completed on the target PE and the transmitted data is made available on the target buffer.

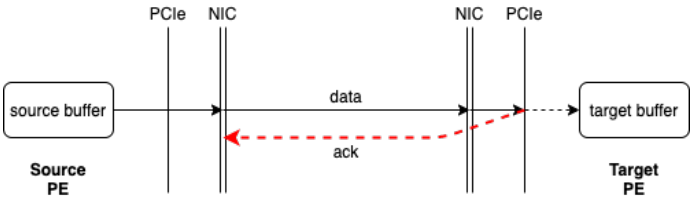

Fig. 3. Showing the completion semantics with respect to the local process. Denoting the acknowledgement (`ack`) for an event as being returned once the data on the local source buffer is transmitted and reliably guaranteed that no re-transmission would be needed. But, the ack does not guarantee the availability of the data on the target buffer.

Fig. 4. Showing the completion semantics with respect to the target process. Denoting the ack for as event as being returned only after reliably guaranteeing that the data from source buffer is transmitted and available on the target PEs target buffer.

As shown in Fig. 3 and Fig. 4, it is necessary to map the right completion semantics with respect to different OpenSHMEM operations (like blocking vs. non-blocking), and also providing fine-grain completion operations to help users in exploiting the various completion semantics supported by Slingshot 11 NIC.

B. RMA Transfer protocols

RMA transfer protocols determines how the payload is consumed by the Slingshot 11 NIC. The `inject` protocol allows packaging of both the payload along with the event header information. This allows sending both the payload as well as the event header information as part of the single *message packet*. Since, the payload is packaged as part of the message packet, it is applicable mostly for small message transfers.

The Direct Memory Access (`DMA`) protocol follows the RDMA mechanism. Here, the payload is transmitted separately as a single or multiple message packets and separated from the packet header information. The payload is split into single or multiple packets based on the size of the payload.

Selecting the right transfer protocols with respect to the data transfer size is critical in determining the performance of the OpenSHMEM RMA operations.

C. Bundled Communication

The general process involved in posting an RMA or AMO event is to prepare the message packet information (or the command queue entry(`CMDQE`)) and post it into the command queue. The Slingshot 11 NIC consumes these CMDQEs and execute them in FIFO order based on the packet information provided.

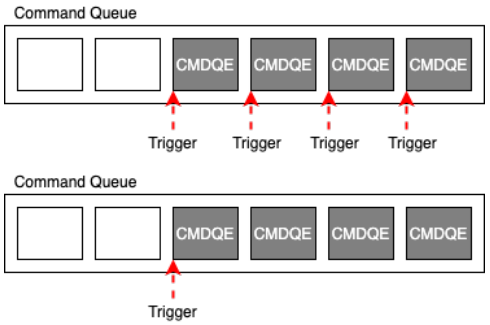

Fig. 5. Showing the available options to bundle communication events in Slingshot 11. Every event enqueued to the command queue can be triggered for execution immediately or bundled together with a single trigger for a group of events.

The Slingshot 11 NIC picks up the CMDQEs only when it observes a trigger entry in the command queue. There are two possible options to generate these triggers: (1) create a trigger per CMDQE, or (2) create a trigger for a group of CMDQEs. Fig. 5 shows two different command queues where the CMDQEs are triggered per event and triggered as a group.

The trigger event has a non-negligible performance impact. Hence, it is essential to group multiple events together and expose this bundling option to the OpenSHMEM users.

D. Memory Mapping

Each DMA protocol based events, as mentioned in Section III-B, requires three verification steps on the source PE: (1) verify the source buffer used for the DMA event is registered with Slingshot 11 NIC, (2) verify the source buffer is ready for data transfer, and (3) verify the source buffer involved in the DMA event is ready for reuse. Every DMA event performs all these three verification steps.

As soon as the DMA request is generated by the source PE, the DMA transfer engine checks whether the source buffer involved in the data transfer is registered with the Slingshot 11 NIC. If the memory is not registered, the memory registration is performed and registration details are cached for later use.

It is the users responsibility to make sure that the source buffer is updated with the required payload information before initiating the trigger event for the data transfer operation, as mentioned in Section III-C.

As mentioned in Section III-A the ack's for every event is consumed through some completion monitoring mechanism (counter[1] or completion queue[2] updates) to determine its re-usability after the data transfer operation is determined to be complete.

The performance factor involved in the memory registration is non-negligible. Hence, it is essential for the users to be aware of these knobs before generating OpenSHMEM RMA operations with unregistered source buffers from the local stack or heap memory instead of the symmetric data objects (SDOs).

	Local completion	Remote completion	Event ordering
`shmem_put` (blocking PUT operations)	return from operation	use `shmem_quiet`	use `shmem_fence`
`shmem_put_nbi` (non-blocking put operations)	Not-available	use `shmem_quiet`	use `shmem_fence`

TABLE I: Table showing the options for local and remote completion semantics using blocking and non-blocking put operations.

E. Deferred Communication

Slingshot 11 NIC supports triggered communication operations with a deferred execution semantics. It allows enqueuing communication events but deferring its execution until a specific condition is met.

Each triggered operation is created with a trigger and completion counter along with a trigger threshold value. The enqueued triggered operations are deferred execution until the trigger counter reaches the trigger threshold value mentioned in the operation.

It is essential to exploit using the triggered operation support available in Slingshot 11 in OpenSHMEM to optimize OpenSHMEM operations like the non-blocking put-with-signal.

F. AMO Reliability

Slingshot 11 supports reliable and single-transmit atomic memory operation. While a reliable AMO allows retrying atomic operations until it is successfully completed, a single-transmit AMO as its name suggests would allow transmitting the atomic operation once and expect it to successfully complete without any re-transmit requirements.

In general, the performance of a single-transmit AMOs are expected to be better than reliable AMOs. But, certain use-cases can still get benefited in using the single-transmit AMOs. While the default OpenSHMEM AMOs are expected to be reliable, we should explore extending the standard AMOs with single-transmit semantics.

IV. Cray OpenSHMEMX Extensions

Most Slingshot 11 NIC specific features discussed in Section III are exploited using standard OpenSHMEM RMA and AMO operations. In this section, we are introducing new Cray OpenSHMEMX specific extensions to allow exploiting some of Slingshot 11 features.

A. Cray OpenSHMEMX Local Completion

The current OpenSHMEM memory ordering operations (`shmem_quiet` and `shmem_fence`) ensure ordering and/or delivery of completion on memory store, blocking, and non-blocking OpenSHMEM routines. `shmem_fence` ensures ordering of delivery of operations on symmetric data objects, `shmem_quiet` waits for completion of all outstanding operations on SDOs issued by a PE.

As specified in Section III-A, the completion semantics are critical in achieving the best performance from Slingshot 11 NIC. TABLE I shows a table of options available to achieve the local and remote completion semantics required by the OpenSHMEM blocking and non-blocking variant of the put operations.

As shown in TABLE I, to effectively get benefitted from completion with respect to local process as mentioned in Section III-A, the current OpenSHMEM specification has no support for the local completion of non-blocking operations. Users are either forced to perform blocking operations or use the non-blocking operations with a heavy remote completion semantics using `shmem_quiet`.

```
void shmemx_local_complete(void);
void shmemx_ctx_local_complete(shmem_ctx_t ctx);
```

Fig. 6. Function Prototypes for Local Completion operations.

Fig. 6 provides the function prototype for introducing local complete operation in Cray OpenSHMEMX. The simple semantics of `shmemx_local_complete` operation ensures reusability of source buffers from all outstanding non-blocking operations previously issued by a PE. It is light-weight on performance when compared to `shmem_quiet`, that ensures the delivery of operations (remote completion) on the target PE.

`shmemx_local_complete` is specifically introduced to provide users an option to exploit the local completion semantics of Slingshot 11 NIC.

B. Cray OpenSHMEMX Sessions

A session in OpenSHMEM is considered an epoch in an application where the users are allowed to provide certain hints to the OpenSHMEM runtime on the application usage model. These hints provided by the users would allow the OpenSHMEM runtime, like Cray OpenSHMEMX, to manage and provide a better NIC resource utilization.

OpenSHMEM sessions can be considered similar to the *#pragma* directives in `C language`, that provides additional information to the compiler beyond what is conveyed in the language itself. Here the session options can be considered as hints that provides additional information to the OpenSHMEM runtime beyond what is conveyed by the OpenSHMEM routines used by the applications.

```
void shmemx_ctx_session_start(
                IN shmem_ctx_t ctx,
                IN int options);
void shmemx_ctx_session_stop(
                IN shmem_ctx_t ctx);
```

Fig. 7. Function Prototypes to Start and Stop an OpenSHMEM Session.

Fig. 7 shows the function prototype for starting and stopping a session on an OpenSHMEM context object. The *options* argument is a set of features passed as a hint to a given *ctx*.

These options can be anything ranging from hints to allow bundling operations or using single-transmit AMOs. These options in Cray OpenSHMEMX are specific to exploiting Slingshot 11 NIC features. Detailed examples about the various options supported by Cray OpenSHMEMX are available in Sections IV-B1, IV-B3, and IV-B2.

1) Session Bundling: Example shown in Fig. 8 provides an apt usage model where hints for bundling non-blocking put operations can be added. The code provided in this example is a common all-to-all usecase, where every PE sends a part of their data to all other PEs.

```
for (int i = 0; i < n; i++) {
    shmem_put_nbi(SHMEM_CTX_DEFAULT,
                  src + off[i], dst + off[i],
                  nelems, i);
}
shmem_quiet();
```

Fig. 8. Example program showing the use-case for bundling operations.

In the example shown in Fig. 8, the additional information that is unknown to the OpenSHMEM runtime is that the completion of all these non-blocking put operations are not expected until the memory ordering operation (`shmem_quiet`) at the end of the epoch. Also, the OpenSHMEM runtime is unaware that there are multiple non-blocking operations that is planned to be created before calling the memory ordering operation at the end of the epoch. Without these two information, the OpenSHMEM runtime would be forced to trigger all the non-blocking put operations separately and this avoids bundling these oeprations together.

```
shmemx_ctx_session_start(
  SHMEM_CTX_DEFAULT,
  SHMEM_SESSION_BUNDLE | SHMEM_SESSION_OP_PUT);

for (int i = 0; i < n; i++) {
    shmem_put_nbi(SHMEM_CTX_DEFAULT,
                  src + off[i], dst + off[i],
                  nelems, i);
}

shmemx_ctx_session_stop(SHMEM_CTX_DEFAULT);

shmem_quiet();
```

Fig. 9. Extending example program from Fig. 8 with Bundled sessions

Fig. 9, extends the example shown in Fig. 8. Fig. 9 excapsulates the example in Fig. 8 with session start and stop operations. As part of the options argument, we use *SHMEM_SESSION_BUNDLE* and *SHMEM_SESSION_OP_PUT*, hinting that the implementation is free to bundle all put operations within that session.

As the session options are an hint, it is upto the OpenSHMEM runtime to make use of the hints to optimize the data transfer with any implementation specific features. In this example, Cray OpenSHMEMX chooses to optimize the data transfer by bundling the non-blocking put operations.

2) Session Bundled PWS: As mentioned in Section III-E, triggered communication operations with deferred execution semantics can be used to efficiently implement the non-blocking put-with-signal operations. It is possible to enqueue the payload and signal data transfers in the put-with-signal operation in such a way that the completion of the payload triggers the execution of the signal operation. This design will make use of a Slingshot 11 NIC HW counter per put-with-signal operation. Needless to say, these NIC HW counter resources are scarse.

Fig. 10 shows an example program for bundling multiple OpenSHMEM non-blocking put-with-signal operations using the *SHMEM_SESSION_BUNDLE* and *SHMEM_SESSION_OP_PUT* session hints.

```
shmemx_ctx_session_start(
  SHMEM_CTX_DEFAULT,
  SHMEM_SESSION_BUNDLE | SHMEM_SESSION_OP_PWS);

for (int i = 0; i < n; i++) {
    shmem_put_signal_nbi(SHMEM_CTX_DEFAULT,
                  src + off[i], dst + off[i],
                  nelems, &signal, sig_val,
                  SHMEM_SIGNAL_SET, i);
}

shmemx_ctx_session_stop(SHMEM_CTX_DEFAULT);

shmem_quiet();
```

Fig. 10. Extending the example program from Fig. 8 using Bundled sessions with non-blocking put-with-signal operations.

The benefit for bundling the put-with-signal operations includes using the event triggering optimizations as mentioned in Section III-C. Apart from this optimization, it is also possible to group the put-with-signal operations in such a way that when using triggered operations, multiple operations are grouped to use a single NIC HW counter.

Fig. 11 shows an example transformation of the code shown in Fig. 10. In this example transformation code shown in Fig. 11, the implementation is free to convert multiple put-with-signal operations internally into groups of payload puts followed by a fence and a signal transfer to all the targets. This transformation will allow the implementation to use a single counter for all the grouped payload operations. If used correctly, the put-with-signal session option combined with the effective session bundling option can provide better performance and effective non-blocking semantics for the users.

3) Session Single Transmit AMOs: The single-transmit AMOs are similar to other session options. Using the *SHMEM_SESSION_UNRELIABLE* and *SHMEM_SESSION_OP_AMO* would convert all AMOs inside a session epoch to single-transmit AMOs.

```
/* bundling all payloads */
for (int i = 0; i < n; i++) {
    shmem_put_nbi(SHMEM_CTX_DEFAULT,
                  src + off[i], dst + off[i],
                  nelems, i);
}
/* order delivery of all signal operations
   with respect to payloads */
shmem_fence();
for (int i = 0; i < n; i++) {
    shmemx_signal_set(&sig, sig_val, i);
}
shmem_quiet();
```

Fig. 11. Showing the transformation of bundling non-blocking put-with-signal operations to bundle all payload transfers before posting the bundle of the signal transfers corresponding to all the payload transfers.

C. Cray OpenSHMEMX Signal Set

There are two types of signaling operations: *SHMEM_SIGNAL_SET*, and *SHMEM_SIGNAL_ADD*. The atomicity of the signaling operations is unique in that it is atomic only with respect to itself and any other signaling operation of the same type. The main performance benefit in using the signaling operation for point-to-point synchronization(Section II-A5, when compared to other atomic operations is that signaling semantic allows the operation to be not atomic with respect to other OpenSHMEM atomic operations, inturn allowing the OpenSHMEM runtime to effectively implement it.

The current OpenSHMEM put-with-signal semantics does not allow passing the signaling operations separetely. To make use of the signaling operation, it has to be used in conjunction with a zero-size payload put-with-signal operation.

In Cray OpenSHMEMX, the signal set operation (`shmem_signal_set`) is a new extension to provide the signaling semantics without the payload usage. The function prototype of the signaling operation is provided in Fig. 12. Using the signaling set operation against using atomic set operation (`shmem_atomic_set`) or zero-sized payload put-with-signal operation shows performance benefits.

```
    void shmemx_signal_set(
        IN uint64_t *sig,
        IN uint64_t sig_val,
        IN int pe);
    shmemx_signal_set(&sig, sig_val, i);
```

Fig. 12. Function prototype and example usage of Signaling set operation.

V. Performance Analysis

In this section, we provide detailed performance analysis of different OpenSHMEM features supported by the Cray OpenSHMEMX SW stack on the Slingshot 11 NIC. We provide details of the different Slingshot 11 features impacting the performance of the OpenSHMEM operations.

A. Test System Overview

Performance analysis reported in this section used the latest Cray OpenSHMEMX SW package released as part of the latest HPE Cray Programming model SW stack. We used an internal HPE Cray EX system equipped with multiple AMD EPYC 7763 processors code named Milan. Each node consists of dual-socket AMD Milan processors and the nodes are connected using HPE Slingshot NIC (Slingshot 11) with one Slingshot 11 NIC per socket. All AMD Milan processors are configured to use 4 NUMA nodes per socket (NPS-4).

B. Test Suite Overview

For the performance analysis, we used to two different test suites: (1) OSU Microbenchmark (OMB) [4] suite, and (2) an internal HPE developed random access (HRA) suite. While the different latency, bandwidth, and message rate tests for various OpenSHMEM operations in the OMB suite are well known, the internal HRA suite performs a random access all-to-all communication pattern. There are two user inputs required in the HRA suite: (1)*size* and (2)*n-updates*. *size* determines the size of a remotely accessible table created, and *n-updates* determines the number of updates performed by a single PE. For each update, the source PE determines a random location within a table created and a random target PE to perform different kinds of update. The type of supported updates includes different types of RMA, put-with-signal, and atomic memory operations.

The results of these updates from these two microbenchmarks are described in the following sections. And, these tests refer to the most commonly used usage models for the applications based on the OpenSHMEM programming model.

C. Impact of Transfer Protocol Selection

Results from running PUT-based tests in HRA suite on 16 nodes with 128 PPN is shown in Fig. 13. We can understand the impact of the right transfer protocol selection from these tests. As mentioned earlier in Section III-B, there are two types of transfer protocol available in Slingshot 11 NIC for data transfer operation.

It is shown in Fig. 13 that using inject and DMA protocol for small and large message sizes provides the best bandwidth close to the Slingshot 11 theoretical max. But for medium sized messages from 256 bytes to 1K it is not efficient to directly shift from using inject protocol to the DMA protocol. The performance of using DMA protocol for the medium sized messages is not good. To accommodate a smooth transition from the inject to DMA protocol selection, Cray OpenSHMEMX implementation makes use of a SW protocol selection optimization module. The impact of this SW optimization module is shown in Fig. 13.

Also, the impact of using unregistered memory for data transfer is shown in Fig. 13. `PUT_NB_lstack` and `PUT_NB_lheap` shows the usage of local stack and heap variables as source buffers for the data transfer operation. Local stack and heap variables are unregistered memory

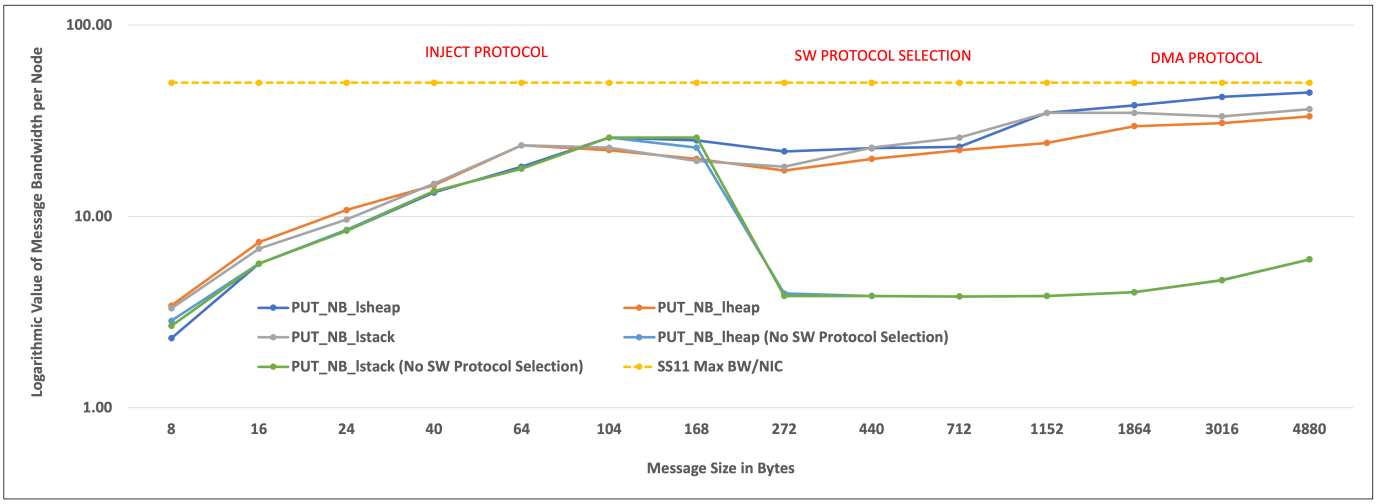

Fig. 13. HRA Random Access Non-blocking PUT Bandwidth Analysis on 16 nodes with 128 PEs per Node. Showing the performance impact of using different transport protocols as mentioned in Section III-B and registered memory for optimized data movement operations.

region. As soon the inject protocol is not used, without the SW optimization layer in Cray OpenSHMEMX the performance of using local stack and heap variables are hit. This is shown by the sudden drop in performance for `PUT_NB_lstack (No SW Protocol Selection)` and `PUT_NB_lheap (No SW Protocol Selection)` runs in Fig. 13.

In brief, through this performance analysis, we show the benefits of using the optimized SW transfer protocol selection module in Cray OpenSHMEMX runtime. Also we show the need for using the registered memory from the SDOs for optimized performance across different data sizes.

D. Impact of Completion Semantics

Fig. 14 shows the impact of Slingshot 11 NIC completion selection semantics. We used OSU blocking and non-blocking PUT microbenchmarks to measure the impact of local and remote completion semantics. The `No Completion` tests denote the usage of memory ordering operation at the end of n-iterations, while the tests with local or remote completion shows the usage of either `shmemx_local_complete` or `shmem_quiet` per operation.

We can see that tracking the completion of the non-blocking PUTs per operation (`PUT_NBI Remote Completion`) performs the worst when compared to all other types of completion tracking. This is because, we are trying to serialize the data transfer operations. Meaning, we post a non-blocking PUT on the source PE and wait for it to reach the remote target memory before posting the next operation.

Consider the run (`PUT_NBI No Completion` and `Blocking PUT No Completion`) that tracks the completion of all operations at the end of n iterations. Both the blocking and non-blocking PUT performance are similar for small messages less than 512 bytes. This is due to the usage of `inject` protocol as mentioned in Section V-C. For large messages, since the non-blocking PUTs when not tracked can generate multiple parallel transmission of data, it performs

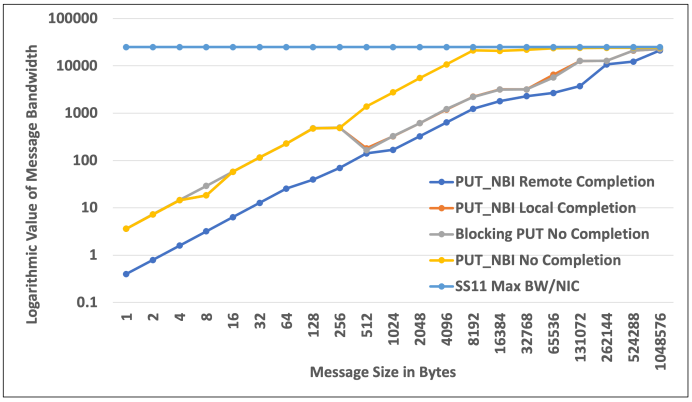

Fig. 14. OSU Blocking and Non-blocking PUT Bandwidth Analysis on 2 nodes with 1 PE per Node. Showing the performance impact of local completion for source buffer re-usability.

better than the blocking PUT operation. And, it can be seen that the non-blocking PUT `PUT_NBI No Completion` can be used to achieve the maximum bandwidth when compared to other runs.

The benefit of using local completion tracking is shown clearly by the `PUT_NBI Local Completion` results. We can see that, it performs very closely to the `Blocking PUT No Completion`. When used correctly on use-cases where the local completion of non-blocking operations is sufficient, the `shmemx_local_complete` extension can be used to achieve better performance.

E. Impact of Memory Registration

Fig. 15 is an extension to the registered memory usage analysis performed in Section V-C. We further analysis the usage of registered memory for data movement operations. This tests shows the performance of OSU non-blocking PUT microbenchmark using global and symmetric memory for the

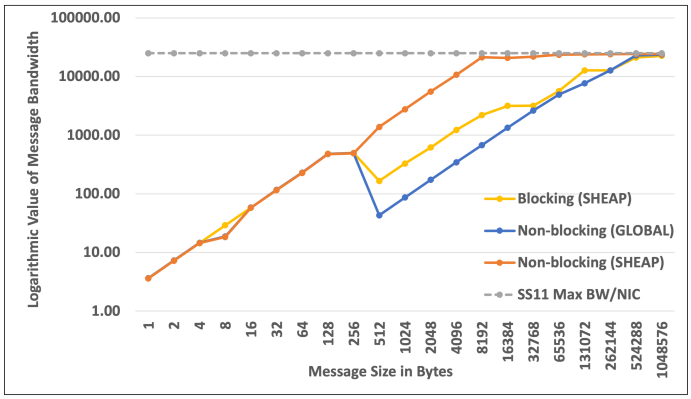

Fig. 15. OSU Blocking and Non-blocking PUT Bandwidth Analysis on 2 nodes with 1 PE per Node. Showing the performance impact of using hugepage backed memory for data movement operations.

data movement. We can see the benefit in using the symmetric data object when compared to global memory on the medium sized messages using the DMA protocol.

This analysis again shows the benefit in using the registered symmetric memory (SDO) for the data movement operations against the usage of global/static memory.

F. Impact of Event Bundling

In this section, we show the performance impact of bundling communication operations with the new proposed OpenSHMEM session extension. Fig. 16 shows the message rate comparison on non-blocking PUTs using OSU microbenchmark with and without bundling the operations across a range of tests with different process per node (PPN) usage. Example program in Fig. 9 shows the bundled operation used for this evaluation.

As seen from Fig. 16, we can understand that when smaller number of PEs are used per node, bundling of operations has a huge performance benefit (close to 2.5X improvement on 16 PPN case) when compared to applications where multiple PEs per node are used. And, this performance benefit is observed mostly for smaller message sizes less than 8K bytes.

In brief, the bundling extensions added in Cray OpenSHMEMX through the OpenSHMEM sessions proposal provides a better performance benefit for smaller messages when smaller number of PEs are involved in the communication operation per node.

G. Impact of Put-with-Signal Communication

This section shows the benefits in using the put-with-signal (`shmem_put_signal`) operations available in the OpenSHMEM specification when compared to manually setting up the put-with-signal semantics using the (`shmem_put_nbi`, `shmem_fence`, and `shmem_atomic_set`) operations.

As shown in Fig.17, we used the put-with-signal tests in HRA suite on 16 nodes with 128 PPN. Using the standard put-with-signal operation shows on average a 1.6X performance improvement over using the manual implementation of the put-with-signal semantics. This test shows the benefits of using the standard put-with-signal operations when compared to manually implementing the put-with-signal semantics in the application.

H. Impact of AMO Reliability

Fig. 18 shows the performance results of running different fetching and non-fetching AMO tests using the HRA suite on 8 nodes with 128 PPN. It shows the performance benefits in using single-transmit AMOs as mentioned in Section IV-B3 and the performance impact of the `shmemx_signal_set` atomic operation.

In general, the performance of non-fetching AMOs are much better than the the fetching AMOs. This is seen in Fig. 18 by comparing the performance of `shmem_int_add` against `shmem_int_fadd`. The single-transmit AMO usage using the session extension shows a 1.4X performance improvement over the default AMOs. This performance improvement is observed only on the non-fetching atomic operations. This is a known behavior as the fetching atomic operations have no effect from the single-transmit AMO settings.

As specified in Section IV-C, the `shmemx_signal_set` atomic operation has an unique atomicity semantics when compared to other atomic operations. This allows the Cray OpenSHMEMX library to efficiently implement the signalling set operation. The performance in Fig. 18 shows a 1.9X performance improvement in using the signaling set operation when compared to other AMOs. The single-transmit AMO settings have no effect on the signaling set operation.

In brief, users can be benefited through the right usage of the single-transmit AMO settings using the sessions extension and the signaling set operations.

VI. Conclusion

In this work, we introduced various HPE Slingshot NIC (Slingshot 11) features that impacts the performance of the operations supported by OpenSHMEM programming model. We discussed specifically on the impact of completion semantics, transfer protocol selection, event bundling, and different types of available atomicity operations. All these discussed features have a varying level of performance impact on the OpenSHMEM remote memory access (RMA) and atomic memory operations (AMO).

We introduced new extensions in Cray OpenSHMEMX, a HPE proprietary library implementation of the OpenSHMEM standard specification, to specifically exploit the various features available in Slingshot 11 NIC. For example, the bundling operation hints provided through the new OpenSHMEM sessions extension provides a 2X performance improvement on certain application models involving communication operations with low process count per node. Similarly, the usage of single-transmit AMOs and signaling set operations can provide around 1.4X and 1.9X performance improvement over the default AMOs.

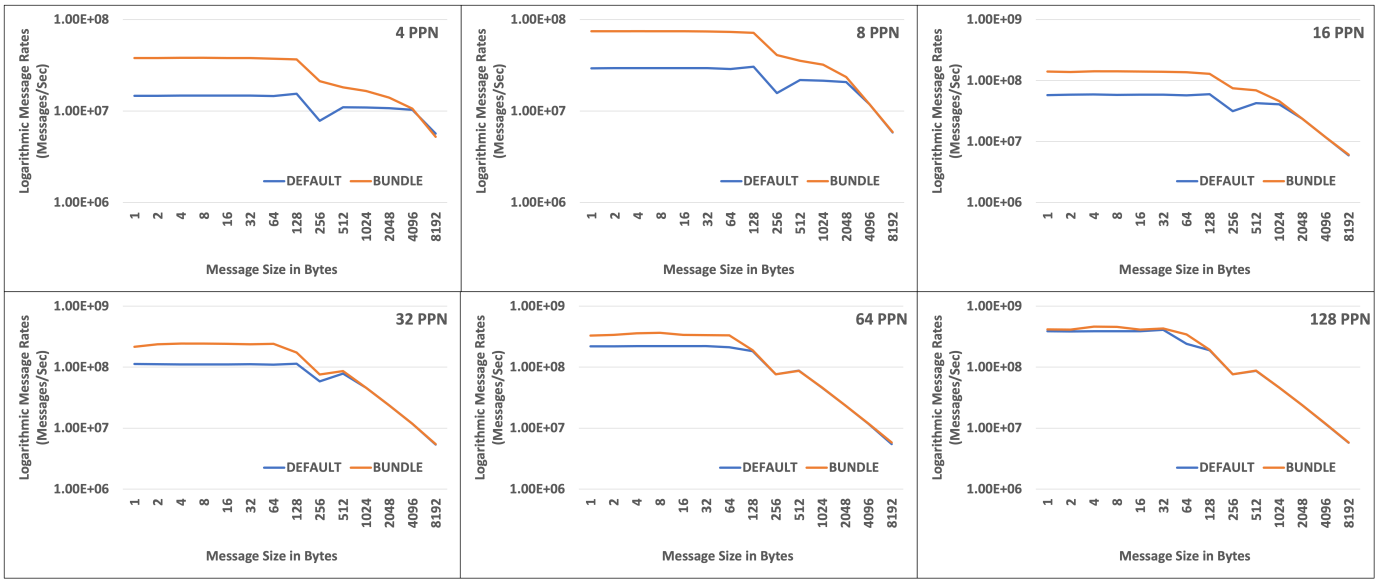

Fig. 16. OSU Non-blocking PUT Message-rate Analysis on 2 nodes with different number of PEs per Node. Showing the performance impact of bundling multiple PUT operations together against posting each PUT operation separately.

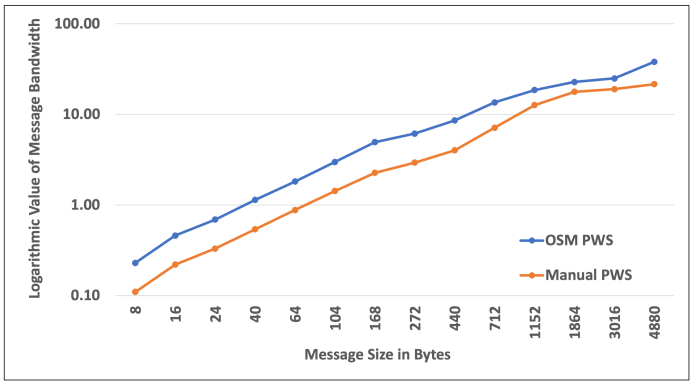

Fig. 17. HRA Random Access Blocking Put-with-Signal Bandwidth Analysis on 16 nodes with 128 PEs per Node. Showing the performance impact of OpenSHMEM Blocking `shmem_put_signal` routines compared to Manually implementing the put-with-signal semantics using blocking `shmem_put` followed by `shmem_fence` and `shmem_atomic_set` signaling operation.

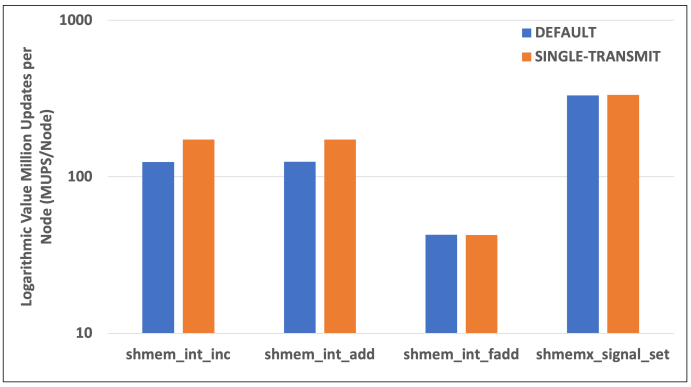

Fig. 18. OSU Blocking and Non-blocking AMO Rate Analysis 16 nodes with 1 PE per Node. Showing the performance impact of fetching and non-fetching semantics of the AMOs.

To conclude, we showed the performance impact on various parameters like the memory buffer, and completion semantic usage on different OpenSHMEM RMA and AMO operations using microbenchmark kernels representing various OpenSHMEM application usage models. A complete support of the OpenSHMEM programming model over the Slingshot 11 NIC is an on-going effort. In future, similar performance analysis and tuning suggestions is planned to be performed on other OpenSHMEM operations like collectives and team (process subset) management.

VII. Acknowledgment and Disclaimer

We would like to thank HPE Slingshot NIC architects (Keith Underwood, Igor Gorodetsky, and Bob Alverson), HPE MPT developers (Kim McMahon, Krishna Kandalla, and Nick Radcliffe), and HPE Libfabric developers (Ian Ziemba) in understanding various Slingshot 11 NIC features required to effectively implement OpenSHMEM programming model over Slingshot 11. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of associated organizations.

References

[1] Libfabric Counters. https://ofiwg.github.io/libfabric/v1.9.1/man/fi_cntr.3.html, .
[2] Libfabric Completion Queues (CQ). https://ofiwg.github.io/libfabric/v1.9.1/man/fi_cq.3.html, .
[3] Cray - Message Passing Toolkit. http://goo.gl/Cts1uh.
[4] OSU Micro-benchmarks. http://goo.gl/LgMc8e.

[5] HPE Slingshot Interconnect. https://www.hpe.com/in/en/compute/hpc/slingshot-interconnect.html.

[6] Using the GNI and DMAPP APIs. http://goo.gl/Ncjt9r.

[7] XPMEM - Linux Cross-Memory Attach. https://github.com/hjelmn/xpmem.git.

[8] OpenSHMEM standard version-1.4. http://openshmem.org/site/sites/default/site_files/OpenSHMEM-1.4.pdf, 2017.

[9] Cray's Slingshot Interconnect is at the Heart of HPE's HPC and AI Ambitions. https://tinyurl.com/22rt7utz, 2022.

[10] G. Almasi. In D. A. Padua, editor, *Encyclopedia of Parallel Computing*. 2011.

[11] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, and K. Hill. Exascale software study: Software challenges in extreme scale systems. Technical report, 2009.

[12] W. W. Carlson, J. M. Draper, and D. E. Culler. S-246, 187 Introduction to UPC and Language Specification, 1996.

[13] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing OpenSHMEM: SHMEM for the PGAS Community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, 2010.

[14] D. De Sensi, S. Di Girolamo, K. H. McMahon, D. Roweth, and T. Hoefler. An In-Depth Analysis of the Slingshot Interconnect. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

[15] J. Dinan and M. Flajslik. Contexts: A Mechanism for High Throughput Communication in OpenSHMEM. In *Proceedings of the 8th International Conference on*

[22] S. R. Paul, A. Hayashi, K. Chen, and V. Sarkar. A Scalable Actor-based Programming System for PGAS *Partitioned Global Address Space Programming Models*, PGAS '14, pages 10:1–10:9, 2014.

[16] M. P. Forum. MPI: A Message-Passing Interface Standard. Technical report, 1994.

[17] P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard, and J. M. Squyres. A Brief Introduction to the OpenFabrics Interfaces - A New Network API for Maximizing High Performance Application Efficiency. Aug 2015.

[18] D. R. Kuhn. IEEE's Posix: Making Progress. *IEEE Spectrum*, 1991.

[19] N. Namashivayam, B. Cernohous, D. Pou, and M. Pagel. Introducing cray openshmemx - a modular multi-communication layer openshmem implementation. In *OpenSHMEM 2018: Fifth Workshop on OpenSHMEM and Related Technologies*, Aug. 2018.

[20] R. H. Nieplocha J. and R. Littlefield. Global Arrays: A portable shared memory model for distributed memory computers, 1994.

[21] R. W. Numrich and J. Reid. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17(2), Aug. 1998.

Runtimes. *CoRR*, 2021. URL https://arxiv.org/abs/2107.05516.

[23] S. W. Poole, M. Grossman, V. Sarkar, and H. P. J. Pritchard. SHMEM-ML: Leveraging OpenSHMEM and Apache Arrow for Scalable, Composable Machine Learning. 2021. URL https://www.osti.gov/biblio/1820066.

[24] G. Taylor, D. Ozog, M. W.-u. Rahman, and J. Dinan. Scalable Machine Learning with OpenSHMEM, 2019.

[25] M. ten Bruggencate and D. Roweth. DMAPP: An API for One-Sided Programming Model on Baker Systems. Technical report, Cray Users Group (CUG), August 2010.