2024 IEEE 31st International Conference on High Performance Computing, Data, and Analytics (HiPC) | 979-8-3315-0909-5/24/$31.00 ©2024 IEEE | DOI: 10.1109/HIPC62374.2024.00012

Effective and Efficient Offloading Designs for One-Sided Communication to SmartNICs

Ben Michalowicz[1], Kaushik Kandadi Suresh[1], Hari Subramoni[1],
Mustafa Abduljabbar[1], Dhabaleswar K. (DK) Panda[1], Steve Poole[2]
[1]*Department of Computer Science and Engineering*, *The Ohio State University*, Columbus, USA
{michalowicz.2, kandadisuresh.1, subramoni.1, abduljabbar.1, panda.2}@osu.edu
[2]*Los Alamos National Laboratory*
{swpoole@lanl.gov}

***Abstract*—One-sided communication is one of many approaches to use for data transfer in High-Performance Computing (HPC) applications. One-sided operations require less demand on parallel programming libraries and do not require HPC hardware to issue acknowledgments of successful data transfer. Thanks to its inherently non-blocking nature, one-sided communication is also useful for improving overlap between communication and compute. As with any non-blocking communication, however, we run into the issue of message progression getting interleaved with computation. With the advent of Smart Network Cards (SmartNIC) such as NVIDIA's BlueField Data Processing Units (DPU), we can offload the communication and message progression to these devices to improve the overlap of communication and compute. In this paper, we propose designs for efficient offloading of one-sided communication. We show how our designs can be used for offloading both MPI one-sided "put" and "get" and OpenSHMEM's non-blocking "put" and "get". Using a Block Sparse Matrix-Multiplication Kernel (BSPMM), we show that our designs achieve over 96% improvement in runtime over pure-host execution for communication offload. We also briefly explore initial compute offload ideas for such one-sided kernels and show over 91% improvement in runtime here.**

***Index Terms*—Remote Memory Access, SmartNICs, BlueField DPU, MPI, One-Sided Communication, Matrix Multiplication Kernels**

I. Introduction

Today's HPC clusters utilize super-dense, many-core CPUs in tandem with high-bandwidth, low-latency network cards such as NVIDIA's ConnectX-7 products [17] and their Quantum-2 switches [19]. Communication middleware such as Message Passing Interface (MPI) and Partitioned Global Address Space (PGAS) libraries must continually adapt their designs to use these platforms efficiently.

Advancements in such hardware also allow for faster one-sided "put" and "get" operations, with middleware utilizing hardware-level features to improve end-user functionality. Research along this angle has been done in the context of MPI as well as other programming models and libraries, such as OpenSHMEM [24], where non/blocking one-sided operations exist and form the backbone for its communication.

However, as with non-blocking two-sided communication in MPI, message progression has to be interleaved with computation. This prevents programs from obtaining sufficient overlap between communication and compute, and this generally leads to a longer execution time.

NVIDIA's BlueField Data Processing Units (DPU) have allowed researchers to investigate offloading two-sided communication (see Section VII), but to the best of our knowledge, one-sided communication has never been attempted. One of the primary challenges in offloading one-sided communication is how metadata gets exchanged; two-sided communication can exchange network-level keys on a per-send/per-receive basis because all processes involved must match data from the process sending data to/receiving data from them. By comparison, one-sided communication offload is less trivial to implement, since matching of this sort is not required.

This leads us to ask the following question: *How can we efficiently offload one-sided operations to the DPU and achieve better program runtime and overlap of communication and computation?*

In this work, we propose designs for efficiently offloading one-sided communication operations — "put" and "get" — to the DPU. We also show how these can be applied to MPI, and through frameworks such as OSHMPI [3], we show how these designs can be used in the context of OpenSHMEM's non-blocking put and get primitives. In particular, we show results at the microbenchmark level for non-blocking "get" as well as show how it can lead to application-level benefits in a Block Sparse Matrix Multiplication (BSPMM) kernel — a kernel whose get-compute-update pattern is found in the NWChem computational chemistry software [7].

To the best of our knowledge, this is the first paper to propose designs for offloading one-sided communication primitives to a SmartNIC.

II. Motivation

A. Why one-sided operations?

One-sided communication (1SC) gives a developer a significant amount of flexibility in developing applications and communication patterns. One of the earliest papers introducing one-sided operations for the MPI-2 standard [11] showed how one-sided communication could outperform two-sided communication by avoiding bottlenecks such as Rendezvous-protocol handshakes and a need for individual synchronization [21]. One-sided communication only requires operations from

2640-0316/24/$31.00 ©2024 IEEE
DOI 10.1109/HiPC62374.2024.00012

the process performing a given put/get/fetch-and-op/etc. operation; its peer is not required to do anything. This allows developers and researchers to implement more flexible one-sided collective algorithms and communication patterns.

B. Challenges in Offloading One-Sided Communication

At the same time, use of one-sided communication poses some challenges. Firstly, there are more options for window/heap synchronization. MPI has a mix of passive and active synchronization capabilities that a user chooses from (Fences, Post-Start Complete Wait, and Locks) alongside functions to "complete" a put or get operation. Similarly, OpenSHMEM has analogs of these functions in the form of `shmem_wait_XXX` and `shmem_set/clear_lock`, with `shmem_quiet` being the main function used to "complete" the non-blocking communication. Each function may have different requirements/dependencies, particularly in the latter [1]; some variant of these synchronization techniques is required to prevent race conditions when many processing elements (PE) operate on either the same window (MPI-based) or on the same variable within the symmetric heap (OpenSHMEM-based). Secondly, ordering is not explicitly guaranteed with one-sided semantics, though fence operations usually solve this with a small overhead.

Offloading one-sided communication has its challenges in addition to those mentioned above. We present the following questions/challenges:

1) How do we best keep track of these operations when we do not have, for example, MPI_Request structures to use (such as with non-blocking two-sided communication)?
2) How can we effectively extend shared resources (OpenSHMEM symmetric heap, MPI Windows, etc.) to be detectable by proxy processes (placed on the DPU to perform offloaded communication)?
3) With a lack of peer process acknowledgments, how do we effectively notify the issuing process that a transfer is complete/complete the operation when introducing a proxy process for offloading said communication?
4) Message progression in two-sided communication exists for non-blocking collectives. How can we efficiently extend one-sided synchronization/progression primitives to account for the completion of data transfers via proxy processes?
5) With the above, how can we preserve the semantics and standard compliance of parallel programming libraries that utilize our designs (less trivial than with two-sided communication)?

At a high level, the answers to these questions are deceptively simple and require some serious consideration (see Section V for further details).

C. Applicability of 1SC

Many applications have previously or currently used one-sided semantics. As mentioned in Section I, NWChem's [7] primary communication pattern is a BSPMM kernel on a multi-dimensional mesh, and its popularity has led others to develop/utilize smaller kernels utilizing MPI-based Remote Memory Access (RMA) [25]. The Graph500 ([9], [28]) also utilized one-sided communication until version 3.0, and a variant was also designed using OpenSHMEM primitives for Oak Ridge National Laboratory's set of OpenSHMEM benchmarks [20]. The OSU Microbenchmark suite [22] also has a set of MPI-based RMA benchmarks and benchmarks to cover OpenSHMEM one-sided communication for put, get, and collectives.

D. How 1SC/RMA is Used in Programming Models

Both MPI and OpenSHMEM are popular programming models whose standards are continuously in active development. Both provide a host of functionalities for one-sided primitive operations. Given these, we are motivated to design and implement efficient offloading schemes for one-sided communication that can apply to both programming models/libraries. While the use of non-blocking OpenSHMEM primitives is not as easy to find in practice, MPI-based RMA is non-blocking by design (simply avoid immediately calling window synchronization/flush functions). Thanks to this, we are also motivated to make our designs adaptable to these and other parallel programming models.

III. Contributions and Paper Breakdown

We offer the following contributions in this paper:

1) A novel design for offloading one-sided operations to DPUs, namely "put" and "get" operations
2) Given our abstracted interface, the ability to integrate this into any OpenSHMEM and MPI library.
3) Demonstration of scalability and improvement over blocking and nonblocking implementations of a BSPMM kernel, where we achieve up to 71% improvement in runtime performance
4) Up to 91% runtime benefits with naive offloading of computation onto the DPU with the same kernel.

The rest of the paper is organized as follows:

- Section IV details the background on one-sided communication, the OpenSHMEM library, and a brief overview of the BlueField DPU.
- Section V describes our design for offloading one-sided operations and how they can be applied to MPI and OpenSHMEM.
- Section VI details our experimental setup and the experiments we performed for non-blocking communication and compute offloading.
- Section VII explores related work to what this paper proposes.
- Section VIII wraps up our paper and details future plans.

IV. Background

A. One-Sided Communication

One-sided communication comes in many flavors of operations: "fetch-and-op", "compare-and-swap", "put" and "get", and other atomic operations beyond the paper's scope. From a hardware perspective, one-sided operations do not necessarily

require an acknowledgment from the peer whose data is being updated/obtained.

B. The Message Passing Interface

In the case of MPI, one-sided communication and remote memory access (RMA) have existed since MPI-2.0 [27] and were massively revamped in the MPI-3.0 standard. This came in the form of: 1) a larger API to handle more fine-grained window and buffer manipulation (window attachment/detachment, allocation/shared/shared_query, etc.), and 2) extended APIs to handle more desired operations such as hybrid get/put-accumulates. MPI is one of the de facto standards for HPC communication, thus our API (see Section V) accounts for its semantics.

As mentioned in Section I, these put/get operations form the base for "point-to-point" operations in PGAS models. Furthermore, research/development like that found in [3] allows one to use MPI to mimic OpenSHMEM-based operations. We will discuss how we use this framework to our advantage later in the paper.

C. OpenSHMEM

OpenSHMEM was first standardized in 2012, though its predecessors date as far back as 1993 [1] with vendor-specific implementations such as Cray-SHMEM, SGI SHMEM, and Quadrics SHMEM. At a high level, any static, global variable, or those allocated through `shmem_malloc` to be placed on a symmetric heap, can be used inside OpenSHMEM's API calls; while this appears simpler than windows used in MPI, the main drawback of this is a user cannot pass any piece of data they see fit through OpenSHMEM's "put" or "get" operations. Compared to MPI-based RMA operations, it provides a simpler approach with explicitly defined operations per datatype and more fine-grained control without the need for explicit window allocation. Our API also accounts for its semantics as we must keep it generic enough to fit under this and MPI.

D. Overview of the BlueField SmartNIC

The BlueField DPUs from NVIDIA [5] are a system-on-chip SmartNIC with a variety of features such as on-board compression/decompression engines, DMA engines, data-path accelerator cores, and, for our purpose, general-purpose ARM CPU cores. Previous works (See Section VII) have utilized various features of the BlueField DPUs to achieve performance or further understand what they can/cannot do. In our work, we utilized advanced network capabilities provided by the InfiniBand Verbs API and these SmartNICs (see Section V) to achieve our offload.

V. Design and Implementation

This section will summarize our design for offloading non-blocking, one-sided "get" operations. We note the processes placed on the host server as "host" processes and those on the DPU as "proxy" processes. Alternatives to this in Section VII also define these as "service" processes or "worker" processes.

The proposed designs are classified into "put", "get", and "synchronize" operations. For brevity, we only show the last two, as the "put" design is very similar to the "get."

A. Extended Exchange of Network-Level Components

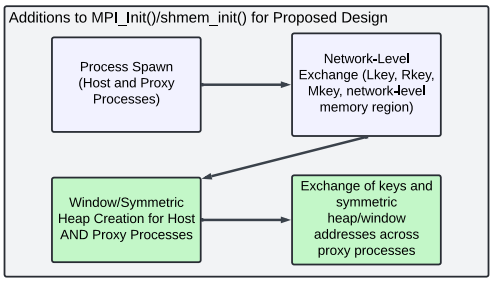


Fig. 1: High-Level Proposed modification from our framework to MPI and OpenSHMEM initialization to allow for an enhanced exchange of windows/symmetric heap so that DPU-based proxy processes are also made aware of the symmetric heap.

We previously mentioned the challenge of exchanging network-level elements such as memory keys (mkey), local keys (lkey), and remote keys (rkey) in offloading one-sided communication to the DPU. During the initialization of a parallel programming library, we extend the remote data exchange so that all keys and addresses are shared similar to the setup of a symmetric heap in OpenSHMEM. This is done with the help of the GVMI firmware found on BlueField DPUs, and the cost of key generation and exchange will ultimately be amortized during program runtime. The authors of [26] explain how GVMI can be used more in-depth. Figure 1 shows the component that could be added during initialization to any given MPI or OpenSHMEM library, where the green box for address exchange denotes this particular enhancement at a high level. This addresses the first, second, and third challenges mentioned in Section II.

B. Non-Blocking Get: Data Transfer

Figure 2 shows how the design works at a high level. We assign a proxy process for each host process that performs a non-blocking "get" operation, and internally we will keep track of each request. Load balancing will need to be done as the process-per-node (PPN) count increases, regardless of someone using a BlueField-2 or a BlueField-3 DPU. GVMI is used here to allow proxy processes to access data needed on the peer process and return it to the host.

The following list explains the steps in Figure 2 at a high level:

- We first send metadata from the host process requesting the "get" to the DPU; this involves information such as the remote rank, number of elements requested, and (through the datatype-specific OpenSHMEM primitive), the datatype used. The use of OpenSHMEM's symmetric heap versus static/global buffers is not a concern here.

- Steps (2) and (3) follow quickly after this; a proxy process on the DPU will issue a "get" to the remote process, and (upon completion of the operation) will return the data to the buffers initially passed in to the "get" function all.

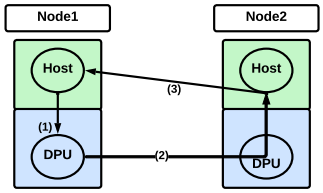


Fig. 2: "Get" Design Part 1: Steps involved in issuing a non-blocking "get" to the DPU

C. Non-Blocking Operation: Synchronization

Figure 3 continues where the previous figure left off and details the high-level approach to extending synchronization functionality to account for these offloaded operations:

- In step (4), each get operation results in an atomic fetch-and-add operation being made to keep track of the number of requests made.
- Once a synchronization or completion operation is called — namely, `MPI_win_sync/flush`-based functions or `shmem_quiet` — the requests are "flushed" and signaled as completed. A FIN signal is sent to the host-based portion of the code from the DPU-based portion for this. This will also ensure that any has successfully reached either the calling process's ("get") or the remote process's buffer ("put").

This addresses the fourth challenge mentioned near the end of Section II

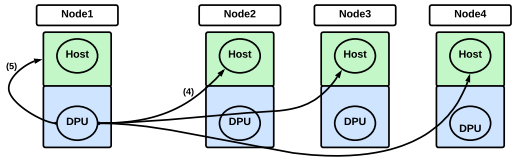


Fig. 3: "Get"Design Part 2: Performing a synchronization operation that is extended to the DPU.

D. Application to Parallel Programming Models

Thanks to the OSHMPI project, MPI-based RMA semantics can be used to emulate OpenSHMEM put/get/atomic operations as well as collectives. Because of this, we place our designs into an MPI library to show how both pure MPI programs and OpenSHMEM programs can benefit from this design. For example, an OpenSHMEM symmetric heap can be represented as an exchange of memory regions so that each processing element knows when another one issues an operation into the equivalent representation of a set of MPI-RMA Windows or the OpenSHMEM symmetric heap.

Listing 1 shows how our designs would be used inside an MPI library. Similarly, Listing 2 shows the same, simple integration for an OpenSHMEM library. This subsection addresses the fifth and final challenge mentioned near the end of Section II.

```
MPI_Win_allocate(..., win, &win_buffer){
    window = win_init(win, win_buffer);
    win_populate(window, win_buffer);
    proxy_exchange_win(&window, win_buffer);
    return window;
}
MPI_Put(addr1, count1, datatype1, target_rank,
    target_disp, target_count, target_datatype,
    window){
    // Other metadata setup ...
    addr2 = buf_of (window) + disp
    bytes = count*get_size(datatype);
    return Offload_put(addr1, addr2, target, bytes);
}
MPI_Get(addr1, count1, datatype1, target_rank,
    target_disp, target_count, target_datatype,
    window){
    // Other metadata setup ...
    addr2 = buf_of (window) + disp
    bytes = count*get_size(datatype);
    return Offload_get(addr1, addr2, target, bytes);
}
MPI_win_flush_all(window){
    return Offload_flush(window);
}
```

Listing 1: Use of One-Sided Offload in MPI. Datatypes are passed in as a parameter with MPI. This lets us utilize internal functions of a given MPI library to determine the equivalent size of that type before calculating the destination address.

```
shmem_init(){
    initiate_symm_heap();
    proxy_exchange_symm_heap();
}
shmem_malloc(size){
    buffer = allocate();
    barrier(); /* All procs allocate */
    proxy_exchange_update(buffer);
    return buffer;
}
shmem_TYPE_put_nbi(TYPE *src, const TYPE *dst, int
    count, int target){
    bytes = count * sizeof(TYPE);
    Offload_put(dst, src, target, bytes);
}
shmem_TYPE_get_nbi(TYPE *src, const TYPE *dst, int
    count, int target){
    bytes = count*sizeof(TYPE);
    Offload_get(dst, src, target, bytes);
}
shmem_quiet(){
    Offload_flush(NULL);
}
```

Listing 2: Use of One-Sided Offload in OpenSHMEM. Because of different functions existing for different datatypes there is a little less overhead involved before offloading communication. We specify types here to show this distinction from MPI. In the case of a `void*` type we can omit the calculation of bytes.

VI. Experiments

This section discusses our experimental setup and results. Due to the nature of OpenSHMEM not being as popular of a

programming model as the Message Passing Interface, there are not many applications that utilize OpenSHMEM, and to the best of our knowledge, there are fewer that utilize its non-blocking primitives.

A. System Setup

We utilize a cluster consisting of 32 "host" servers with Intel(R) Xeon(R) Gold 6138 CPUs @ 2.00 GHz (40 cores across 2 sockets). 16 of these are equipped with NVIDIA BlueField-2 DPUs with 100Gb/s InfiniBand. For simplicity, we develop our solutions into an MPI library that runs underneath the OSHMPI project [3]. We make most of our comparisons in this fashion against the OSHMPI framework running over the MVAPICH2-2.3.7 MPI library as developed by The Ohio State University [23], followed by a few comparisons against the reference OpenSHEM library as developed by Open Source Software Solutions [24].

B. OSU Microbenchmarks

We utilize the OSU Microbenchmark (OMB) suite's OpenSHMEM benchmarks to show improved latency and overlap for non-blocking get operations [22]. Figure 4 shows latency and overlap, respectively, of "pure" OpenSHMEM and our proposed design for intra-node transfers (1 Node, 2 Processes Per Node (PPN)). Because we still need to deal with the host server performing message progression in non-blocking communication, offloading the operation allows for a larger overlap in communication and compute.

We note that, while latency improves only at much higher message sizes, *our aim and focus is to be able to achieve a high level of overlap*, and we show that at larger message sizes, we achieve up to 75% overlap in the intra-node case. Further profiling has shown that our design offloading communication to the DPU demonstrates a higher sensitivity to cache behavior than standard intra-node schemes. In applications utilizing non-blocking communication, large compute portions showcase the benefits of offloading better than small/light portions.

Figure 5 shows similar trends for inter-node, one-sided operations. Like with the intra-node results, we also show results when using the symmetric heap versus global, static variables. Here, we see similar behaviors in pure communication latency and once again consistently see up to 75%-78% overlap as we increase the message size.

In intra-node and inter-node communication, the use of symmetric heap against static global "allocation" of buffers plays a role in performance in the case of pure-host, where our designs are less impacted by such behavior.

C. Ported Block Sparse Matrix Multiplication (BSPMM) Mini-application – Communication Offload

We took the BSPMM mini-application found at [25] and ported its code to utilize both blocking and non-blocking OpenSHMEM "get" primitives — that is, `shmem_get/get_nbi`. It follows a simplified version of the communication pattern found in NWChem: a get-compute-update algorithm. For a given number of work units, we perform a remote "get" operation for two separate buffers A and B across each process, with a local copy performed in the event the source rank is also the target rank. This will prevent unnecessary function calls into a given pure or MPI-backed OpenSHMEM library. This is followed by a DGEMM operation on each rank's local buffer into a third buffer C. Each process then accumulates the data into a "global" copy of C before moving on to the next portion of data. In our implementation of a non-blocking variation of the kernel, the first two "get" operations are performed before the start of the main loop via blocking operations. During the loop, computation is done while the next two "get" operations are performed using non-blocking primitives, which gives us the potential for overlap. After computation, the original A and B buffers get updated and an accumulation is done over each PE's local copy of C into PE 0.

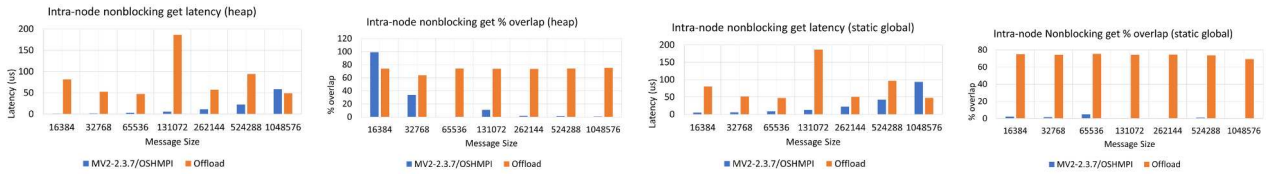
The kernel "prefers" a square number of processes, and when possible, a square number of compute nodes for its internal mapping of divided data to each of the processing elements. Directly going to a full-subscription process count may not always be optimal.

*1) **Challenges in Porting to OpenSHMEM**:* Several factors had to be considered here when porting the kernel to use non-blocking OpenSHMEM functions. These include: 1) management of buffers through OpenSHMEM's global heap and/or labeling buffers as "static" with a global scope, thereby replacing MPI windows, 2) explicit management of results from OpenSHMEM "atomic fetch-add" operations instead of MPI's "fetch-and-op", and 3) the use of locks for more fine-grained control over the initial kernel's use of Window-locking mechanisms and flushing. The latter is nontrivial as using enough locks incur a performance penalty from serializing code that was meant to be parallelized; similarly not using enough locks can incur data races, which also is not beneficial. The issue of data races and efforts to combat them drastically increases with the introduction of non-blocking primitives.

The above challenges lead to potential performance issues: 1) instead of letting an MPI library perform offset calculation internally, we must point OpenSHMEM primitives to the right position at the application level. This can allow user-made fine-grained optimizations for this, but with changing offsets at the application level, this puts more effort on the programmer; 2)

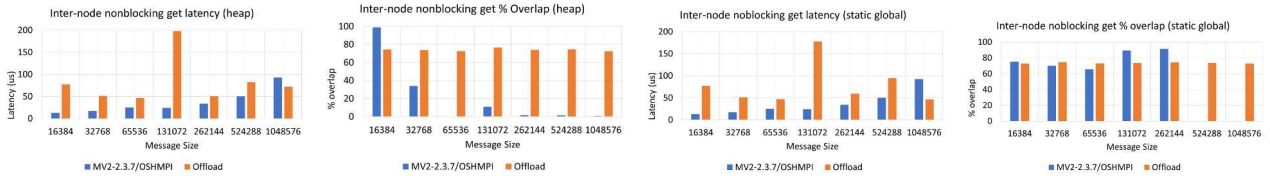
*2) **Communication Offload Results**:* Figures 6 and 7 show how our design behaves in single-node experiments. Our initial implementation of the non-blocking BSPMM kernel slowly becomes less performant as communication dominates the application runtime (get operations in particular), though our offload design shows up to 71% runtime improvement against it even at smaller scales. At present, we see many of the benefits in smaller mesh sizes (8×8 and 16×16) because of a mixture of work distribution and having sufficient compute to dominate program runtime.

This also implies a further need to refine how the non-blocking kernel is implemented to show further performance improvements – proper lock usage to avoid data races becomes



(a) Intra-node, heap-based, "get" latency (b) Intra-node, heap-based, "get" overlap (c) Intra-node, static-global-based, "get" latency (d) Intra-node, static-global-based, "get" overlap

Fig. 4: Intra-node latency and overlap performance when offloading one-sided "get" operations through OMB OpenSHMEM benchmark suite



(a) Inter-node, heap-based, "get" latency (b) Inter-node, heap-based, "get" overlap (c) Inter-node, static-global-based, "get" latency (d) Inter-node, static-global-based, "get" overlap

Fig. 5: Inter-node latency and overlap performance when offloading one-sided "get" operations

non-trivial when OpenSHMEM-based accumulate and the compute portions are among the few remaining portions of this kernel (and possibly others) that are left. These critical require every PE to individually touch buffers located on the symmetric heap.

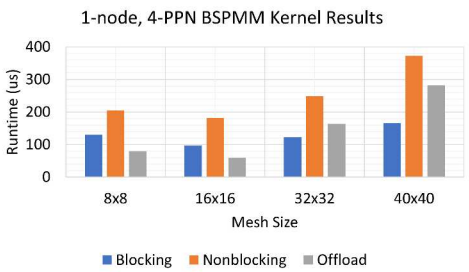


Fig. 6: 1-Node, 4-PPN Communication Offload Results (MV2-2.3.7/OSHMPI comparison)

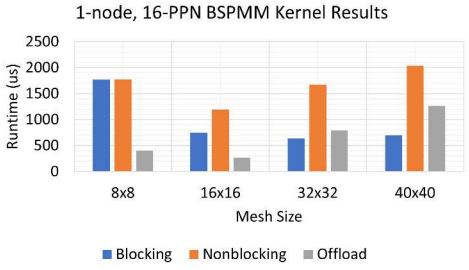


Fig. 7: 1-Node, 16-PPN Communication Offload Results (MV2-2.3.7/OSHMPI comparison)

Similar trends are seen at 4-node scales with Figures 8 and 9. In addition to the above reasoning, the extended `shmem_quiet` to "complete" one-sided offload operations may also generate overhead. In particular, we would like to highlight much more massive benefits at larger problem sizes and larger scales (8 Nodes, up to 32 PPN) as seen in Figure 10. Here, we see up to 96% improvement against the blocking kernel implementation, and up to 76% improvement against the nonblocking kernel implementations with our designs. Digging further into these larger-scale runs, we also see up to an 84% reduction in the "get" operation time within the kernel, as shown in Figure 11. Given this, we would like to emphasize that the majority of runtime benefits come from an improved overlap in communication and computation as seen from the OMB results in Section VI-B.

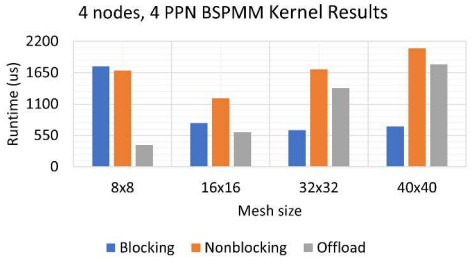


Fig. 8: 4-Node, 4-PPN Communication Offload Results (MV2-2.3.7/OSHMPI comparison)

We also compare against the reference implementation of

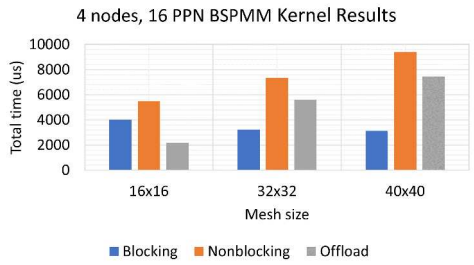


Fig. 9: 4-Node, 16-PPN Communication Offload Results (MV2-2.3.7/OSHMPI comparison)

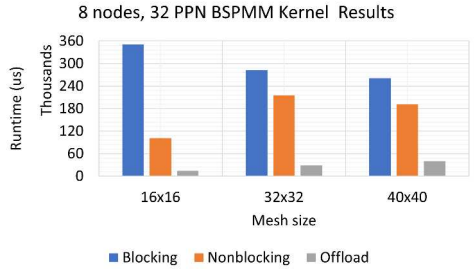


Fig. 10: 8-Node, 32-PPN Communication Offload Results (MV2-2.3.7/OSHMPI comparison)

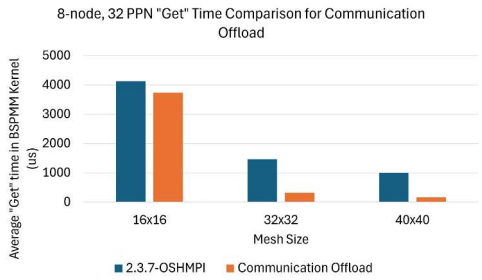


Fig. 11: 8-Node, 32-PPN "Get" Comparison for Communication Offload (Non-blocking Kernel, MV2-2.3.7/OSHMPI comparison)

OpenSHMEM[1]. Figures 15 and 16 show our single-node results; here, we see over 60% improvement in runtime performance against the reference implementation when using the non-blocking kernel port.[2]

To further emphasize the above point, we also present figures for a subset of the results previously shown. Figures 12, 13, and 14, show the communication breakdown of how our design performs compared to that of the MVAPICH2-2.3.7/OSHMPI configuration and, where possible, the OSSS-UCX configuration; namely, we focus on the large 40×40 mesh size. While local copies are minuscule, we include them for completeness in the breakdown.

We note two things from these results. The first is that we still achieve performance benefits against other parallel programming library configurations despite spending a proportionately longer time in accumulate and "get" in the 1-node case (0% or 2% against 5-6%), 4-node case (2-3% to over 20%), and 8-node case, though the latter shows a much different picture when accounting for a larger PPN count and scale than previous results.

[1]This is dubbed as OSSS-UCX through their GitHub repository: https://github.com/openshmem-org/osss-ucx/

[2]Runtime errors have prevented us from scaling OSSS-UCX beyond 1 node for this kernel. We are working to debug this at the application level.

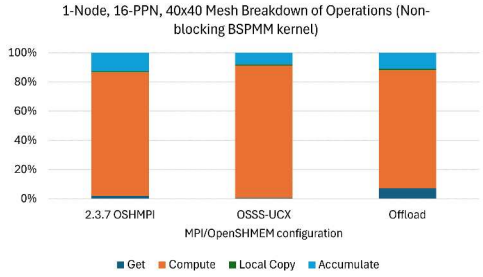


Fig. 12: 1-Node, 16-PPN Communication Offload Breakdown of OSSS-UCX, MVAPICH2-2.3.7/OSHMPI, and our offloaded designs when using the 40×40 mesh size

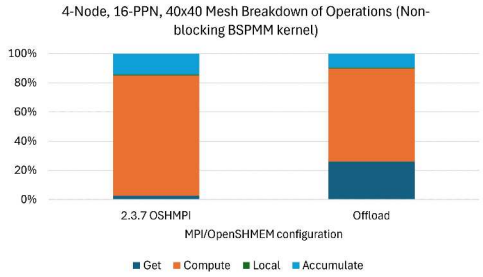


Fig. 13: 4-Node, 16-PPN Communication Offload Breakdown of OSSS-UCX, MVAPICH2-2.3.7/OSHMPI, and our offloaded designs when using the 40×40 mesh size

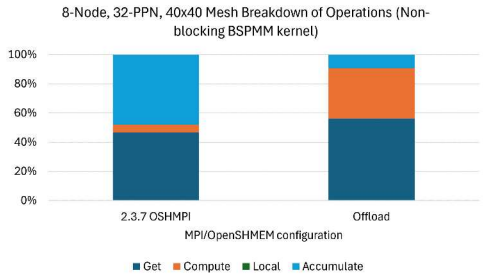


Fig. 14: 8-Node, 32-PPN Communication Offload Breakdown of OSSS-UCX, MVAPICH2-2.3.7/OSHMPI, and our offloaded designs when using the 40×40 mesh size

D. Ported BSPMM: Naive Compute Offload

We also offer the possibility of naive compute offload as an alternative to communication offload. By using the general-purpose CPUs on the BlueField DPUs, we can also place some of the compute onto the DPUs and help reduce the overhead of

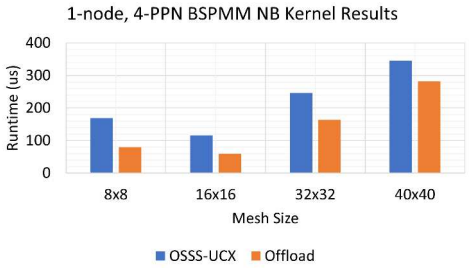


Fig. 15: 1-Node, 4-PPN Communication Offload Results Against the Reference Implementation of OSSS-UCX

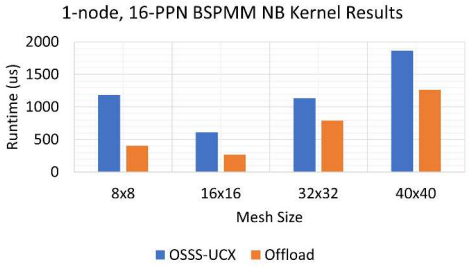


Fig. 16: 1-Node, 16-PPN Communication Offload Results Against the Reference Implementation of OSSS-UCX

data movement during application runtime. These results use only the MVAPICH2/OSHMPI software stack on the blocking kernel for brevity. While the kernel's get-compute-update design easily lends itself to compute offload, the following subsections show that the improvements largely grow as we increase the scale at which we execute it.

1) ***Results: Small-Scale, Small-Host-PPN****:* Not all scales are meant to be offloaded. To showcase the lack of benefits in smaller scale and small-host-PPN experiments, Figures 17, 18 and 19 detail experiments that, when we have fewer host-based PPN than DPU-based PPN (or workers per node/WPN), show anything from lack of benefits to performance degradations. In cases like these, it may not be better to offload unless a more intelligent design is made available for this kernel or any application in mind that can adapt to smaller scales. In conjunction/alternatively, use of the NVIDIA BlueField-3 DPUs, with improved hardware, can further alleviate some of the bottlenecks shown in these cases.

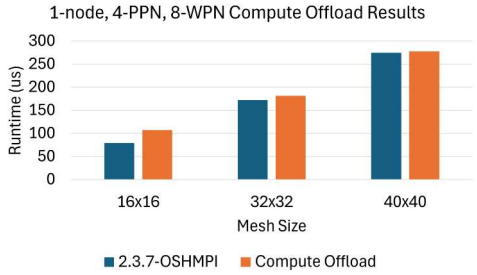


Fig. 17: 1-Node, 4-PPN, 8-WPN Compute Offload Results

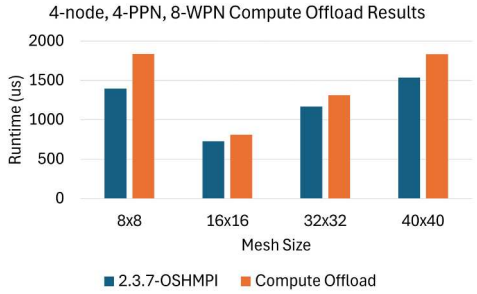


Fig. 18: 4-Node, 4-PPN, 8-WPN Compute Offload Results

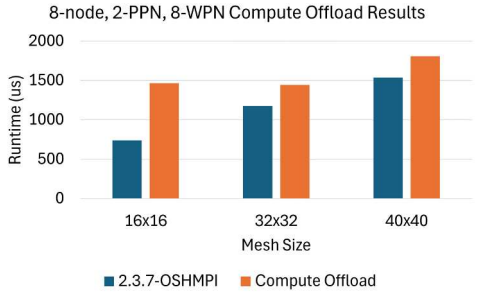


Fig. 19: 8-Node, 2-PPN, 8-WPN Compute Offload Results

2) ***Results: Larger-scale, Larger-Host-PPN****:* Figures 20 and 21 showcase single-node results. Placing compute onto the DPU, outside of performance variations with smaller mesh sizes, allows for up to a 10% runtime improvement, though with naively placing processes onto the DPU, we do not gain a significant benefit unless we significantly scale up and out. For these and other experiments, we focus solely on utilizing all available cores on the BlueField-2 SmartNICs.

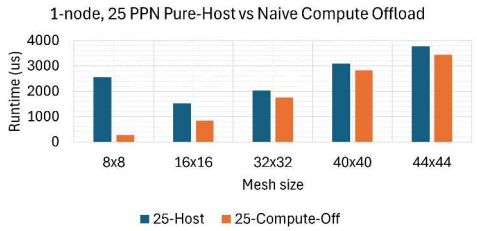


Fig. 20: 1-Node, 25-PPN Compute Offload Results

Figures 22 and 23 shows experimental results at 4 nodes with 25 and 36 PPN. While we see up to 91.5% improvement in total runtime with smaller mesh sizes and up to 60% with larger meshes, there is a slowly increasing overhead from the increasing mesh sizes. Similar trends are shown at the 8-node scale, up to 32 PPN (256 processes total) with offload (seen in Figures 24 and 25), where we see up to a 91% improvement in runtime with the 32x32 mesh.

Much of the improvement comes from a reduced remote "get" operation time; this is shown for 8 Nodes and 18/32PPN in Figures 26, 27, 28 and 29. In the 4-node scales, we see up

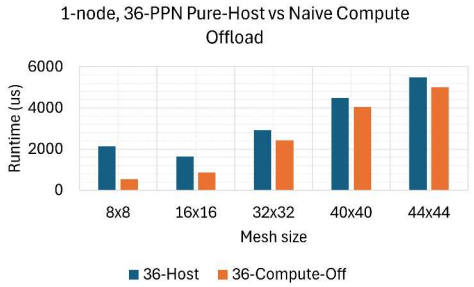


Fig. 21: 1-Node, 36-PPN Compute Offload Results

to a 96.6% reduction in the average "get" operation time per iteration of the BSPMM kernel (specifically 4 nodes, 25 PPN, 32×32 mesh). In the 8-node experiments, we up to a 98.9% improvement in the "get" operation time per iteration (8 nodes, 18 PPN, size 16×16 mesh).

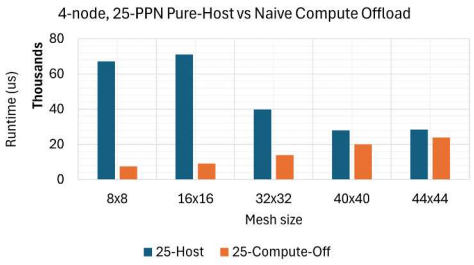


Fig. 22: 4-Node, 25-PPN Compute Offload Results

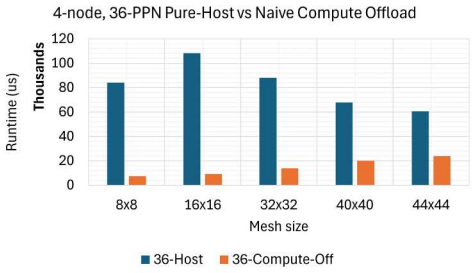


Fig. 23: 4-Node, 36-PPN Compute Offload Results

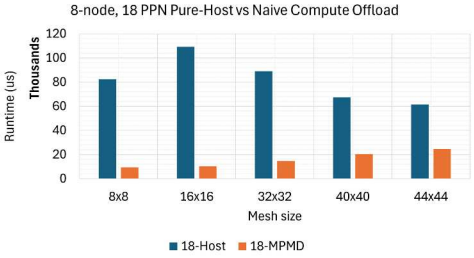


Fig. 24: 8-Node, 18-PPN Compute Offload Results

3) Key Takeaways of One-Sided Offload: As we have seen, offloading communication and compute will vary at different scales and problem sizes. At smaller scales, offloading

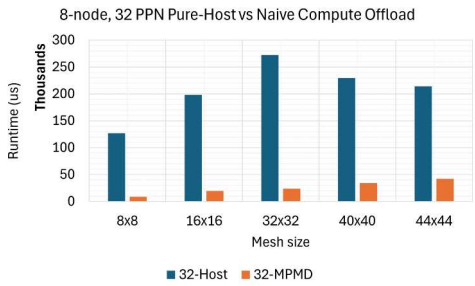


Fig. 25: 8-Node, 32-PPN Compute Offload Results

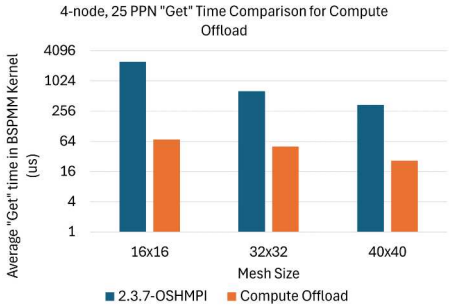


Fig. 26: 4-Node, 25-PPN Compute Offload Results

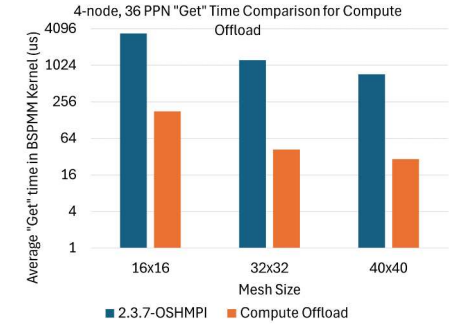


Fig. 27: 4-Node, 36-PPN Compute Offload Results

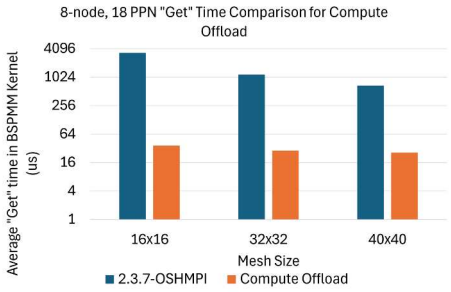


Fig. 28: 8-Node, 18-PPN "Get" Comparison for Compute Offload

communication and/or compute will either incur overhead or simply be less beneficial than at larger scales. In this work and in others (See Section VII), small-message offloading is not beneficial unless additional schemes are used to saturate

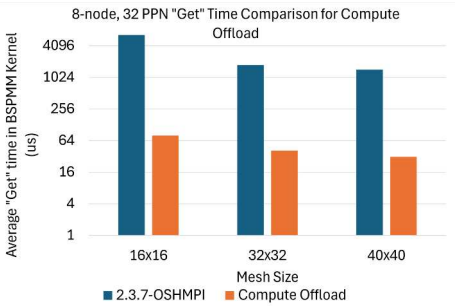


Fig. 29: 8-Node, 32-PPN "Get" Comparison for Compute Offload

the network bandwidth and make the offloading a less expensive operation. At larger scales, offloading dense/complex communication allows for our design to shine given the level of network strain even when performing one-sided put/get operations. Larger message sizes also prove beneficial as the host will not need to wait for completion of/issue progress "checks" on a given message transfer. This, coupled with increasing scales, offloading proves to be beneficial.

E. Further Discussion on Porting

Our attempt at porting the BSPMM kernel was meant as an aid in showcasing the efficacy of our design; our initial attempts involved mapping MPI one-sided primitives to OpenSHMEM's non-blocking counterparts. Non-blocking communication requires more careful synchronization of buffer usage than blocking communication, and this contributes, in part, to the increased overheads shown in previous sections. Other quasi-mappings of MPI-related functions to their OpenSHMEM counterparts were met with varying success; for example, our attempts at using `shmem_reduce` in place of `MPI_Accumulate` did not give us the desired results, thus requiring a naive, hand-made attempt to emulate the behavior of the latter with blocking put/get operations. However, we do not aim to enhance the performance of accumulation primitives in this work, and this is not the main portion of the results shown. Further, non-trivial profiling of pure-host performance and engineering of the kernel would allow us to show two things: 1) that non-blocking performance will show some level of benefit (i.e. $> 5\%$) compared to blocking performance; 2) that our offload designs will still scale and out-perform pure-host, non-blocking communication.

VII. Related Work

A. SmartNICs

SmartNICs such as the BlueField DPU have been gaining traction since the BlueField-2's release in 2020. Many researchers have dedicated time to understanding and utilizing them to the best of their abilities.

1) Offloading Non-blocking communication: BluesMPI [4] was the one of the first to use DPUs to accelerate non-blocking Alltoall communication in MPI. A follow-up work [26] moved from the staging designs found in BluesMPI to more fine-grained network primitives to handle non-blocking communication. Graham et al [8] recently presented the first paper to use DPUs for blocking collectives as well.

2) Benchmarks and Evaluation: [15] proposed the first DPU-Aware benchmark suite to show how one could evaluate offload efficiency through microbenchmarks designed for DPUs. [16] and [14] showcased different performance comparisons between the BlueField-2 and BlueField-3 SmartNICs; the former focused on the evaluation of the general purpose cores and network adapters while the latter focused on the use of the DPU's compression/decompression engines. To the best of our knowledge, many of the engines and capabilities of the later BlueField SmartNICs have yet to be evaluated, thus leading to an interesting research direction for more network-based/network-centric computing in HPC clusters..

3) Application Usage: The authors of [12] presented how BlueField SmartNICs can be used for performance improvement of application-specific workloads, such as those found in the MiniMD molecular dynamics application. Similarly, the authors of [29] performed studies through the PENNANT mini-application to also offload its workloads to the DPU.

B. One-Sided Communication

One of the earliest one-sided communication designs was presented in [11], where they present one-sided operations mapped directly to InfiniBand RDMA primitives. [13] utilized more advanced InfiniBand primitives by mapping software-level atomics to InfiniBand-level atomic operations. When GPUs began getting leveraged by communication middleware, an effort was made to utilize them in communication middleware for one-sided operations. The authors of [6] recently presented an evaluation of one-sided communication on CPUs and GPUs. They present a roofline model for one-sided communication and compare current state-of-the-art libraries such as NVSHMEM [18] and ROCM_SHMEM [2] as well as one-and-two-sided MPI-based communication.

Bridging the use of SmartNICs and one-sided communication, the authors of [10] designed and developed extensions to OSSS-UCX for fine-grained access to persistent memory. One of this project's goals was to also extend this work to SmartNICs such as the BlueField DPUs.

VIII. Conclusion and Further Research

In this paper, we have presented two things: The first is a novel design for offloading non-blocking one-sided "put" and "get" operations to the DPU and how they can be applied to different parallel programming models such as MPI and PGAS (namely OpenSHMEM). We have also shown how these can be applied to communication patterns such as those found in NWChem. In particular, we have shown up to a 96% reduction in runtime for a BSPMM kernel thanks to improved overlap of communication and compute using non-blocking "get" operations. The second is that we have also demonstrated how even the BlueField-2 DPU can aid a program through computation offload by reducing the cost of data movement. We believe these findings with this communication pattern can

be further extended to NWChem and other applications that utilize one-sided communication.

Given the performance shown in Section VI, we also wish to evaluate when it would be best to offload communication or compute operations through dynamic runtime tuning and how much should get offloaded to the DPU for the latter. [15] investigated the notion of offload efficiency with emphasis on larger message sizes. Their work could be applicable to investigate simple get/put efficiency using advanced network primitives such as those presented in this paper. With the growing popularity and visibility of NVIDIA's BlueField-3 SmartNICs, we also wish to perform experiments with them to show how advancements in hardware allows for further benefits for both communication and compute offload.

IX. Acknowledgments

We would like to thank the HPC-AI Advisory Council for allowing us to use their resources for experiments. We acknowledge that this work is funded by the LANL/US DoD SOW #19537, NSF Grant #2312927, and NSF Grant #2007991.

References

[1] "OpenSHMEM Application Programming Interface," June 2020. [Online]. Available: http://openshmem.org/site/sites/default/site_files/openshmem_specification-1.5.pdf

[2] AMD, "ROCSHMEM." [Online]. Available: https://github.com/ROCm/ROC_SHMEM

[3] Argonne National Laboratory, "The OSHMPI project." [Online]. Available: https://github.com/pmodels/oshmpi

[4] M. Bayatpour, N. Sarkauskas, H. Subramoni, J. Maqbool Hashmi, and D. K. Panda, " BluesMPI: Efficient MPI Non-blocking Alltoall Offloading Designs on Modern BlueField Smart NICs ," in *High Performance Computing*, B. L. Chamberlain, A.-L. Varbanescu, H. Ltaief, and P. Luszczek, Eds. Cham: Springer International Publishing, 2021, pp. 18–37.

[5] I. Burstein, "Nvidia Data Center Processing Unit (DPU) Architecture," in *2021 IEEE Hot Chips 33 Symposium (HCS)*, 2021, pp. 1–20.

[6] N. Ding, M. Haseeb, T. Groves, and S. Williams, "Evaluating the performance of one-sided communication on cpus and gpus," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1059–1069. [Online]. Available: https://doi.org/10.1145/3624062.3624182

[7] Environmental Molecular Sciences Laboratory, Pacific Northwest National Laboratory, "NWChem: Delivering High-Performance Computational Chemistry to Science," `http://www.nwchem-sw.org/index.php/Main_Page`.

[8] R. Graham, G. Bosilca, Y. Qin, B. Settlemyer, G. Shainer, C. Stunkel, G. Vallee, B. Williams, G. Cisneros-Stoianowski, S. Ohlmann, and M. Rampp, "Optimizing application performance with bluefield: Accelerating large-message blocking and nonblocking collective operations," in *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*, 2024, pp. 1–12.

[9] Graph500, "The Graph500 Benchmark." [Online]. Available: https://github.com/graph500/graph500

[10] M. Grodowitz, P. Shamis, and S. Poole, "Openshmem i/o extensions for fine-grained access to persistent memory storage," in *SMC2020*. SMC2020, 2020.

[11] W. Jiang, J. Liu, H.-W. Jin, D. Panda, W. Gropp, and R. Thakur, "High performance mpi-2 one-sided communication over infiniband," in *IEEE International Symposium on Cluster Computing and the Grid, 2004. CCGrid 2004.*, 2004, pp. 531–538.

[12] S. Karamati, C. Hughes, K. S. Hemmert, R. E. Grant, W. W. Schonbein, S. Levy, T. M. Conte, J. Young, and R. W. Vuduc, ""Smarter" NICs for faster molecular dynamics: a case study," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2022, pp. 583–594.

[13] M. Li, S. Potluri, K. Hamidouche, J. Jose, and D. K. Panda, "Efficient and truly passive mpi-3 rma using infiniband atomics," in *Proceedings of the 20th European MPI Users' Group Meeting*, ser. EuroMPI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 91–96. [Online]. Available: https://doi.org/10.1145/2488551.2488573

[14] Y. Li, A. Kashyap, Y. Guo, and X. Lu, "Characterizing Lossy and Lossless Compression on Emerging BlueField DPU Architectures," in *2023 IEEE Symposium on High-Performance Interconnects (HOTI)*, 2023, pp. 33–40.

[15] B. Michalowicz, K. Kandadi Suresh, H. Subramoni, D. Panda, and S. Poole, "DPU-Bench: A Micro-Benchmark Suite to Measure Offload Efficiency Of SmartNICs," in *Practice and Experience in Advanced Research Computing*, ser. PEARC '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 94–101. [Online]. Available: https://doi.org/10.1145/3569951.3593595

[16] B. Michalowicz, K. K. Suresh, H. Subramoni, D. K. Panda, and S. Poole, "Battle of the BlueFields: An In-Depth Comparison of the BlueField-2 and BlueField-3 SmartNICs," in *2023 IEEE Symposium on High-Performance Interconnects (HOTI)*, 2023, pp. 41–48.

[17] NVIDIA, "NVIDIA ConnectX-7 NDR 400 InfiniBand Adapter Card." [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/networking/infiniband-adapters/infiniband-connectx7-data-sheet.pdf

[18] ——, "NVIDIA NVSHMEM Documentation." [Online]. Available: https://docs.nvidia.com/nvshmem/index.html

[19] NVIDIA, "Nvidia quantum-2 infiniband platform." [Online]. Available: https://www.nvidia.com/en-us/networking/quantum2/

[20] Oak Ridge National Laboratory, "ORNL-OpenSHMEM benchmarks." [Online]. Available: https://github.com/ornl-languages/osb

[21] Oracle, "Sun HPC ClusterToolstrademark 6 Software Performance Guide." [Online]. Available: https://docs.oracle.com/cd/E19061-01/hpc.cluster6/819-4134-10/1-sided.html

[22] OSU Micro-benchmarks, http://mvapich.cse.ohio-state.edu/benchmarks/.

[23] D. K. Panda, H. Subramoni, C.-H. Chu, and M. Bayatpour, "The mvapich project: Transforming research into high-performance mpi library for hpc community," *Journal of Computational Science*, vol. 52, p. 101208, 2021, case Studies in Translational Computer Science. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877750320305093

[24] S. W. Poole, O. Hernandez, J. A. Kuehn, G. M. Shipman, A. Curtis, and K. Feind, *OpenSHMEM - Toward a Unified RMA Model*. Boston, MA: Springer US, 2011, pp. 1379–1391. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_490

[25] Rohit Zambre and Subhadeep Bhattacharya, "BSPMM Mini App." [Online]. Available: https://github.com/rzambre/bspmm

[26] K. K. Suresh, B. Michalowicz, B. Ramesh, N. Contini, J. Yao, S. Xu, A. Shafi, H. Subramoni, and D. Panda, " A Novel Framework for Efficient Offloading of Communication Operations to Bluefield SmartNICs ," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023, pp. 123–133.

[27] The MPI Forum, "MPI-2.0 Report." [Online]. Available: https://www.mpi-forum.org/docs/mpi-2.0/mpi-20-html/mpi2-report.html

[28] K. Ueno and T. Suzumura, "Highly scalable graph search for the Graph500 benchmark," in *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 149–160. [Online]. Available: https://doi.org/10.1145/2287076.2287104

[29] B. K. Williams, W. K. Poole, and S. W. Poole, "Investigating Scientific Workload Acceleration using BlueField SmartNICs [Slides]," 3 2020. [Online]. Available: https://www.osti.gov/biblio/1607904