2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 979-8-3503-3766-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/IPDPS54959.2023.00022

A Novel Framework for Efficient Offloading of Communication Operations to Bluefield SmartNICs

Kaushik Kandadi Suresh, Benjamin Michalowicz, Bharath Ramesh,
Nick Contini, Jinghan Yao, Shulei Xu, Aamir Shafi, Hari Subramoni, Dhabaleswar Panda
Department of Computer Science and Engineering
The Ohio State University Columbus, USA
{kandadisuresh.1, michalowicz.2, ramesh.113, contini.26, yao.877, xu.2452, shafi.16, subramoni.1, panda.2}@osu.edu

***Abstract*—Smart Network Interface Cards (SmartNICs) such as NVIDIA's BlueField Data Processing Units (DPUs) provide advanced networking capabilities and processor cores, enabling the offload of complex operations away from the host. In the context of MPI, prior work has explored the use of DPUs to offload non-blocking collective operations. The limitations of current state-of-the-art approaches are twofold: They only work for a pre-defined set of algorithms/communication patterns and have degraded communication latency due to staging data between the DPU and the host. In this paper, we propose a framework that supports the offload of any communication pattern to the DPU while achieving low communication latency with perfect overlap. To achieve this, we first study the limitations of higher-level programming models such as MPI in expressing the offload of complex communication patterns to the DPU. We present a new set of APIs to alleviate these shortcomings and support any generic communication pattern. Then, we analyze the bottlenecks involved in offloading communication operations to the DPU and propose efficient designs for a few candidate communication patterns. To the best of our knowledge, this is the first framework providing both efficient and generic communication offload to the DPU. Our proposed framework outperforms state-of-the-art staging-based offload solutions by 47% in Alltoall micro-benchmarks, and at the application level, we see improvements up to 60% in P3DFFT and 15% in HPL on 512 processes.**

***Index Terms*—HPC, Infiniband, MPI, SmartNIC, DPU, Offload, GVMI, Cross-GVMI**

I. Introduction

Modern CPU-based High-Performance Computing (HPC) clusters employ the use of powerful processors with high core counts, and high-bandwidth, low-latency network interface cards (NICs)/switches. The emergence of network hardware such NVIDIA's ConnectX-7 NICs [1]/Quantum-2 switches [2] capable of 400Gbps per port, and 128+ core AMD EPYC CPUs [3], indicate a trend toward supporting massive amounts of compute and network parallelism for AI/HPC workloads. The onus is on communication libraries and applications to efficiently utilize these platforms. A popular strategy to achieve this goal is by **offloading** communication operations to another thread/hardware resource to overlap them with compute operations. The idea of "offloading" a communication pattern can be viewed from two perspectives: 1) The set of *APIs* that define how to orchestrate the offload of these patterns, and 2) The underlying *mechanisms* used to efficiently offload communication operations.

The *mechanisms* used for offloading communication operations either involve an asynchronous thread [4] on the CPU, or an external hardware device such as a SmartNIC or switch. While using another thread on the CPU for progressing communication is convenient, the presence of additional threads steals CPU cycles that would otherwise be used for application computation. Switch-based offload techniques, such as NVIDIA SHARP [5], are highly effective for certain communication operations, especially because switches can eliminate redundancy, are aware of the network topology, and can perform operations at very fast rates. However, current switch-based solutions only support barrier/reduction operations and operate on limited switch-based resources.

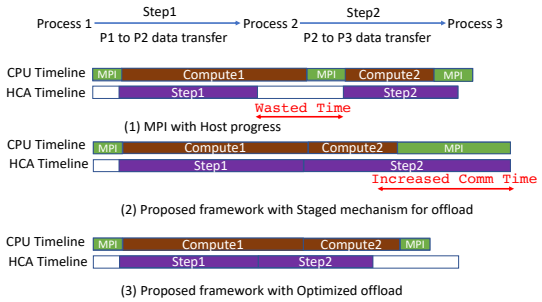


Fig. 1. Comparison of a Ring-based broadcast communication pattern per process between 1) Standard MPI implementation, 2) Staging offload mechanism utilizing new proposed primitives/APIs, 3) Proposed, advanced offload mechanism utilizing new proposed primitives/APIs

From an *API* perspective, popular programming models such as MPI [6] and OpenSHMEM [7] provide primitives for non-blocking point-to-point and collective operations. While these non-blocking functions are effective at expressing a wide variety of offload scenarios, they fail to capture the semantics of ordering without synchronized calls such as MPI_Wait. This is problematic in cases where the host tries to offload a series of ordered operations to another device (such as a SmartNIC). Issuing an MPI_Wait would force the host to block until the operation completes, which results in sub-optimal utilization of CPU resources. In the context of MPI, the

*This research is supported in part by NSF grants #1818253, #1854828, #1931537, #2007991, #2018627, #2112606, and XRAC grant #NCR-130002.

1530-2075/23/$31.00 ©2023 IEEE
DOI 10.1109/IPDPS54959.2023.00022

same problem is exacerbated with certain collective algorithms that are implemented using only point-to-point operations. This is explained in Figure 1 which shows a simple ring communication pattern that has two steps. In the first step data from transferred from process 1 to process 2. Then, process 2 transfers the received data to process 3 in the second step. Existing non-blocking primitives in MPI do not give the flexibility to the implementation to efficiently overlap this kind of communication pattern as there is a data dependency between successive steps. Figure 1 shows the timeline of the progression of events on CPU and HCA of process 2 for three cases. In the first case, the ring communication pattern is implemented using non-blocking MPI point-to-point primitives. In this case after step 1, due to the computation on the CPU, there is a delay in the initiation of the second step at process P2 which impacts the overall time. Our proposed primitives alleviate this issue by providing a set of APIs that will record the entire data-exchange pattern in the beginning, with a call to complete the transfer. This provides the implementation the flexibility to offload any kind of pattern to an external device. Next, we look at the mechanisms to implement our proposed framework.

Existing solutions to offload communication to the DPUs such as [8], [9] provide offload support only for specific communication patterns (MPI_Ialltoall and MPI_Ibcast, respectively) using staged mechanisms. This limits the usability of such solutions for various applications that use basic point-to-point primitives to perform custom communication patterns. Our framework extends the applicability of staged mechanisms for any generic communication pattern, such as point-to-point operations or patterns with dependencies (e.g. ring). However, in staging-based mechanisms, data has to be moved from the host to the DPU and to the destination host process from the DPU. Though the scheme provides overlap, it incurs additional overhead in the communication latency, which significantly degrades performance as shown in Figure 1, case (2). To alleviate this, we propose efficient mechanisms that avoid staging using X-GVMI, which leads to better performance (shown in Figure 1, case (3)).

A. Problem Statement

NVIDIA's BlueField Data Processing Units (DPUs) [10] provide advanced networking capabilities and processor cores that can be used to offload communication operations. Current state-of-the-art DPU-based MPI offload solutions [8], [9] incur significant data movement costs due to the limitation of the DPU processes not having direct access to host memory for RDMA operations. To alleviate this problem, modern NVIDIA SmartNIC DPUs provide a feature called cross-GVMI, which allows host processes to expose their memory regions to DPU processes.

This paper tackles the problem of efficient communication offload to DPUs by proposing a framework that defines a set of *APIs* to express hardware-based offload for any generic communication pattern, and designs to alleviate bottlenecks in using the cross-GVMI *mechanism*. The framework is designed to be programming model agnostic, and to provide perfect overlap with low communication latency for both point-to-point as well as collective operations.

II. Motivation and Challenges

We motivate the need for our proposed framework by discussing the limitations of higher-level programming models such as MPI in expressing offload primitives and the bottlenecks involved in DPU-based offloading techniques.

A. Semantic mismatch between higher-level MPI primitives and DPU-based offload

We show the limitations of higher-level MPI-like primitives in effectively offloading communication patterns by taking a simple ring-based exchange code similar to the communication pattern found in High-Performance Linpack (HPL) Code (see Section VIII-D)

```
1
2  int next, prev;
3  next = (my_rank + 1)% num_ranks;
4  prev = my_rank - 1;
5  if(rank == 0) {
6     MPI_Isend(..next, &req);
7     while(!complete) {
8        do_compute();
9        MPI_Test(&complete, req);
10    }
11 } else {
12    MPI_Irecv(..prev, &req);
13    while(!complete) {
14       do_compute();
15       MPI_Test(&complete, req);
16    }
17    //Received data from left, now send
18    //it to my right neighbor
19    complete = 0;
20    MPI_Isend(..next, &req);
21    while(!complete) {
22       do_compute();
23       MPI_Test(&complete, req);
24    }
25 }
```

Listing 1. Ring Bcast Example

Listing 1 provides a pseudo-code for a simple ring pattern where the data from process/rank 0 broadcasts to other processes in a ring-like exchange as explained in figure 1. This type of communication pattern has a data dependency in each step. Therefore, even with the non-blocking MPI primitives, it is necessary for the application to call MPI primitives in order to progress the communication. In the Listing 1 a rank receives data while the application is busy doing computation in line 14, and the data transfer gets delayed as shown in case 1 of Figure 1. The transfer is started only when the code reaches line 20. This kind of CPU intervention is required with the existing MPI's APIs because of the ordering of the steps forced by the communication pattern. This brings us to our first challenge: **How do we design a generic set of APIs for offloading any communication to DPUs that does not force the requirement of the CPU's intervention even with ordered communication patterns such as the ring?**

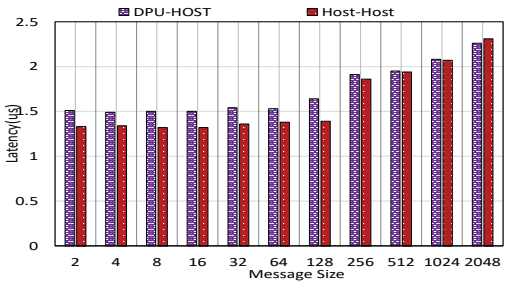


Fig. 2. RDMA-Write Latency of Host-to-Host versus Host-to-DPU

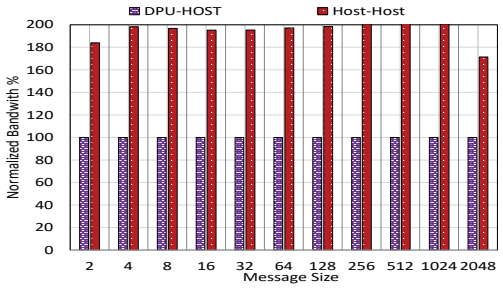


Fig. 3. RDMA-Write Bandwidth of Host-to-Host versus Host-to-DPU. Normalized with respect to Host-to-Host, higher is better.

B. Synchronization overheads between Host and DPU

The offload of collective operations requires the exchange of synchronization messages and metadata (such as the buffer addresses, registered keys, sizes of data, etc.) between the Host and DPU processes. Figures 2 and 3 show the latency and bandwidth of host-to-host transfers versus DPU-Host transfers. Though the DPU-Host numbers shown here are between a local Host and a remote DPU, the numbers remain the same for local Host-DPU-based transfer. We observe that while the latency of DPU-Host transfers is close to Host-to-Host transfers, Host-to-Host transfers have close to twice the bandwidth of DPU-Host transfers. This can be explained by the slower ARM cores (relative to the host) on the current generation of NVIDIA DPUs, since the bandwidth of smaller messages (their injection rate) is sensitive to the frequency of the processor. This brings us to the second challenge: **Can we design APIs and mechanisms to minimize the overheads involved in synchronization and metadata exchange between the Host and the DPU for various communication patterns?**

C. Registration and other data movement overheads

Authors in [8], [9] proposed collective algorithms using DPUs for offloading communication operations. Figure 4 shows results of a simple ping-pong benchmark that contains non-blocking sends/receives followed by an MPI_Waitall operation. The brown bars show the default host-to-host latency for different message sizes. The green bars show the latency of using a staging-based design that uses DPU memory as an intermediate memory to transfer data between two hosts. The figure shows a degradation for staging-based transfers when compared to direct host-host transfers. Cross-GVMI aims to fix this by allowing the DPU to perform transfers on behalf of the host. However, it has a few caveats.

For a DPU process to transfer data with cross-GVMI, two kinds of memory registration must happen. First, the source buffer must be registered by the host process. Second, the DPU (or worker) process must use the same source address/key from the first registration to perform another registration operation. This registration needs to be performed every time a send-receive operation occurs. Section V explains this process in further detail. The costs of registration are quantified in figure 5. These overheads are significant and will hurt application/benchmark performance if not handled correctly. Standard MPI libraries use a registration cache to amortize the cost of memory registration which is useful when requests have a repetition of buffer addresses. Traditional registration caches cannot be used here as they maintain only local buffer information on the host. With cross-GVMI, registration occurs on both the host and the DPU. Moreover, the second registration not only depends on the buffer address but also requires additional parameters from the first registration. This requires designing registration caches that work on both the host and the DPU, as well as efficient synchronization between the designed registration caches to maintain correctness and achieve good performance. This brings us to the third challenge: **How can we design an efficient registration cache for cross-GVMI transfers that amortizes various overheads and gives performance comparable to the host?**

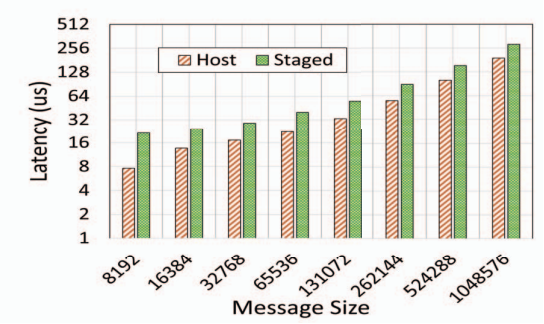


Fig. 4. Communication Latency for Nonblocking Pingpong (Concurrent two-way isend/irecvs) using Host and Staging-based MPI Designs

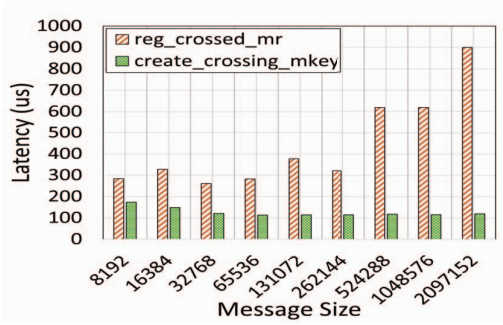


Fig. 5. The overhead time of two kinds of memory registration for a DPU process to transfter data with cross-GVMI

III. Contributions

We propose a framework that offloads communication operations to the DPU (from the host). Our framework is designed

to tackle two fundamental shortcomings of state-of-the-art solutions: 1) Semantic mismatches arising from higher-level abstractions that lead to sub-optimal offload for certain communication patterns. 2) Avoid staging overheads between DPU and host, and other associated overheads involved in cross-GVMI-based transfers. To achieve these, we propose a set of primitives (APIs) that allows the offload of any communication pattern, and designs to amortize various overheads associated with DPU-based transfers. We implement our framework inside a production MPI library to demonstrate the efficacy of our designs. We make the following key contributions:

- Identify limitations of higher-level programming models such as MPI in expressing the offload of complex communication patterns
- Analyze synchronization and data movement bottlenecks for generic point-to-point and collective transfers offloaded to the DPU
- Propose a framework with APIs to conveniently express the offload of generic communication patterns to the DPU
- Propose basic and optimized designs to implement the APIs
- Demonstrate the efficacy of the proposed designs on real systems using micro-benchmarks and applications. Our framework outperforms state-of-the-art solutions by up to **47%** in ialltoall micro-benchmarks and up to **60%** and **15%** in the P3DFFT and HPL respectively.

IV. Background

To further understand overlap and offload in the context of MPI, we need to understand MPI semantics, implementation, and underlying hardware/capabilities.

MPI provides both blocking and non-blocking semantics. Blocking semantics dictate that, on the sender's side, the buffer can be re-used after returning from MPI_Send, and for the receiver, the data is available in the buffer after a call to MPI_Recv returns. Non-blocking semantics provides MPI_Isend and MPI_Irecv, which return a request object, and MPI_Wait which takes in a request object to ensure that the request is complete. After returning from MPI_I* functions, MPI_Test can be used with the same request object to progress the communication and check for completion.

InfiniBand [11] is one of many popular network-level interconnects supported by a large number of MPI libraries. Infiniband requires memory regions to be registered with the HCA before they can be used for data transfers. Every memory registration with the HCA using the ibv_reg_mr function returns an "lkey" and "rkey". For any RDMA operation (READ or WRITE) posted local on a buffer requires an "lkey" returned from the buffer registration. For a process to perform any RDMA operation on a remote buffer, an "rkey" of the remote buffer is needed. This is why MPI processes typically exchange rkeys of buffers before performing actual data transfer.

V. Overview of Cross-GVMI

For a DPU process to transfer data from one host process to another, the data has to be read from the source host process' memory to the DPU's memory, and then the data is written to the destination host's memory. This mechanism is called staging. However, GVMI enables a DPU process to transfer data from a process on the local host machine to any process on a remote host. Figure 6 contrasts the steps involved in transferring data from one node (NODE 1) to the other (NODE 2) using GVMI and staging mechanism. We observe that a staged transfer requires an additional movement of data to NODE1's DPU's memory which is not required in a GVMI-based transfer.

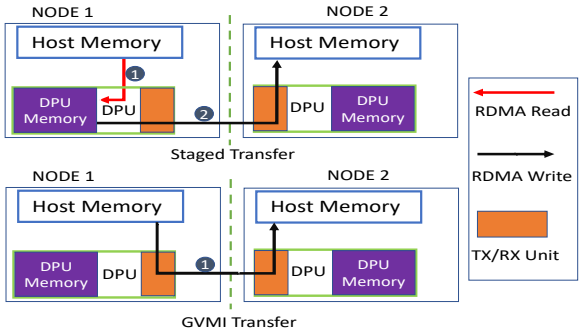


Fig. 6. Data transfer path for staged and GVMI-based transfer. Staged suffers from the overhead of an extra RDMA-write to DPU

First, a DPU process on NODE1 generates a GVMI-ID (done only once per protection domain) and sends it to the host process on NODE1. The host process then registers the source buffer with this GVMI-ID. After registration, the host process sends the source address, buffer size, and mkey (obtained from registration) to the DPU process. Next, the destination process on host-2 registers the destination buffer and sends the buffer information to the DPU process, which registers the source buffer to obtain mkey2. This process is called "cross-registration". It requires the source buffer address, mkey (obtained from the source process), buffer size, and source GVMI-ID. Finally, after generating mkey2, the DPU process does an RDMA-Write to the remote process. For the final RDMA-Write, the DPU process uses mkey2 as the lkey and all other parameters do not change. The final mkey2 is analogous to an lkey in a normal InfiniBand transfer as described in Section IV. However, now the mkey2 is used to transfer data on behalf of a host process' local buffer.

VI. Designing APIs for expressing generic communication patterns

In this section, we address the challenge mentioned in section II-A by proposing a set of APIs. We first propose Basic primitives which are required to support basic transport operations such as sends and receives available in any communication library. Then we propose Group primitives which provide APIs to completely offload any kind of communication pattern which cannot be expressed with current communication models such as MPI, and OpenSHEM.

A. Proposed Primitives

Listing 2 presents the set of proposed basic primitives. The simple primitives are largely parallel to MPI point-to-point primitives. Listing 3 explains how to write a simple

ping-pong program with the Basic Primitives and MPI. The `Init_Offload` function initializes the library by creating and assigning ranks to all the processes. `Send_Offload` and `Recv_Offload` functions are two-sided non-blocking functions that offload the sends and receives to the DPU. The `Wait` function is for the calls to complete. `addr` is the buffer address from/to which the data is moved when `Send_Offload`/ `Recv_Offload` primitive is invoked. The `dst` and `src` fields specify the destination and source host ranks involved in the data transfer. For every `Send_Offload` there should be a matching `Receive_Offload`. `tag`, `dst`, and `dst` fields are used for matching the sends and receives. `req` is the identifier for a particular transfer that can later be used in the `Wait` function to complete the transfer.

B. Group Primitives

Listing 4 lists the set of APIs in Group Primitives. A Groups Primitives-based implementation of the ring pattern described in Listing 1 is shown in Listing 5. Any algorithm using group primitives must begin with `Group_Offload_start`, which returns a request object. Any subsequent calls to `Send_*` or `Recv_*` functions must use the same request object. After all the calls are made, `Group_Offload_end` must be called to mark the termination of the communication pattern. Then `Group_Offload_call` must be used to offload the entire communication graph that is recorded by the request object. To enforce ordering between local transfers, we introduce `Local_barrier_Goffload` which ensures that any operations after the barrier will begin only after the completion of all the operations before it. This operation is local to a process. This operation cannot be done in a nonblocking manner with the MPI APIs. Therefore, the group primitives provide users the flexibility to implement any kind of communication pattern with the goal of complete non-blocking semantics which will allow the implementation to offload the entire pattern to DPU. In addition, it also makes it convenient for the implementation to provide optimizations which are described in a later section. Lines 19 to 21 in Listing 5 show how we offload the entire ring communication pattern to the DPU and achieve overlap without the CPU intervention unlike the ring code shown in Listing 1.

```

Init_Offload();
Send_Offload(void* addr, size_t size,
             OffloadRequest *req,
             int dst_rank,
             int tag);
Recv_Offload(void* addr, size_t size,
             OffloadRequest *req,
             int src_rank,
             int tag);
Wait(OffloadRequest req);
Finalize_Offload();
```

Listing 2. Basic Primitives

```
//Ping-Pong with Basic Primitves
Init_Offload();
void *sbuf, *rbuf;
size_t size = 1024;
OffloadRequest *req;
Send_Offload(sbuf,size,&req, 1, 3);
Recv_Offload(rbuf,size,&req,1,3);
Wait(&req);
Finalize_Offload();

//Ping-Pong with MPI
MPI_Init();
void *sbuf, *rbuf;
size_t size = 1024;
MPI_Request req; MPI_Status st;
MPI_Isend(sbuf,size,...);
MPI_Irecv(rbuf,size,...);
MPI_Wait(&req, st);
```

Listing 3. Ping-Pong Example with Basic Primitives

```

Send_Goffload(void* addr, size_t size,
              OffloadGroupRequest *req,
              int dst_rank,
              int tag);
Recv_Goffload(void* addr, size_t size,
              OffloadGroupRequest *req,
              int src_rank,
              int tag);
Local_barrier_Goffload(OffloadGroupRequest *req);
Group_Offload_start(OffloadGroupRequest *req);
Group_Offload_end(OffloadGroupRequest *req);
Group_Offload_call(OffloadGroupRequest *req);
Group_Wait(OffloadGroupRequest *req);
```

Listing 4. Group Primitives

```
//Bcast with Ring Exchange
Init_Offload();
void *buf;
OffloadRequest *req;
left = (rank - 1 + numProcs) % comm_size;
right = (rank + 1) % numProcs;
Group_Offload_start(&req);
if(rank == 0) {
    Send_Goffload(sbuf, bufsize, &req, right, 4);
    Local_barrier_Goffload(&req);
} else {
    Recv_Goffload(rbuf, bufsize, &req, left, 4);
    Local_barrier_Goffload(&req);
    Send_Goffload(rbuf + idx_send * bufsize, bufsize
    , &req, right, 4);
}
Group_Offload_end(&req);
//Overlap with compute
Group_Offload_call(&req);
do_compute();
Group_Wait(&req);
```

Listing 5. Ring Pattern Example with Group Primitives

VII. Designing Efficient Mechanisms using DPU-based Offload for Host-Host Transfers

This section describes the underlying mechanisms used by our proposed APIs and designs to solve challenges described in Sections II-B and II-C. In our offload framework, we launch a set of processes on the DPU that are referred to as proxy or worker processes. These processes perform the data transfer on behalf of the host processes. This is how the host communication is offloaded to the DPU processes.

A. Implementation of Basic Primitives

The `Init_Offload()` method launches worker processes and assigns ranks to each of them similar to MPI ranks.

As described in section V, for GVMI-based registration, a DPU process needs the remote host process' GVMI-ID. Since the GVMI-ID does not change for a process, it gets generated inside `Init_Offload()` and exchanged with all

other processes in the global communicator, which includes all processes on the host and DPU.

Figure 7 lists the steps of RDMA-Write-based implementations of `Send_Offload()` and `Recv_Offload()` on the host's side. After choosing the proxy process, the sender does a GVMI-based memory registration of the source buffer with the proxy process' GVMI-ID to generate a mkey. A host process is mapped to a proxy process on the DPU in the same Node. Mappings are calculated using `proxy_local_rank = host_source_rank % num_proxies_per_dpu`. The receiver does normal InfiniBand-based registration of the destination address to generate rkey. Then sender and receiver processes send Ready-To-Send (RTS) and Ready-To-Receive (RTR) control messages to the chosen proxy process with the necessary information such as buffer addresses, mkey/rkey, buffer sizes.

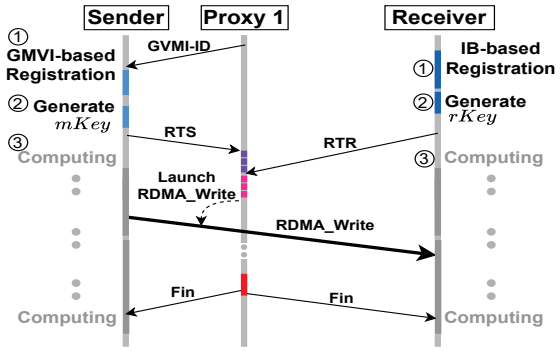


Fig. 7. Generation of GVMI's mKey and IB-based rKey used in setup for control-messages and RDMA-Write-based implementation of Send/Recv_Offload()

Figure 8 provides a sequence of events that occur in the DPU handler when RTR and RTS packets are received. As shown in Figure 8, a proxy process has two sets of request queues: one is for send requests, which has the data provided by the source host process that includes the source address, data size, mkey, and source request ID; the other is for receive requests, which contains destination information: size, rkey, destination request ID. Since multiple source and destination ranks can send control messages to a given DPU process, the DPU process maintains a list of request queue headers which are ordered by the destination rank number. In Figure 8, once the RTS from 1 arrives, it searches for a matching request in the receive queue. Since it does not find one, it adds it to the send queue. Then the RTR arrives, which searches the send queue and finds a match. Then the request is removed from the send queue and added to the combined queue. After this, the requests from the combined queues are processed. Once the RDMA operations are complete the FIN packets are sent to the host processes.

B. Optimizing GVMI registration and exchange Overheads

In this section, we outline the optimizations for the Basic Primitives implementations described in the previous section. As described in Section V, GVMI involves two types of registration: one which is done at the host process, and the

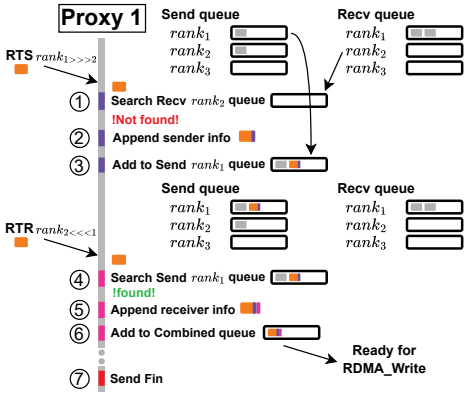


Fig. 8. Proxy Portion of Handling Control Messages on the DPU side Using GVMI.

other which is done at the proxy process. Each of these uses a different set of input parameters. Source address, data size, and remote GVMI-ID are the parameters needed for registration on the host side. This generates an mkey which is sent to the chosen DPU process. Since remote GVMI-ID depends on a remote rank, the registration cache needs to use the source address, data size, and remote-rank as the keys. The DPU process uses a source address, data size, remote GVMI-ID, and remote mkey (obtained from the host process) for registration.

Though there are two additional parameters (GVMI-ID, mkey) for the registration cache, we can still use source address, data size, and remote rank as the key to the registration cache in the DPU process; this is because the value of the mkey depends on the source address and source data size, so it is impossible to have multiple values of the same mkey given a source address and GVMI-ID. Similarly on the DPU side, for every host rank there are only distinct entries of the set (source-address, data-size, mkey) provided by the source since mkey depends on source-address and data-size. Therefore, one can again use source-adress, data-size, and rank as the parameters for the DPU's GVMI-cache.

Based on this, we use an array of Binary Search Trees to represent the registration cache of both the host and DPU sides. The array is indexed by remote rank and the BST is indexed by memory address. The cache has an array at the first level and a BST at the second level. An array is used at the first level because there is only a finite number of ranks allowed in a communicator. When a host process attempts to register a mkey using source address, data size, and GVMI-ID, the array of BSTs is queried using the mapped DPU proxy's rank as the key to obtain a reference to the correct BST for it. Then the BST is queried using the address and size to obtain a cache entry that contains the mkey, GVMI-ID, address, and data size.

On a cache hit, the entry is returned. On a cache miss, a new cache entry object is created, the host's registration function is called, and the cache entry is updated with the object's mkey and inserted to the BST corresponding to its mapped DPU proxy's rank. On the DPU side, when a RTS comes, similar set of querying happens with source-address and host-process' rank provided in RTS.

C. Implementation of Group Primitives

Here, we will introduce Group Primitives as a way of improving upon Simple Basic Primitives.

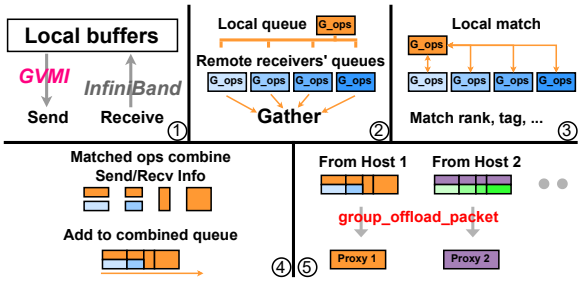


Fig. 9. Steps shown when performing Group_Offload_call between GVMI and InfiniBand, including send/receive, gathering, rank/tag matching, queue insertion, and offloading to proxy processes.

When a `Group_Offload_start` function is invoked, a new request object is created and returned. Any subsequent calls to Group_* functions with a given request object will create a new `Group_op` structure and add all the local and remote buffer details to it. This will then be added to a queue, where the request will have the head pointer to that queue.

Figure 9, lists the steps involved in the host process when `Group_Offload_call` is invoked. Local buffers here are the buffer addresses (`addr` field) obtained from group primitive calls. First, the send buffers are registered using GVMI cache, and receive buffers are registered using IB registration cache. Then `Group_op` queue is gathered from all receiving processes. After this, each `Group_op` send entry from the source queue is matched with a matching receive entry in the remote queues based on destination rank, tag. Then the queue of matched entries is added to a `Group_Offload_packet` and sent to the corresponding proxy process that is mapped to the host process. `Group_Offload_packet` contains `Group_op` queue, host-rank, request-ID obtained from `Group_Offload_call`.

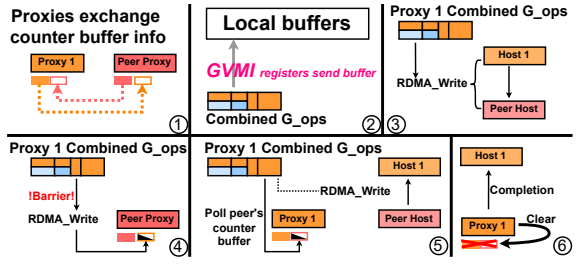


Fig. 10. Steps shown when performing `Group_offload_packet` handling, including RDMA-based operations and updating of buffer counters.

Figure 10 shows at a high level the steps involved in the proxy side when a `Group_Offload_packet` is received. First, all the proxies exchange a pre-registered buffer counter which is used for implementing barrier. Then for each entry in the received queue, if a send operation is encountered the sender buffer is registered with GVMI cache to generate the final mkey. Then RDMA write is posted. If a barrier is encountered, the sender writes the barrier count to its remote peer's counter buffer. Then it polls for the local barrier counter if any receive operation was posted. Only after the expected barrier counter is read, the next entry is processed. This process repeats until all the entries are processed. Once all entries are processed, the completion counter on the host memory is updated using RDMA-write and the barrier counters are cleared

When a `Group_Wait` is called, each host process waits until the completion counter associated with the given request object is set. Once the completion counter is set, the function returns.

The barrier-counters are needed because each worker process must know the receive completion progress of its locally mapped host process.

D. Optimizing Group Primitives

Here, we take our baseline implementation and show how we were able to optimize our Group Primitives, by including caches to reduce host-to-DPU control message exchange.

We introduce caches in the host and worker processes to reduce the overhead of control messages and registrations. On the host side, the cache is indexed by the request ID and proxy rank. Each entry of the cache contains the buffer addresses, entries of the GVMI and IB registration caches, and a flag indicating whether request details were sent to the proxy rank. On the DPU side, a cache is created which is indexed by the host's request ID and rank. If the host detects a cache hit, it sends the request ID to the DPU. The DPU queries the cache to retrieve the corresponding `Group_op` entry queue. In addition to this information, the group entry queue also contains the GVMI registration cache entry. This way, the DPU process is saved from searching the GVMI cache for each Group_op entry.

Algorithm 1 concisely describes the steps performed by the proxy process when a `Group_Offload_packet` is received and it is a cache hit. The `PostCachedEntryOps` iterates over the Group_op entry objects and issues send RDMA operation if a send is encountered. If a receive entry is encountered, the source host rank is added to `recvRankSet` hashset. Similarly, the distinct destination ranks are recorded after posting send operations. Once a barrier entry is encountered, we increment the `numBarriers` counter variable.

After all the preceding sends are completed, the value of `numBarriers` is written to all proxy process' barrier counters which are mapped to the list of destination ranks present in the `sendRankSet` hashset. The local barrier counter is then polled for the value of `numBarriers` from all the proxy ranks mapped to the ranks present in the `recvRankSet`. If the counter values are not found, then the request is marked incomplete and the code returns to the progress engine. This is needed because the proxy process could be handling multiple host processes. Therefore, to avoid deadlock in the event of one proxy waiting for the receive that is posted by the same proxy, we need to break from the function to the progress engine to progress other requests as well.

Algorithm 1: High-Level Proxy Side Algorithm to Process GroupOffload packet

```
Function PostCachedEntryOps(cacheEntry):
    foreach curOpEntry in cacheEntry->GroupOpEntries do
        if curOpEntry->type == SEND then
            PostRDMAWrite(curOpEntry)
            sendRankSet.add(curOpEntry->dstRank)
        else if curOpEntry->type == RECV then
            recvRankSet.add(curOpEntry->srcRank)
        else if curOpEntry->type == BARRIER then
            cacheEntry->numBarriers++
            writeRemoteBarrierCntr(sendRankSet, cacheEntry)
            sendRankSet.clear()
            if isRecvBarrierDone(recvRank, cacheEntry) == False then
                break;
            else
                recvRankSet.clear()
```

VIII. Experimental Evaluation

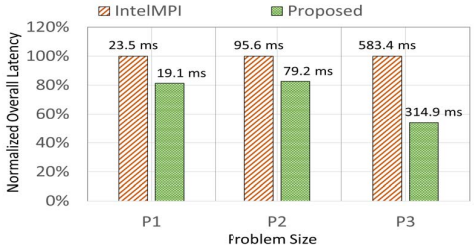


Fig. 11. 3DStencil Normalized Overall Time for Proposed GVMI and Intel MPI, running with 16 Nodes 32 PPN

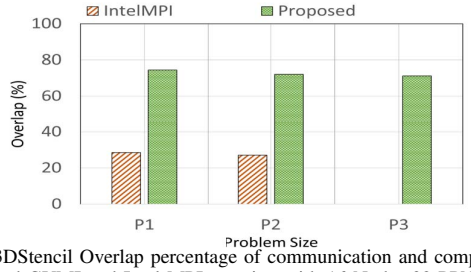


Fig. 12. 3DStencil Overlap percentage of communication and compute time for Proposed GVMI and Intel MPI, running with 16 Nodes 32 PPN

Our experimental platform has 32 nodes, each equipped with a BlueField-2 SmartNIC as well as a separate ConnectX-6 HCA, the Broadwell series of Xeon dual-socket, 16-core processors with 3.40 GHz and 256 GB RAM.

We implemented the proposed schemes in an MPI library. We used basic primitives to implement MPI non-blocking sends and receives. We used Group Primitives to implement non-blocking collectives. In this section, we compare the performance of the proposed APIs implemented in our base MPI library with the following MPI runtimes: Intel MPI 2021 (IntelMPI), BluesMPI framework. We would like to point out that BluesMPI is a DPU-offload-support-based MPI library that uses a staging mechanism to offload `MPI_Ialltoall`.

For micro-benchmark-level evaluation, we utilize OMB [12] to show the benefits of our Group Primitives over the state-of-the-art DPU offload framework BluesMPI. Each test was run for 100 iterations and an average of 3 runs is reported. For evaluating Simple Primitives, we use an in-house 3D-Stencil Overlap benchmark. For application-level results, we compared the performance of a non-blocking version of P3DFFT [13] which overlaps the computation with MPI_Ialltoall communication and HPL 2.3 with ring-based broadcast.

A. Basic Primitives Evaluation with 3DStencil Benchmark

3D Stencil benchmark follows a near-neighbor communication pattern which is common in HPC applications. In this benchmark, each process sends and receives data buffers using MPI_Isend/MPI_Irecv to/from at most 6 neighbors. A dummy compute is overlapped with the communication. The % Overlap is measured in a manner similar to OMB Non-Blocking Collectives. We ran this benchmark for 3 problem sizes: $512 \times 512 \times 512$, $1024 \times 1024 \times 1024$, and $2048 \times 2048 \times 2048$.

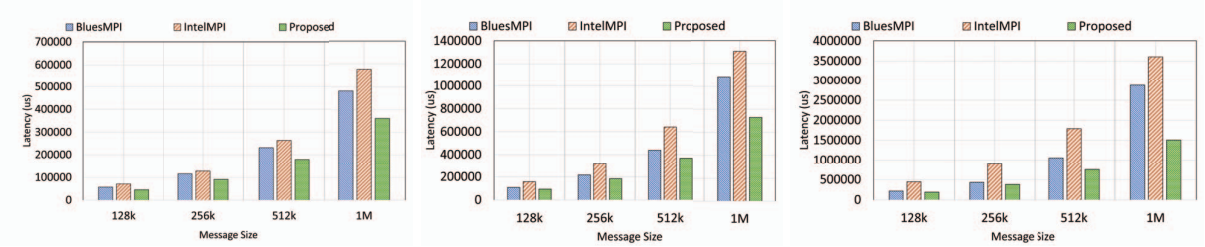
We compared this with IntelMPI default as BluesMPI does not support point-to-point offload. Figure 11, gives the overlap time which has both compute and communication overlapped. We observe that we get more than 20% benefits. Figure 12 shows the percentage overlap obtained for all the cases. We observe that the percentage overlap remains more or less constant at around 78% for the Proposed schemes, however, it drops for IntelMPI for the largest problem size because of which the overall time is also impacted. We note that the Proposed Scheme's overlap is not 100% because the intra-node MPI transfers do not use our Simple Primitives for offloading, therefore they are more or less blocked by the CPU.

B. Group Primitive Evaluation with MPI_Ialltoall

We implemented a scatter-destination Algorithm using Group Primitives in MPI_Ialltoall. Figure 13 shows the overall time for MPI_Ialltoall which includes communication and compute time on 4, 8, and 16 nodes with 32 processes per node. On 4 nodes, the Proposed scheme performs up-to 25% better than BluesMPI, and 35% better than IntelMPI. On 8 nodes, we get 40% improvement compared to IntelMPI and up-to 30% improvement compared to BluesMPI. As we scale up to 16 nodes, we achieve up to 58% improvement compared to IntelMPI and 47% improvement compared to BluesMPI. The primary reason for the improvements compared to BluesMPI is the benefits from the absence of staging, and the caching design contributes to further enhancement. This is corroborated by the communication latency where GVMI outperforms BluesMPI for all scales. Figure 14 shows the percentage overlap for MPI_Ialltoall on 4,8,16 nodes with 32 processes per node. We observe that both BluesMPI and the Proposed scheme provides close to a 100% overlap. Since the Proposed scheme's communication time is superior, it outperforms BluesMPI.

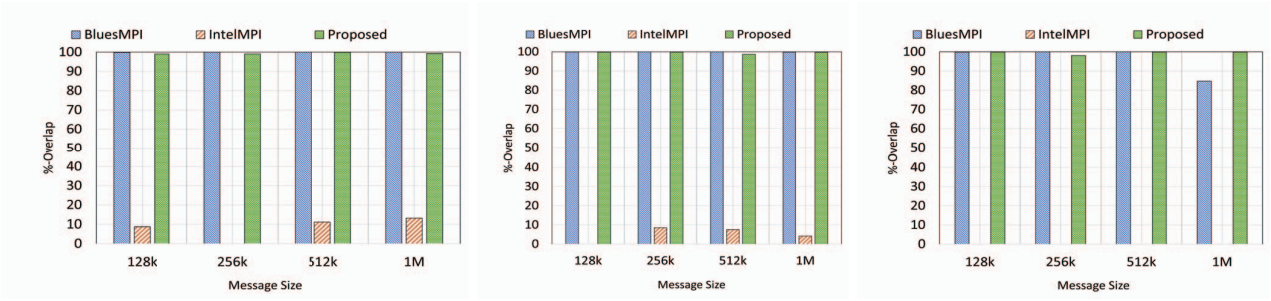
C. Benefits of Group Primitives for Dense Patterns

In Figure 15, we show the performance of a simple scatter-destination pattern for a personalized alltoall exchange on 8



(a) 4-Node Overall Time (b) 8-Node Overall Time (c) 16-Node Overall

Fig. 13. Overall time for MPI_Ialltoall communication and compute, with BluesMPI, Proposed, and IntelMPI on 4, 8 and 16 Nodes with 32 PPN



(a) 4-Node %-Overlap (b) 8-Node %-Overlap (c) 16-Node %-Overlap

Fig. 14. Percentage overlap for MPI_Ialltoall communication and compute, with BluesMPI, Proposed and IntelMPI on 4, 8 and 16 Nodes with 32 PPN

nodes/32 PPN. We implemented it using a) Simple Primitives and b) Group Primitives. We observe that the Group Primitives based implementation gives an improvement of up-to 40% over the Simple Primitives based implementation. In the Simple Primitives version, for each send-recv transfer from host-to-host, four control messages are exchanged between the host and DPU: two for RTS and RTR during initialization and two for FIN packets at the end. However, in the Group Primitives

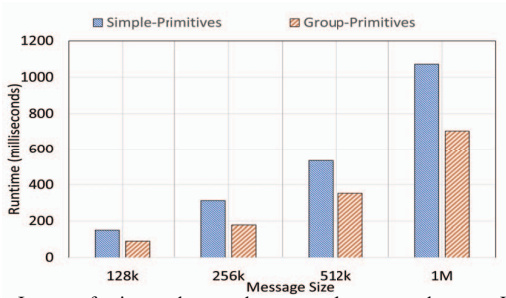


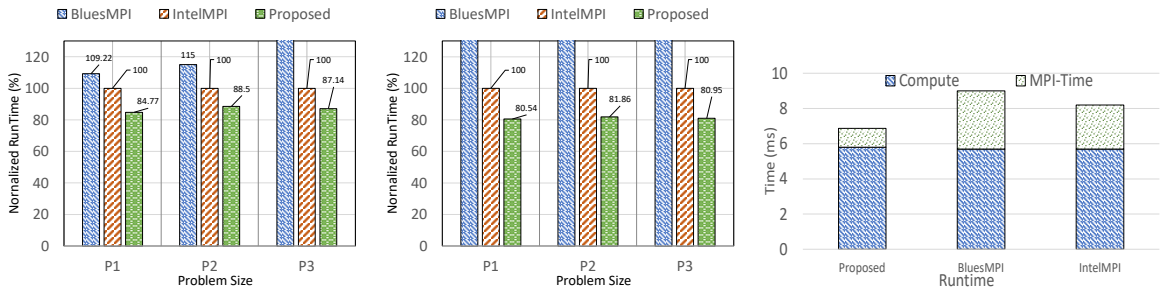
Fig. 15. Impact of using cache to reduce control messages between Host and DPU on 8 Nodes 32 PPN for Scatter-Destination Pattern

case, first all the destination buffers needed for a proxy process are gathered by the host process. Then a single contiguous message is sent to the proxy process by each host process. This method of gathering data from the Host is advantageous because the RDMA operations performed by the host are superior in performance compared to those performed by the DPU as shown in section II-B. Another advantage of Group Primitives is that thanks to the cache design all the meta-data (e.g. receive buffer addresses, rkeys), is exchanged only once and thus avoids the expensive Host-to-DPU transfers for exchanging the meta-data. This is useful from an application standpoint because many applications tend to exhibit temporal locality when it comes to buffer addresses being used in a group communication pattern.

D. Application Evaluation

PD3FFT: In this section, we evaluate the impact of the MPI Collectives implemented with the proposed Group framework on performance of the Parallel Three-Dimensional Fast Fourier Transforms (P3DFFT) application [13]. This library uses a 2D, or pencil, decomposition and increases the degree of parallelism and scalability of FFTs. The data grid is transformed during each iteration using nonblocking Alltoall collectives. We used test_sine.x program for our evaluation. In this program, given an input grid size in X,Y,Z it performs forward and backward Fourier transforms through pencil decomposition. In Figure16(a) we show P3DFFT application runs on 8 nodes with 32 PPN. We fixed X and Y to 256 and increased Z from 512 to 2048. We observe up to 16% improvement compared to IntelMPI and up to 55% improvement compared to BluesMPI. Figure 16(b) shows P3DFFT runs on 16 nodes with 32 PPN in which we fix X and Y to 512 and vary Z from 1024 to 4096. Here we observe up to 20% improvement compared to IntelMPI and up to 60% improvements compared to BluesMPI. The improvement over BluesMPI mainly stems from the overall latency improvement of our proposed schemes. This is attributed to the caching and direct GVMI-based transfer designs in the group offload framework.

To understand the relative performance of MPI runtimes, we profiled the P3DFFT application to calculate the time spent in compute and MPI for a single iteration of the forward phase as shown in figure 16(c). We found that the computation loop initiates two MPI_Ialltoall calls with different buffers of sizes 65KB for Problem size P1. It then performs some computation, waits for one call to complete. The loop then repeats the process by performing further computation before waiting for another call to complete. At any point, the computation phase



(a) P3DFFT 8N-32PPN (b) P3DFFT 16N-32PPN (c) P3DFFT Profile

Fig. 16. Normalized Application Run-Times and profile for P3DFFT. P3DFFT values are normalized with IntelMPI. Figure 16(c) shows the runtime of a single phase of the forward FFT. Compute and Time spent in MPI are shown for 8 Nodes 32 PPN run for problem P1. (Lower is better.)

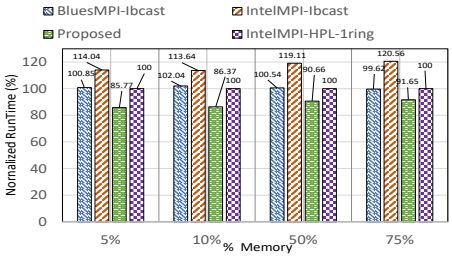


Fig. 17. Normalized values of HPL total runtime for different % memory sizes running on 16 Nodes 32 PPN. Values are normalized with respect to IntelMPI-HPL-1ring. (Lower is better)

can be overlapped with 2 MPI_Ialltoall calls. The computation time was the same for all 3 libraries. BluesMPI spent the most time in MPI_Wait. This differs from benchmark-level measurements, where BluesMPI outperforms IntelMPI. When we measured the pure-communication time of BluesMPI with 2 back-to-back MPI_Ialltoalls with different buffer sizes in P3DFFT, we found that BluesMPI has a lot of degradation in performance compared to IntelMPI for the first several iterations. Since these libraries are closed-source, it is difficult to concretely define the cause of this degradation demonstrated at the application layer by evaluating the MPI implementation. However, one difference between the application level and benchmark level evaluations is the use of warm-up iterations. At the benchmark level, degraded performance is hidden through the use of multiple warm-up iterations and evaluating a single MPI_Ialltoall call, whereas at the application level with no warm-up iterations, there are two back-to-back MPI_-Ialltoall calls with different source and destination buffers. We determined this effect of warm-up by evaluating BluesMPI at the benchmark level without utilizing any warm-up iterations and witnessed poor performance for the first few evaluations of the BluesMPI implementation of the MPI_Ialltoall call.

HPL: HPL [14] is a portable High-Performance implementation of Linpack — the standard for ranking the world's TOP500 supercomputers. The code records the time required for LU factorization of a dense matrix. Based on the time, it calculates the rate of floating point operations of the system. HPL follows a 2-D block decomposition strategy for load balancing. The total number of processors is split into a PxQ grid. The broadcast operation is used in HPL to forward a panel for the factorization of blocks owned by individual processors.

The existing HPL application implements ring-based algorithms using MPI point-to-point primitives which is called the HPL-1ring algorithm. The HPL code attempts to overlap this broadcast with independent computation via a look-ahead strategy. The overlap portion is roughly similar to the code shown in the Listing 1. Therefore, it requires CPU intervention to check the arrival of the message. This would be an ideal candidate for our proposed offload framework as it will use DPUs to progress communication without CPU intervention.

We modified the HPL code to add support for MPI_Ibcast. Since broadcast algorithms are implemented with point-to-point operations, we can compare our proposed design with BluesMPI which does not support point-to-point offload. We also compare with IntelMPI's HPL-1ring. In addition, we also compare with IntelMPI's MPI_Ibcast so that we can compare with Intel-MPI's best Ibcast Algorithm. We have added all possible combinations of libraries to make the comparison fair. We ran the problem of sizes that occupied 5% to 75% of the system memory (256GB) which is the standard for HPL. We observed that using our proposed scheme results in a runtime of approximately 18% lower than using Intel-MPI's Ibcast, 15% lower than using BluesMPI on 5% and 10% problem sizes. Intel 1-Ring roughly performs similarly to BluesMPI for all problem sizes. Our proposed scheme's runtime increases to 90% compared to IntelMPI-HPL-1ring on 50 and 75% memory inputs. We believe this is due to the overheads caused by large Ibcasts-based transfers using GVMI. Nevertheless, our proposed scheme manages to perform at least 8.5% better than IntelMPI which saves roughly **13 minutes** of the execution time for 75% memory input. The reason the proposed scheme outperforms BluesMPI is due to 1) the absence of the staging overhead and 2) the optimized cache designs. The HPL-1Ring has delays caused by the need for the CPU to initiate the next phase of the ring. Intel's Ibcast also requires the CPU's intervention to progress the transfers using MPI_Test which reduces the overlap potential.

IX. Related Work

A previous work that leverages DPUs to offload nonblocking collectives is BluesMPI [8], [9]. In this work, DPUs were used to offload nonblocking collectives to achieve 99%

overlap of communication and computation. In BluesMPI, however, its offload functionality is tightly coupled with MPI, is algorithm-specific, and does not cover other communication patterns. The semantic issue is also not addressed by this work. BluesMPI also incurs a staging overhead between the host and the DPU, due to the fact that the DPU could not directly issue RDMA operations on behalf of the host. Our paper alleviates these issues with carefully designed APIs and efficient usage of the cross-GVMI mechanism. Floem [15] is a framework to ease the development of applications that offload work to SmartNICs. It includes a language, compiler, and runtime that allows developers to define "elements" of C code that are executable on SmartNICs. This project differs from ours in that it focuses on offloading computation rather than communication. Another offloading framework, iPipe [16], implements an actor-based programming model that allows developers to easily offload complex algorithms that were not previously synthesizable on older, FPGA-based SmartNICs. This work differs from ours in its focus on offloading computation and packet processing rather than point-to-point communication primitives. Other projects focusing on specifically offloading MPI communication include NVIDIA's SHARP [5]. SHARP leverages NVIDIA switches to execute in-network collectives, specifically Reduce and Allreduce. This significantly decreases the time spent executing these collectives compared to implementing them on the host processor. However, it does not allow offload of nonblocking communication nor point-to-point communication, nor does it enhance MPI primitives beyond All/Reduce. A similar project to SHARP is MPI_FPGA [17]. This library focuses on using in-switch FPGAs to implement multiple collective algorithms in reconfigurable logic. However, this work does not augment nonblocking communication or point-to-point communication. Past works [18], [19] designed nonblocking collectives that utilize a communication thread. However, this inhibits the use of one of the CPU cores. Other works [4] reduce the effect of this asynchronous progress thread, but fail to achieve the overlap reached by GVMI.

X. Conclusion and Future Work

Asynchronous communication and overlap is an important field of research within the HPC community. Many applications still fail to leverage non-blocking collectives due to sub-optimal designs. Since DPU technology has enabled new ways of offloading tasks, work has been done to leverage this device in the context of progressing communication. Despite previous efforts, state-of-the-art designs proved to be inflexible. Previous works utilizing DPUs to offload computation were limited to a few collective operations and locked to a single programming model. In this paper, we identify the limitations of these higher-level designs. We identify bottlenecks in point-to-point and collective communication patterns and design a generic framework for offloading communication to the DPU. We develop two primitives as part of our design. One primitive provides basic point-to-point functionality while the second enables optimized group communication patterns. We describe initial implementations of these primitives and detail further optimizations. Our design also leverages NVIDIA's new cross-GVMI capability that enables the DPU to access host memory. This functionality is important in allowing the DPU to do RDMA reads and write on behalf of the host. With these advances, we are able to outperform state-of-the-art designs by 47% in ialltoall, 60% in P3DFFT and up-to 15% in HPL. In future work, we would like to experiment with next generation BlueField-3 SmartNICs and Infiniband NDR interconnects to enhance our designs.

References

[1] 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-datasheet.pdf

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

[3] AMD, "Amd epic server processors for data centers." [Online]. Available: https://www.amd.com/en/products/epyc-server

[4] A. Ruhela, H. Subramoni, S. Chakraborty, M. Bayatpour, P. Kousha, and D. K. Panda, "Efficient asynchronous communication progress for mpi without dedicated resources," in *Proceedings of the 25th European MPI Users' Group Meeting*, ser. EuroMPI'18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3236367.3236376

[5] NVIDIA, "NVIDIA Scalable Hierarchical Aggregation and Reduction Protocol (SHARP)." [Online]. Available: https://docs.nvidia.com/networking/display/SHARPv200

[6] MPI Forum, http://www.mpi-forum.org/.

[7] OpenSHMEM, http://openshmem.org/site/.

[8] 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.

[9] N. Sarkauskas, M. Bayatpour, T. Tran, B. Ramesh, H. Subramoni, and D. K. Panda, "Large-message nonblocking mpi_iallgather and mpi ibcast offload via bluefield-2 dpu," in *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2021, pp. 388–393.

[10] I. Burstein, "Nvidia data center processing unit (dpu) architecture," in *2021 IEEE Hot Chips 33 Symposium (HCS)*, 2021, pp. 1–20.

[11] "InfiniBand Trade Association," http://www.infinibandta.com, 2017.

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

[13] D. Pekurovsky, "P3dfft: A framework for parallel computations of fourier transforms in three dimensions," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C192–C209, 2012. [Online]. Available: https://doi.org/10.1137/11082748X

[14] "HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers." [Online]. Available: http://www.netlib.org/benchmark/hpl/

[15] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson, "Floem: A programming system for nic-accelerated network applications," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, p. 663–679.

[16] M. G. Liu, A. Krishnamurthy, S. Peter, and K. Gupta, "ipipe : A framework for building datacenter applications using in-networking processors," 2018.

[17] J. Stern, Q. Xiong, A. Skjellum, and M. Herbordt, "A novel approach to supporting communicators for in-switch processing of mpi collectives," in *Workshop on Exascale MPI*, 2018.

[18] K. Kandalla, H. Subramoni, K. Tomko, D. Pekurovsky, and D. K. Panda, "A novel functional partitioning approach to design high-performance mpi-3 non-blocking alltoallv collective on multi-core systems," in *2013 42nd International Conference on Parallel Processing*. IEEE, 2013, pp. 611–620.

[19] T. Schneider, S. Eckelmann, T. Hoefler, and W. Rehm, "Kernel-Based Offload of Collective Operations - Implementation, Evaluation and Lessons Learned," in *Proceedings of the 17th international conference on Parallel processing - Volume Part II*. Springer-Verlag, Aug. 2011, pp. 264–275.