Journal of Parallel and Distributed Computing 144 (2020) 1–13

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

FALCON-X: Zero-copy MPI derived datatype processing on modern CPU and GPU architectures

Jahanzeb Maqbool Hashmi *, Ching-Hsiang Chu, Sourav Chakraborty, Mohammadreza Bayatpour, Hari Subramoni, Dhabaleswar K. Panda

Department of Computer Science and Engineering, The Ohio State University, United States of America

ARTICLE INFO

Article history:
Received 8 November 2019
Received in revised form 21 April 2020
Accepted 17 May 2020
Available online 28 May 2020

Keywords:
HPC
MPI
Derived datatypes
CPU and GPU
NVIDIA DGX2

ABSTRACT

This paper addresses the challenges of MPI derived datatype processing and proposes FALCON-X — A Fast and Low-overhead Communication framework for optimized zero-copy intra-node derived datatype communication on emerging CPU/GPU architectures. We quantify various performance bottlenecks such as memory layout translation and copy overheads for highly fragmented MPI datatypes and propose novel pipelining and memoization-based designs to achieve efficient derived datatype communication. In addition, we also propose enhancements to the MPI standard to address the semantic limitations. The experimental evaluations show that our proposed designs significantly improve the intra-node communication latency and bandwidth over state-of-the-art MPI libraries on modern CPU and GPU systems. By using representative application kernels such as MILC, WRF, NAS_MG, Specfem3D, and Stencils on three different CPU architectures and two different GPU systems including DGX-2, we demonstrate up to 5.5x improvement on multi-core CPUs and 120x benefits on DXG-2 GPU system over state-of-the-art designs in other MPI libraries.

1. Introduction

Modern High-Performance Computing (HPC) systems are enabling scientists from different research domains to explore, model, and simulate computation-heavy problems at different scales. The availability of multi- and many-core architectures (e.g., Intel Xeon, Xeon Phi, OpenPOWER, and NVIDIA Volta GPUs) has significantly accelerated the impact and capabilities of such large-scale systems. The current multi-petaflop systems are powered by such multi- and many-core CPUs and GPUs, and the adoption of these many-core architectures is expected to grow in future exascale systems [1]. Message Passing Interface (MPI) [19] has been used as the de-facto programming model for developing high-performance parallel scientific applications for such systems while Compute Unified Device Architecture (CUDA) being the primary programming interface to exploit NVIDIA GPUs. The emergence of CUDA-aware MPI [35] has relieved the application developers of manually moving the data between the host (CPU) and device (GPU) memories by switching between MPI and CUDA programming models for the communication phases of the applications. This allows the decoupling of CUDA and MPI programming models within the applications as the CUDA kernels are now used for computation while MPI is used to derive applications' communication. This ubiquity of MPI as a de-facto programming model for modern CPU and GPU-based systems mandates that the MPI libraries must be carefully designed to deliver the best possible performance for different communication primitives.

High-performance parallel algorithms and scientific applications often need to communicate non-contiguous data. For example, matrix-multiplication or halo-exchange often requires communicating one or multiple columns of large matrices stored in row-major format. To achieve this, the application can 'pack' the data into a temporary contiguous buffer and send it to the recipient process, which can then 'unpack' the data. However, this approach (known as "Manual Packing/Unpacking") provides poor performance due to the multiple copies of the data and the increased memory footprint of the application. Researchers have shown that this packing/unpacking can take up to 90% of the total communication cost [29]. Moreover, this places the burden of managing these temporary buffers and manually copying the data on the application developer, leading to poor productivity.

To address this, MPI provides a feature called Derived Datatypes (DDT) for communicating non-contiguous data in a portable and efficient manner. In this approach, the application composes a *Derived Datatype* using simple datatypes predefined by the MPI standard; and uses this datatype in the communication primitives. However, state-of-the-art MPI libraries suffer

* Corresponding author.
E-mail addresses: hashmi.29@osu.edu (J.M. Hashmi), chu.368@osu.edu (C.-H. Chu), chakraborty.52@osu.edu (S. Chakraborty), bayatpour.1@osu.edu (M. Bayatpour), subramoni.1@osu.edu (H. Subramoni), panda.2@osu.edu (D.K. Panda).

https://doi.org/10.1016/j.jpdc.2020.05.008
0743-7315/

from the poor performance of derived datatype processing causing many applications such as WRF [38], MILC [18], NAS MG [21], and SPECFEM [32] to still rely on the manual pack/unpack method instead of using DDTs [29]. While researchers have proposed designs to improve the communication performance of DDTs on interconnects like InfiniBand [16,27,33], some of the fundamental bottlenecks in datatype processing such as efficient translation from the datatype to memory-layout remain unsolved. Furthermore, the challenges involved in handling the derived datatype communication for both CPU and GPU resident data brings forth several new challenges. For instance, MPI libraries still use shared-memory-based designs for intra-node datatype communication for CPU resident data, which requires multiple copies and offers poor performance and overlap. Similarly, state-of-the-art designs in CUDA-aware MPI implementations employ CUDA kernel-based solution to accelerate the packing/unpacking phases [4,30,39] for GPU-resident data. However, it still suffers from the significant synchronization overhead between CPU and GPU.

While zero-copy techniques for improving intra-node communication performance have been studied in depth [2,3,9,11,15], the trade-offs involved in using these techniques for non-contiguous communication have not been explored in the literature. In this work, we show that using zero-copy techniques for MPI datatypes exposes novel challenges in terms of correctness and performance, and propose efficient designs to address these issues for modern CPU and GPU systems. We also propose designs to reduce the layout translation overhead through memoization-based techniques. Finally, we show that current MPI datatype routines are not able to fully take advantage of the zero-copy semantics and propose enhancements to address these limitations.

2. Motivation

The poor performance of datatype-based communication in MPI libraries has been well documented in the literature [34,40]. To understand the bottlenecks involved in datatype-based communication in MPI for CPU and GPU systems, we analyze the communication latency of one such transfer from various representative application kernels — WRF, MILC, NAS, and SPECFEC3D_CM provided by DDTBench [28]. We have also modified the DDTBench to support evaluating CUDA-aware MPI libraries.

Fig. 1 shows the profile of the application kernels on the CPU and GPU. The performance trends were obtained using the MVAPICH2 MPI libraries [20] on a Broadwell cluster. More details about the experimental setup are described later in Section 7. Similar trends were observed for MPICH and its derivatives like Intel MPI as well. As shown in Fig. 1(a), the communication cost of CPU-resident data involves two major components - (a) translation of the datatype's definition to the actual memory layout, and (b) copying of the data from the source to the target buffers. The relative costs also depend on the amount of data transferred as well as the complexity of the datatype used. Fig. 1(b) reveals the major performance issues of GPU-based datatype communication associated with state-of-the-art solutions in MVAPICH2-GDR [31]. As it can be seen, the communication cost is dominated by the various CUDA operations used for preparing the datatype layout for pack/unpack kernels, i.e., only less than 20% of the time is spent on moving the data Again, similar trends were observed for other CUDA-aware libraries such as Open MPI. To design an efficient MPI derived datatypes communication runtime, we need to consider the observed overhead and investigate which designs and techniques can be used to improve their performance.

2.1. Inefficient translation of datatype to memory layout

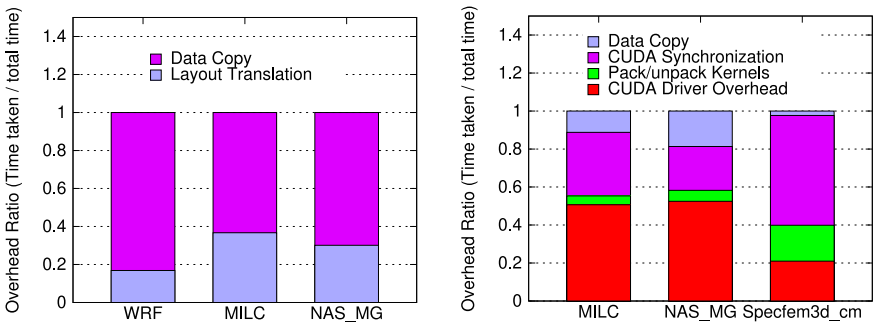
MPI derived datatypes are described using a *Type Map*, which consists of an ordered list of primitive datatypes and displacements from the base address [34]. A derived datatype can also contain other derived datatypes, known as *Nested Datatypes*. Since the sender and the receiver process can use different datatypes, they must independently translate the datatype definition to the memory layout. This *Layout Translation* is done for each transfer since the outcome depends on the various user inputs such as the base address of the buffer, the datatype itself, and the number of datatype elements. This step can quickly become very costly if a complex datatype or a large count is specified. Since most scientific applications use the same datatype for many iterations, there is an opportunity to amortize this cost by reusing the translation for multiple communications. This raises the following new challenges for MPI library designers: *a) How can the cost of inferring the memory layout from derived datatypes be reduced? (b) Can this cost be amortized through reusing the layout for multiple communications? (c) How can this be achieved while maintaining correctness and without introducing additional synchronization requirements?* Note that these challenges are independent of the communication mechanism and are applicable to both shared-memory and zero-copy-based designs.

2.2. Inefficient data movement

MPI libraries rely on shared-memory-based designs for intra-node datatype communication. Once the sender has computed the individual segments to be sent, it starts copying them into a pre-allocated shared memory region. Since the shared memory region is usually divided into multiple equal-sized blocks, the receiver can start copying the data into its local buffers as soon as the first block has been filled while the sender moves on to the next block. This strategy is commonly referred to as *Pipelined Shared Memory* design. However, this approach requires creating multiple copies of the data (once at the sender and once at the receiver), and requires both the sender and the receiver CPU to actively progress the communication, which leads to poor performance and overlap potential. To improve the performance and overlap, we explore the efficacy of kernel-assisted "*zero-copy*" data transfer techniques such as Cross Memory Attach (CMA) [17], and XPMEM [24] that allow a process to directly read from or write to the memory of another process on the same node. Similarly, to avoid expensive data copies and packing/unpacking kernels for GPU-resident data. We investigate the Peer-to-peer (P2P) technique, such as GPUdirect technology [22] through CUDA Inter-Process Communication (IPC) APIs for NVIDIA GPUs, to allow direct data movement between two GPUs on the same node. Although most modern MPI libraries support using zero-copy mechanisms for intra-node communication of contiguous buffers, the challenges and trade-offs involved in using these techniques for non-contiguous buffers have not been explored in the literature.

2.3. Challenges for zero-copy datatype communication

The use of zero-copy techniques requires sending the data layout to the remote process before the copy can be initiated. However, since the MPI standard only provides routines for local creation and destruction of derived datatypes, the receiver process is oblivious to the memory layout of the sender's datatype. Hence, zero-copy-based designs introduce an additional overhead of *Layout Exchange*, which is not present in current shared-memory-based designs. For heavily fragmented datatypes, the cost of this exchange can be significant and sometimes be even



(a) CPU-based DDT Communication (b) GPU-based DDT Communication

Fig. 1. Cost breakdown of existing designs for datatype processing in MPI when using CPU and GPU resident data in communication on different application kernels.

higher than the cost of the actual data movement. Furthermore, our prior works [3,11] have shown that there are other overheads such as *kernel-level contention*, *remote address translation*, etc., that can affect the performance of zero-copy techniques. For GPU-resident data, the overhead primarily lies in issuing and waiting for CUDA operations of the P2P data movement [26]. Based on these observations, we identify the following four challenges introduced by the goal of zero-copy datatype communication: *(a) What are the overheads involved in using zero-copy techniques for communicating CPU- and GPU-resident non-contiguous buffers or datatypes? (b) What designs can be employed to minimize the cost of such overheads? (c) How can zero-copy designs efficiently handle different datatypes at the sender and the receiver? (d) Can enhancements to the MPI semantics be proposed to mitigate the layout translation and exchange overheads?*

3. Contributions

These observations lead us to the following broad challenge: How can we design a high-performance and efficient zero-copy-based communication runtime for MPI derived datatypes on modern CPU/GPU systems? In our prior work [12] we have discussed the challenges involving CPU-based derived datatype processing. In this paper, we enhance our earlier work on CPUs and augment it further by proposing designs for GPU-based MPI derived datatype processing in CUDA-aware MPI libraries.

In this work, we address the challenges mentioned in Section 2 and propose FALCON-X — Fast and Low-overhead Communication designs for zero-copy MPI datatype processing on CPU/GPU architectures. To the best of our knowledge, this is the first study to identify and analyze the trade-offs involved in designing zero-copy communication primitives for non-contiguous data movement and address them in an efficient manner. We also propose solutions to mitigate the overhead of layout translation using designs internal to the MPI library and enhancements to the MPI datatype creation semantics. We integrate our designs in the popular MPI library MVAPICH2 [20] and MVAPICH2-GDR [37] and show their efficacy using various microbenchmarks and applications. Our proposed designs can reduce the intra-node communication latency of MPI derived datatypes by up to 2× and improve the performance of the communication kernels of different applications such as 3D-Stencil, MILC, WRF, NAS_MG, and Specfem3D by up to 5.5× for host-based communication and up to 120× for GPU-based communication on a DGX-2 system. To summarize, we make the following key contributions:

- Identify the challenges and trade-offs involved in using zero-copy techniques for MPI datatype processing
- Design efficient pipeline and novel caching mechanisms to mitigate various overheads associated with zero-copy communication schemes
- Propose design optimizations to enable reuse of datatype layout information and amortize the cost of layout translation
- Optimized host and device-based derived datatype communication in MVAPICH2 and MVAPICH2-GDR MPI libraries
- Propose enhancements to current MPI datatype creation semantics to enable further high-performance designs for datatype-based communication
- Demonstrate the efficacy of the proposed designs on real state-of-the-art CPU/GPU systems using micro-benchmarks and applications

4. Designing zero-copy MPI datatype processing on modern CPUs

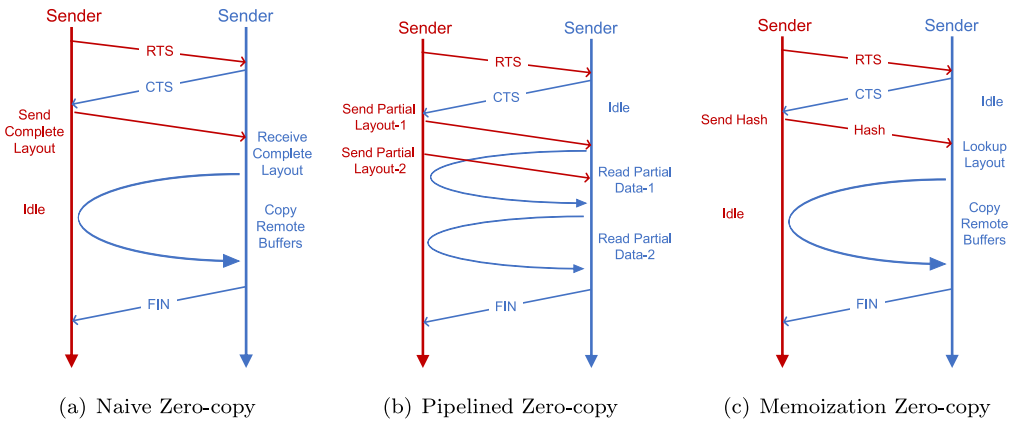
In this section, we look at the detailed designs for using zero-copy techniques for datatype-based intra-node communication, and propose mechanisms to improve the performance of such designs.

4.1. Naive zero-copy design

In contrast to shared-memory-based designs, zero-copy designs rely on the sender transmitting the layout of the data instead of the actual contents to the receiver. Fig. 2(a) illustrates this design at a conceptual level.

Based on the parameters given to `MPI_Send(buffer address, datatype, count)`, the sender first translates the datatype layout and creates a list of individual contiguous segments that need to be transferred. Each segment is defined using a base address and an offset, commonly referred to as an I/O vector or IOV. The sender then sends an RTS (Request-to-Send) packet to the receiver, which contains various metadata such as the source, tag, and the communicator context id. The receiver uses this information to match the send to an appropriate receive operation and responds with a CTS (Clear-to-Send) packet. The sender then sends a data packet to the receiver, which contains this list of segments or IOVs. Once this packet reaches the receiver, it copies the data directly into the target buffers by copying the individual segments from the sender's memory.

In the case of CMA, the receiver process issues a `process_vm_readv` system call, which takes the list of local and remote segments/IOVs as input, and copies the data from the source buffers to the target buffers inside the kernel. In the case of XPMEM, the receiver process needs to "attach" the remote pages to its own address space, and then directly read the contents of the individual segments into its local buffers. The reverse, where the receiver describes the layout of the target buffers to the sender which then writes the data is also possible.



(a) Naive Zero-copy (b) Pipelined Zero-copy (c) Memoization Zero-copy

Fig. 2. A high level overview of the various proposed designs for zero-copy datatype communication. These designs are applicable to both CPU and GPU-based MPI communication.

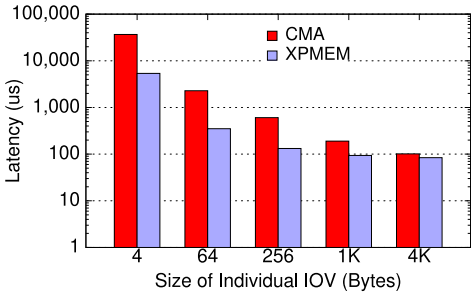


Fig. 3. Performance comparison of CMA and XPMEM-based zero-copy with varying IOV size. The total size of the buffer is 1 MB while individual IOV sizes vary between 4 bytes to 4 KB.

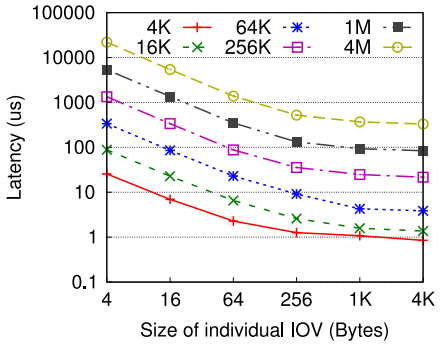


Fig. 4. Impact of data fragmentation on naive zero-copy datatype processing using XPMEM. The lines represent total size of the communication buffer ranging from 4 KB to 4 MB.

The performance of CMA and XPMEM-based approaches heavily depend on the amount of total data transferred as well as the size of the individual segments. Since CMA incurs the overhead of system call for each contiguous segment, its performance is highly affected by the fragmentation of the datatype. The relative performance of CMA-based naive zero-copy datatype processing against XPMEM-based naive zero-copy with for 1 MB buffer with varying IOV size (fragments) is shown in Fig. 3. The IOV size ranges from 4 bytes to 1 KB which also varies the fragmentation factor from high to low. The performance of CMA suffers from expensive system calls when data is more fragmented, e.g., the size of each IOV is small, and the total IOV count is high. However, as the fragmentation decreases along the *x*-axis, the relative performance difference between CMA and XPMEM also decreases. At 4 KB IOV size (256 total IOVs), CMA is only 16% slower than XPMEM as compared to 6.8× when IOV size is 4 bytes (256 K total IOVs). Based on this evaluation, we will be focusing on using XPMEM for the remainder of this paper. However, the designs proposed later to mitigate various overheads are independent of the underlying transport mechanism and would be applicable to CMA or other zero-copy schemes as well.

In the following sections, we will focus on XPMEM-based solutions to avoid unnecessary overheads posed by CMA.

Fig. 4 shows the impact of varying datatype fragmentation using XPMEM as the underlying transfer mechanism. For a fixed message size, small-sized segments lead to higher IOV counts and increase the communication latency. This is expected as smaller size segments lead to higher fragmentation of the data and increase the total number of segments that need to be communicated. When IOV size is the same as the overall message size, there is no non-contiguity in data e.g., only a single buffer is being communicated. This allows the CPU to prefetch data and exploit spatial locality. This also shows that there are additional overheads in this design (other than the cost of copying) that become prominent as the number of segments increase. To further analyze the impact of this behavior on application performance, we show the breakdown of the time taken by the individual steps inside the XPMEM-based naive zero-copy design using the communication kernels from three applications discussed in Section 2. The results are shown in Fig. 5. Compared to the shared-memory-based designs, the time taken for layout translation remains unchanged while the time taken for data movement is reduced due to avoiding extra copies. However, this improvement is negated by the two additional steps introduced by the zero-copy design — (a) Layout Exchange, and (b) Address Translation. **Layout Exchange** refers to the cost of communicating the in-memory layout of the source buffers to the receiver. This step is required since the receiver cannot assume that the sender is using the same datatype as the receiver. The layout sent by the sender also cannot be used directly by the receiver, since the base addresses are from the sender's address space and invalid in the receiver's address space. Thus, the receiver must 'translate' the remote addresses to valid local addresses before the data can be copied. This step is shown as **Address Translation**. The overhead of both these steps increases as the datatype becomes more fragmented, i.e., contains more non-contiguous segments or IOVs. While we show the breakdown for the design using XPMEM, similar trends can be observed for other zero-copy techniques as well. For example, the cost of creating and exchanging the layout is the same for CMA, but the address translation and copying the data are combined inside the `process_vm_readv` system call. In

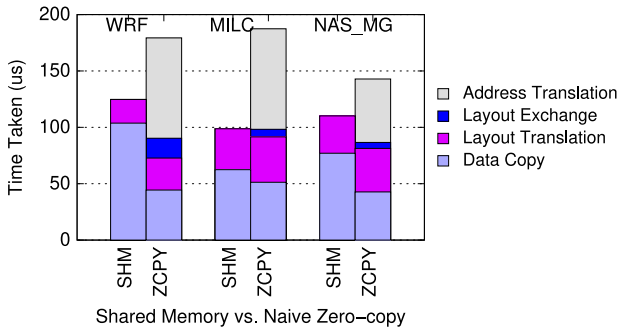


Fig. 5. Time breakdown of various steps involved in Shared Memory (SHM) and XPMEM-based naive zero-copy (ZCPY) based design.

both cases, the actual cost of copying the data contributes only a portion of the overall time. The rest comes from the various preparatory steps that need to be performed before the data movement can take place. These overheads have been analyzed in detail in our prior work [3,11].

4.2. Pipelined zero-copy design

As shown in Fig. 5, the overhead of sending the layout from the sender to the receiver can consume a significant portion of the total communication time. This also delays the receiver from performing useful work until all the IOVs have been received. In order to address this limitation posed by the naive design, we propose a pipelined zero-copy design. Fig. 2(b) describes this design at a conceptual level. The main idea behind this design is to send a partial list of sender's IOVs allowing the receiver to initiate the data transfer without waiting for the sender to finish sending the entire list of IOVs. Meanwhile, the sender continues to send subsequent chunks of the layout in a pipelined fashion. The advantage of pipelined design over naive design is twofold: (1) it allows the receiver to progress communication as soon as the first chunk of IOVs is received, and (2) it overlaps the costs of sending the data layout and copying of the data and thereby reducing the overall latency. Although pipelined zero-copy design hides most of the overheads associated with the layout exchange, it still requires the sender to actively progress the communication until all the layout information is sent. Thus, it reduces the availability of the sender to progress other communication or application computation when non-blocking sends are used.

4.3. Memoization-based zero-copy design

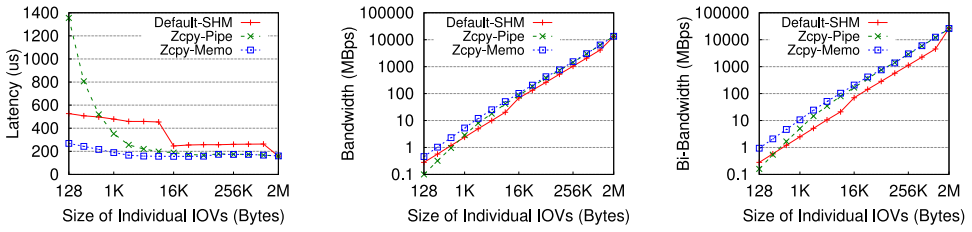
To address the limitations of pipelined zero-copy design, we propose a memoization-based scheme where the communicating peer of processes *memoize* the derived datatype layouts that are being exchanged and use the *memoized* layouts for subsequent communications. In this design, both the sender and the receiver processes maintain a hashtable for memoization. The table on the sender side stores the translated layout of the sender's datatype, while the table on the receiver side stores the sender's layouts that have so far been used in the communication between these pairs of processes. When the sender process issues a send request containing a non-contiguous derived datatype for a particular receiver, the MPI runtime, after translating the layout, generates a `SHA1` hash of this translated layout. This hash is then searched in the local hashtable. If the entry is found, this means the sender has communicated with its peer using this IOV layout and thus, the sender proceeds to ignore sending of the translated layout (list of IOVs) to the receiver. If the hash does not match, then the sender performs two tasks: (1) it *memoizes* the current datatype layout by storing its associated hash as a *key* in the local hashtable, and (2) it embeds the hash along with the translated IOVs into data packet before sending it to the receiver. When a request arrives at the receiver, it first pulls the value of the hash from the packet and looks it up in the local hashtable associated with the sender's rank. If the hash is not found, this indicates that the sender is trying to communicate a new datatype layout, and the receiver can find a list of IOVs containing the sender's datatype layout in the received packet. The receiver then proceeds to *memoize* this layout by creating a new entry in its local hashtable by using the received hash as a *key* and the received list of IOVs as the value. In the case where the received hash matches, the receiver does not pull any IOV information from the packet. Fig. 2(c) shows a high-level overview of our memoization-based design. We employ a standard chaining mechanism to avoid hash collisions by embedding *request_id* in each hashed entry. The memoization-based design completely eliminates the overhead of communicating the sender's datatype layout. Since most applications perform many communications using the same datatype, this design effectively performs the layout translation only for the first transfer and amortizes the cost over the rest.

4.4. Avoiding remote address translation

The IOV layout of the sender contains addresses that are valid only in the sender's context while the receiver process needs to 'translate' these remote addresses to appropriate local addresses before any access is made. However, as shown in Fig. 5, this step contributes significantly to the overall communication cost. To mitigate this, we enhance the memoization-based design so that the receiver stores the IOV list that has already been translated into local addresses. This ensures that the costly steps of attaching to remote pages and translating the addresses are done only once for each pair of processes. However, attaching to a large number of remote pages can increase the page table size and impact the overall performance. To avoid this, we implement a simple LRU cache that discards the least recently used datatypes and their translated mappings and detaches from the associated remote pages. If a discarded datatype is used again, the receiver requests the sender to resend the layout using the CTS packet and performs the translation again. Each entry of the LRU cache contains a 20-byte hash value and a 4-byte pointer to a list of IOVs. The total number of entries is a tunable parameter.

4.5. Design discussion

To validate the efficacy of the proposed zero-copy designs, we compare their performance against the existing shared memory design for point-to-point communication. Fig. 6 shows the results of these experiments on a Broadwell system. We compare the latency, bandwidth, and bi-directional bandwidth of three approaches: existing shared-memory-based design (Default-SHM), pipelined zero-copy design (Zcpy-Pipe), and memoization-based zero-copy design (Zcpy-Memo). The naive zero-copy design described in Section 4.1 is not shown due to its extremely poor performance. For all the cases, we fix the message size to 2 MB and vary the size of each segment from 128 bytes (16,384 segments) to 2 MB (one contiguous segment). As shown in Fig. 6(a), the latency of Zcpy-Pipe is poor when the number of IOVs is large, but it gradually improves as the number decreases. The Zcpy-Memo design avoids the layout exchange overhead and hence is not affected significantly by the number of IOVs. Compared to the shared-memory-based design, the zero-copy design is up to 3× faster till 8 KB and roughly 50% faster till 1 MB. The discontinuity in the 8–16 KB range is due to the switch from eager to rendezvous protocol. At 2 MB, the buffer is contiguous;



(a) Latency (b) Unidirectional Bandwidth (c) Bidirectional Bandwidth

Fig. 6. Performance comparison of proposed zero-copy designs against shared-memory-based designs on Broadwell.

hence, all three designs show similar performance. Figures 6(b) and 6(c) compare the performance of unidirectional and bidirectional bandwidth. The zero-copy design shows similar benefits for bandwidth. However, the benefit for bi-bandwidth is higher since the shared memory design requires both the sender and the receiver processes to progress the communication while the zero-copy design only requires the receiver process, thus delivering more application-level performance.

5. Designing zero-copy design on multi-GPU systems

In the modern multi-GPU systems, high-performance interconnects such as PCIe and NVLink are widely used to connect GPUs. This enables peer-to-peer (P2P) access by using either the driver APIs (i.e., via copy call) or within a compute kernel (i.e., via direct load-store operations). In this section, we elaborate on the proposed MPI-level solutions to address the challenges of leveraging the P2P feature for achieving the zero-copy data movement of non-contiguous GPU-resident data. We implemented the designs similar to the designs discussed for the host in Sections 4.1 and 4.2. However, in the following sections, we only discuss the optimized memoization-based zero-copy design for GPUs to avoid redundant insights.

5.1. Naive P2P-based zero-copy design

When the sender's layout is properly exchanged, and P2P access is available between GPUs, the receiver process can issue multiple asynchronous copy primitives `cudaMemcpyAsync`, i.e., one for each contiguous block, to directly move non-contiguous data without additional copies. Although this achieves packing-free data transfer, issuing multiple copies still incurs significant overhead due to multiple calls made to the CUDA driver API. Past research has extensively studied the overheads of CUDA drive API calls [4,5]. We find that the proposed memoization-based design while showing the best performance for CPU, exhibits poor performance on GPUs for highly fragmented layouts, e.g., datatype with a large number of contiguous blocks. The overheads caused by CUDA driver level calls for P2P communication outweigh the benefits obtained from memoization-based designs for GPU resident data. It is worth noting that each CUDA call, e.g., `cudaMemcpyAsync`, may incur 3–10 μs driver overhead in the state-of-the-art GPU hardware. Using a naive zero-copy approach for sparse datatypes used in applications such as MILC can result in significant performance overheads, as shown in Fig. 7. Thus, a more efficient zero-copy design is required for GPU-resident data.

5.2. Kernel-based zero-copy design

To fully exploit the zero-copy designs on multi-GPU systems, we propose a load-store-based solution to minimize the driver overhead by performing data movement of non-contiguous buffers in one shot. To maximize the throughput, one warp (e.g., 32 threads in NVIDIA GPUs or a workgroup in AMD GPU) is responsible for moving one single contiguous block at a time, so that it can achieve higher memory access throughput by taking advantage of memory coalescing. That is, within each warp, each thread loads an element from remote GPU and stores it to local GPU, i.e., through GPUDirect P2P support. If the number of blocks is larger than the number of available warps, each warp will be working on multiple contiguous blocks in a round-robin manner.

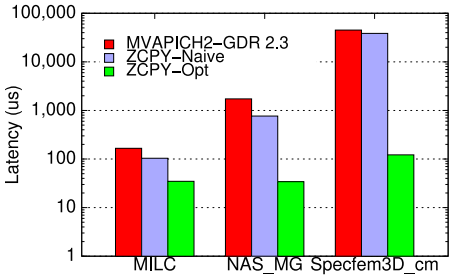


Fig. 7. Improvements of the optimized zero-copy over naive design on GPU-resident data.

One significant advantage of using load/store kernel is that the receiver can launch multiple kernels to quickly saturate a high-speed interconnect or transfer data on separate NVLinks in parallel without contention. Furthermore, the existing datatype-processing methods [30,39] launch multiple packing/unpacking kernels or data copies to perform packing/unpacking. This incurs a significant driver overhead, and the communication cannot proceed before the packing phase is completed. In contrast, the proposed design asynchronously launches one kernel for each *MPI_Recv or MPI_Irecv* to minimize the driver overhead and overlaps multiple kernels to hide the driver and synchronization overheads while maintaining high-throughput and low-latency. Moreover, the kernels provide overlap opportunities with application-level computation, which can further reduce the overall application execution time. Fig. 7 demonstrates the significant performance benefits over the naive design, up to 316× for the Specfem3D_cm kernel.

6. Efficient layout translation designs for CPU/GPU systems

The designs proposed in Section 4 focus on reducing the cost of exchanging the sender's layout. However, they still involve the layout translation from the local datatypes on the receiver process. As shown in Fig. 5, this datatype to memory layout translation can be very costly for nested datatypes due to the recursive nature of the datatype parsing. To mitigate this overhead, we propose two approaches that eliminate the layout translation overheads for both the sender and the receiver processes. The first approach involves caching the results of the datatype parsing, while the second approach involves enhancing the MPI semantics for datatype creation and destruction.

6.1. Layout reuse through caching

As described in Sections 4.3 and 4.4, the sender and the receiver maintain hash tables to keep track of the sender's translated layouts. However, these designs suffer from two limitations — (a) the sender calculates the hash-based on the list of IOVs, and (b) the receiver recreates its own data layout for every receive operation. We enhance this design by hashing the communication pattern instead of the translated IOVs. Listing 1 shows the structure describing the information of the communication request that is used as the input to the hash function. The contents of this structure are chosen to ensure that a combination of these can uniquely identify a particular set of translated IOVs. Any change in any of these parameters in this object means that it requires creating a new layout. In our proposed design, when the sender or the receiver posts a new communication request, they create their local data layout according to the request and store this information against a hash generated by this combination. If the hash value is already present in the table, then both the translation and the exchange of the layout can be skipped. This way, both the sender and the receiver avoid this costly step after the first iteration. To allow for discarding the rarely used datatype layouts, additional flags are used in the RTS and CTS packets to force the re-calculation and re-transmission of the IOV layouts.

```
struct dt_buf_info {
  void *addr;      // base address
  MPI_Aint count; // number of elements
  int dt_handle;  // unique datatype identifier
  int rank;       // rank of source/destination
};
```

Listing 1: User provided parameters for reusing layout

6.2. Proposed enhancements to MPI semantics

While the *memoization*-based design proposed in Section 6.1 reduces the cost of exchanging the layout information between the sender and the receiver, the sender still needs to communicate its data layout to the receiver when a datatype is used for the first time. Thus, the first communication using a datatype incurs more overhead compared to the subsequent transfers. Furthermore, the sender calculates the hash-based on the datatype's unique identifier, commonly referred to as the 'handle'. Thus, each datatype needs to be assigned a unique handle for this scheme to work. However, the MPI standard allows freeing of existing datatypes and does not guarantee that the handles of freed datatypes will not be reused. This can lead the sender or the receiver to incorrectly conclude that the datatype in a particular send/recv operation is being reused, leading to incorrect behavior.

MPI libraries can tackle the second issue by ensuring that datatype handles are never reused once they have been allocated. However, the first issue still exists because datatype creation is a 'local' operation (i.e., only the caller process participates) in the current MPI standard. Thus, two processes cannot easily assign the same unique identifier (handle) to the same datatype. To address this, we propose two new MPI routines to create and free derived datatypes shown in Listing 2.

These new functions are collective; thus, all the processes in the specified communicator must participate. This allows the MPI library to ensure that the newly created datatype is allocated the same handle across all processes in the communicator. We implement this function to perform an allreduce operation internally to determine the largest handle (L) used by the participant processes and allocate the following handle ($L + 1$) to the datatype. Since MPI libraries can internally call MPI_Type_commit and continue to use the existing non-zero-copy-based designs, the complexity of adoption for the new API is negligible.

Table 1
Hardware specification of different CPU testbed clusters.

Specification	Xeon	Xeon Phi	OpenPOWER
Processor family	Intel broadwell	Knights landing	IBM POWER8
Processor model	E5 v2680	KNL 7250	S822LC (8335-GTA)
Clock speed	2.4 GHz	1.4 GHz	3.4 GHz
No. of sockets	2	1	2
Cores per socket	14	68	10
Threads per core	1	4	8
RAM (DDR)	128 GB	96 GB	256 GB
Interconnect	IB-EDR (100 G)	IB-EDR (100 G)	IB-EDR (100 G)

```
/* Create a new datatype */
int MPIX_Type_commit_comm(
      MPI_datatype *datatype,
      MPI_Comm comm);

/* Free an existing datatype */
int MPIX_Type_free_comm(
      MPI_datatype *datatype,
      MPI_Comm comm);
```

Listing 2: Proposed enhancements to MPI datatype routines. The new functions allow collective creation and freeing of derived datatypes.

7. Experimental evaluation

We used three production MPI libraries — MVAPICH2-X v2.3rc1, MVAPICH2-GDR 2.3.2, Intel MPI (IMPI) v2018.1.163 and v2019.0.045, and Open MPI v3.1.2 with UCX v1.3.1. MVAPICH2-X and Open MPI+UCX were configured to use XPMEM as the intra-node transport mechanism. Our early experiments showed IMPI 2018 to be performing better than IMPI 2019 for certain benchmarks. We attribute this to the lack of optimizations for derived datatypes in libfabric. Thus, we present the results for both versions of Intel MPI to compare against the best available designs. MVAPICH2-GDR and Open MPI are used to evaluate the datatype processing for GPU-resident data. In our prior work, we had observed vendor-provided MPI implementations to be performing best on systems like KNL and OpenPOWER. Thus, on these systems, we compare our designs against IMPI and Spectrum MPI, while on Broadwell, we use Open MPI instead of SpectrumMPI.

For micro-benchmark evaluation, we enhanced OSU Micro-Benchmarks (OMB) v5.4.4 [23] to add support for MPI derived datatypes. Each OMB test was run for 100 iterations, and the average of 5 runs is reported. DDTBench [28] was used to evaluate the communication performance of derived datatypes used in popular scientific application kernels. Moreover, we extended DDTBench to support GPU based derived datatype communication. Tables 1 and 2 list the hardware specifications of our CPU and GPU testbeds, respectively.

7.1. Evaluation on multi-/many-core CPUs

In the following sections, we demonstrate a detailed evaluation of our proposed optimized design (MV2X-OPT) on three different CPU architectures using microbenchmarks and application kernels.

Table 2
Hardware specification of different GPU-enabled testbed clusters.

Specification	Cray CS-Storm	NVIDIA DGX-2
CPU processor	Intel Xeon E5-2690 (Haswell) 2.6 GHz	Intel Xeon Platinum 8168 (Skylake) 2.7 GHz
CPU cores/socket	12	24
GPU processor	NVIDIA Tesla K80	NVIDIA Tesla V100
GPU memory	24 GB	32 GB
Interconnects between GPUs	PCIe Gen3 (x16)	NVLink2 and NVSwitch
NVIDIA driver version	410.79	410.48
CUDA toolkit version	9.2.148	9.2.148

7.1.1. Performance of point-to-point communication

To compare the performance of our proposed design for point-to-point communication against state-of-the-art MPI libraries, we enhanced the OSU MicroBenchmark Suite (OMB) to use datatypes. We used an MPI `vector` datatype containing blocks of `MPI_CHAR` datatype. The total message size was fixed to 2 MB while the size of each block was varied from 128 bytes to 2 MB. Consequently, the number of blocks was varied from 16,384 to a single block. The stride was selected to be twice the block size, to allow the same amount of empty space between each block. The results of latency, bandwidth, and bi-directional bandwidth tests on the Broadwell architecture are shown in Fig. 8. As Fig. 8(a) shows, our proposed design (shown as **MV2X-OPT**) improves the latency by up to 3× for small-to-medium block sizes (<16 KB) and by up to 2× for larger blocks. The difference at 16 KB is caused by the switch from the "Eager" to the "Rendezvous" protocol. It shows that Intel MPI 2018 and MVAPICH2-X both use the same threshold for this architecture. We note that IMPI 2019 showed worse performance compared to IMPI 2018 for these tests, which we believe is due to the lack of optimizations in the new libfabric-based implementation used by IMPI 2019. We also see similar improvements for bandwidth as shown in Fig. 8(b). For bi-bandwidth, our design achieves twice the performance of unidirectional bandwidth while all other libraries show the same performance as bandwidth. This is because our proposed zero-copy design allows the sender and the receiver processes to progress two communications in opposite directions for non-blocking communication.

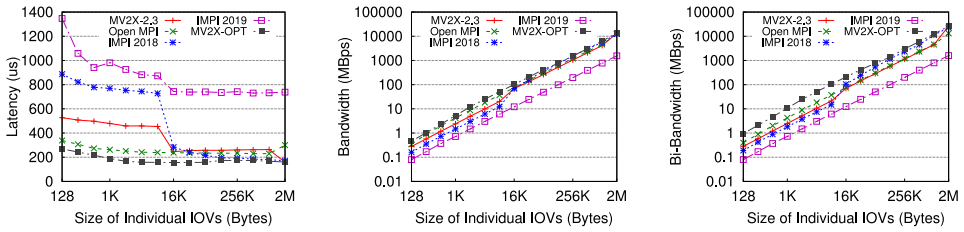
7.1.2. Application performance

MILC — MIMD Lattice Computation (MILC) application studies the interaction of quarks and gluons using Quantum Chromodynamics (QCD). MILC uses 48 different MPI DDTs to accomplish halo exchange in the 4 directions. The `MILC_su3_zd` kernel in DDTBench models the CG solver in the z direction of the `su3_rmd` application from the MILC code. Table 3 describes the datatype layout of the communication kernel. In this evaluation, we compared our proposed design, referred to as MV2X-Opt, against state-of-the-art MPI libraries. Fig. 9 shows the performance of different designs for different grid dimensions on three different architectures. The x-axis represents the increasing grid size, while the y-axis plots the total execution latency. As we can see, the proposed design shows significant improvement over other MPI libraries for almost all the grid dimensions. For instance, at $B = (32, 32, 32, 32)$ dimensions that correspond roughly to a 768 KB message, the proposed design shows 31.2% improvement over Open MPI, and up to 11× improvement over MV2X-2.3 on Broadwell (Fig. 9(a)).

Similar performance trends were observed on KNL system, as shown in Fig. 9(b), where MV2X-2.3 and IMPI 2018 both performed similarly while MV2X-Opt showed an order of magnitude improvement over both the libraries. The datatype layout in the MILC kernel is the most complex with deeper nesting levels among other communication kernels we used. Thus, datatype layout translation constitutes most of the communication time. The performance benefit of MV2X-Opt mainly stems from our memoization-based design, where we completely avoid the datatype translation and exchange overheads. On the OpenPOWER system, our proposed MV2X-Opt shows about 3× improvement over Spectrum MPI and almost two orders of magnitude improvement over MV2X-2.3 as shown in Fig. 9(c). These observations have led us to the findings that the datatype processing designs in MPICH and its derivative MPI libraries (MVAPICH2 and IMPI) are suboptimal for complex datatypes like the ones used in MILC. While Open MPI and its derivatives such as Spectrum MPI employ better designs as they are not impacted much by the datatype nesting levels. Although Open MPI and Spectrum MPI have better designs for complex datatype processing, they do not entirely eliminate the associated overheads. Thus, the proposed MV2X-Opt with our novel designs outperforms both MPICH and Open MPI derivative libraries.

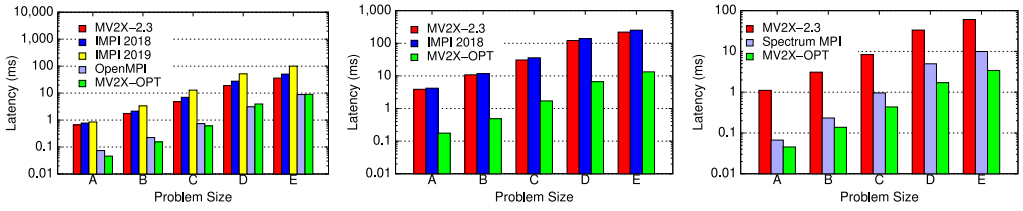
WRF — Weather Research Forecasting (WRF) application models the atmosphere using a regular 3-D Cartesian grid. The data decomposition is done in two horizontal dimensions only. During halo exchange phase, the slices of dimension arrays are communicated. DDTBench models the communication of the dimension arrays by creating either `hvector` of `vector` datatypes, or a `struct` of `subarray` datatypes. Table 3 describes the datatype layout of WRF for each dimension. Fig. 10 shows the performance of running WRF_y_vec kernel with varying problem sizes on Broadwell (Fig. 10(a)), KNL (Fig. 10(b)), and OpenPOWER (Fig. 10(c)) systems. The problem size is increased in such a way that it increases the block size in each `hvector` or `subarray` type creation. The evaluation shows that the proposed design exhibits good scaling with the increasing problem size. For instance, on Broadwell system using small problem set that communicates roughly 42 KB message, we observe 2.1× improvement in latency over Open MPI, 44% improvement over IMPI 2019, and 40.2% improvement over MV2X-2.3. Similarly, for one of the larger problem set with input parameters of $D = (4, 1036, 8, 1032)$ which correspond to a total of 3.8 MB message size, the proposed design shows up to 2.9×, 2.15×, and 22.6% improvement over IMPI 2019, MV2X-2.3, and Open MPI, respectively. This trend continued on the KNL system as well, where both IMPI 2018 and MV2X-2.3 showed comparable performance while MV2X-Opt outperformed both of them. Unlike MILC, the datatype layout of WRF is less complex due to which MV2X-2.3 does not suffer considerably from the layout translation and exchange. Furthermore, the zero-copy mechanism used in Spectrum MPI is CMA which has less performance than XPMEM. While MV2X-2.3 shows better performance than Spectrum MPI, MV2X-Opt shows up to 37% improvement in execution time over MV2X-2.3.

NAS_MG — communicates the faces of a 3-D array in a 3-D stencil where each process has six neighbors. A 2-D slice of the grid is communicated with corresponding neighbors in `x`, `y`, or `z` direction. DDTBench modifies the pack function in MG by constructing appropriate `subarray` or `hvector` datatypes. Different variations of the kernel e.g., NAS_MG_x, NAS_MG_y, and NAS_MG_z, represent data exchange in each of the three dimension using nested `vector` datatypes. Fig. 11(a) shows the performance of NAS_MG_z on a Broadwell system. As we can see, the proposed designs show significant improvement over other



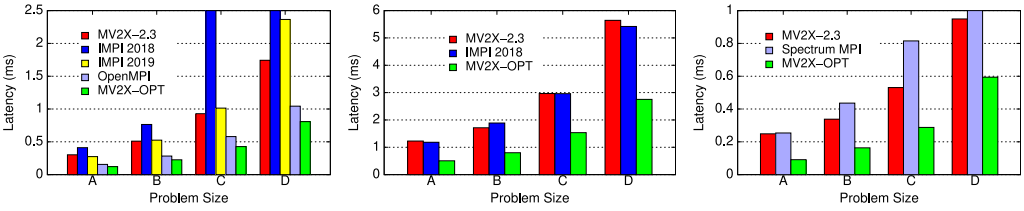
(a) Latency (b) Unidirectional Bandwidth (c) Bidirectional Bandwidth

Fig. 8. Performance comparison of proposed zero-copy designs against state-of-the-art MPI libraries on Broadwell. Total message size is fixed at 2 MB while size of each block creating `vector` type increases along the *x*-axis.



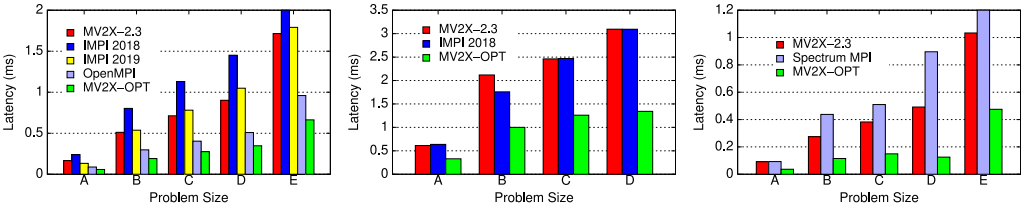
(a) Broadwell (b) KNL (c) OpenPOWER

Fig. 9. Performance comparison with MILC against state-of-the-art MPI libraries on different architectures. Grid Dimensions for $A = (16, 16, 32, 32)$, $B = (32, 32, 32, 32)$, $C = (64, 64, 32, 32)$, $D = (128, 128, 32, 32)$, $E = (128, 128, 64, 64)$.



(a) Broadwell (b) KNL (c) OpenPOWER

Fig. 10. Performance comparison with WRF against state-of-the-art MPI libraries on different architectures. Problem size parameters `(ims, ime, is, ie)` for $A = (4, 140, 8, 136)$, $B = (4, 268, 264, 8)$, $C = (4, 524, 8, 520)$, $D = (4, 1036, 8, 1032)$.



(a) Broadwell (b) KNL (c) OpenPOWER

Fig. 11. Performance comparison with NAS_MG against state-of-the-art MPI libraries on different architectures. Grid Dimensions for $A = (258, 130, 130)$, $B = (512, 258, 258)$, $C = (768, 258, 258)$, $D = (1024, 258, 258)$, $E = (2048, 258, 258)$.

Table 3
Scientific application kernels and their derived datatype Layout. We ran all variations of the kernels e.g., WRF_{x,y}_{vec, sa} and NAS_MG{x, y, z}, however, due to similarity in trends, one variation per kernel is included.

Application kernel	Application domain	Datatype layout
`MILC_su3_zd`	Quantum chromodynamics	Nested vectors for 4D face exchanges
`WRF_y_vec`	Atmospheric science	Nested vectors and subarrays
`NAS_MG_z`	Fluid dynamics	Vectors and nested vectors for 3D face exchanges
`3D-Stencil`	Stencil Communication	7-point stencil using Subarray datatypes

MPI libraries. For instance, at E = (2048, 258, 258) grid size, the proposed designs in MV2X-Opt show up to 2× improvement in latency over default MV2X-2.3, and IMPI 2019. The benefits over IMPI 2018 are more pronounced at scale (up to 3× improvement). Similar trends were seen on KNL system (Fig. 11(b)) as well where both MV2X-2.3 and IMPI performed similarly while MV2X-Opt showed up to 2.2× improvement. The performance trends observed on OpenPOWER (Fig. 11(c)) were similar to the WRF kernel, i.e., MV2X-2.3 showed better performance than Spectrum MPI by taking advantage of the user-space zero-copy designs while our proposed designs in MV2X-Opt showed up to 2× and 3× improvement over MV2X-2.3 and Spectrum MPI, respectively.

3D-Stencil — is a communication kernel that mimics the communication pattern of many stencil-based applications and Adaptive Mesh Refinement (AMR) kernels. This communication kernel performs a 7-point stencil with neighboring processes by using

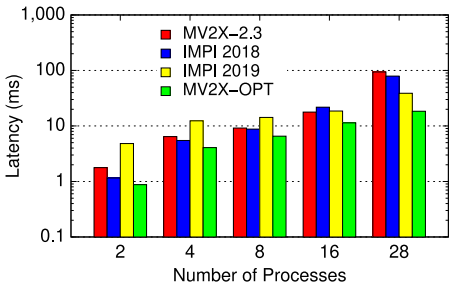


Fig. 12. Performance of 3D-Stencil with state-of-the-art designs in MPI libraries on Broadwell. Grid dimension = 512^3.

`subarray` datatypes to communicate 2-D *faces* of the cartesian grid. In this benchmark, we evaluate the performance of our proposed design by increasing the scale and keeping the problem size constant. Fig. 12 shows the performance of the proposed design against other MPI libraries. We fixed the size of the grid at 512^3 and vary the number of MPI processes on a Broadwell node. For a two-process run, the proposed design shows up to 5.5× improvement over Intel MPI 2019, and up to 2× improvement over MV2X-2.3. This trend is continued with increasing numbers of processes. At 16 processes, the optimized design shows 38.7% improvement over IMPI 2019, 47.8% over IMPI 2018, and 36% improvement over MV2X-2.3. At full subscription, with 28-processes per node, all the other MPI libraries perform worse while the proposed design showed a linear growth in time. The sudden increase in latency at full-subscription could potentially be explained by the imbalanced distribution of the cartesian grid. For KNL and OpenPOWER runs, we observed random crashes on a higher number of processes for all the MPI libraries, which we have reported to the application developers. Open MPI runs for this application crashed on all the systems.

7.2. Evaluation on multi-GPU systems

To evaluate GPU-based derived datatype communication kernels, we extended DDTBench to support the use of multiple GPU buffers. Our modified DDTBench uses the GPU memory instead of system memory, and the appropriate memory management calls were replaced by CUDA API calls, e.g., replacing `malloc` with `cudaMalloc`. For GPU-based designs, we evaluated three representative application kernels — NAS_MG, MILC, and Specfem3D_cm. NAS_MG and MILC represent dense non-contiguous layouts, while Specfem3D_cm represents a distributed layout with a higher number of small-sized blocks (higher fragmentation). Finally, we also present the evaluation of a GPU-enabled 2D-Stencil application. We ran each test five times, and each run has 110 iterations, 10 of which are warm-up iterations. We report the average latency. All the experiments were conducted on the two GPU clusters described in Table 2.

Fig. 13 shows the performance comparison of state-of-the-art CUDA-Aware MPI libraries and our proposed designs when performing DDT transfer between two GPUs connected by a PCIe switch. The proposed optimized zero-copy scheme MV2-GDR-Opt always performs the best for dense layout with large message sizes and distributed layout for all message sizes because of one-shot kernel launch to saturate the PCIe bandwidth and minimize the synchronization overhead. The proposed MV2-GDR-Opt yields up to 22× and 4.8× lower latency than Open MPI and MV2-GDR, respectively, for MILC. For a highly distributed layout such as Specfem3D_cm, MV2-GDR-Opt can achieve up to 5836× and 369× speedup than Open MPI and MV2-GDR, respectively. Fig. 14 shows similar experiments but with NVLink interconnect between the GPUs. Here, MV2-GDR-Opt outperforms other schemes in all message sizes. Moreover, we can observe a lower latency than the CS-Storm system. This is due to the low-latency *load-store* operations over NVLink and reduced kernel launch overhead on the latest NVIDIA Volta GPU architecture.

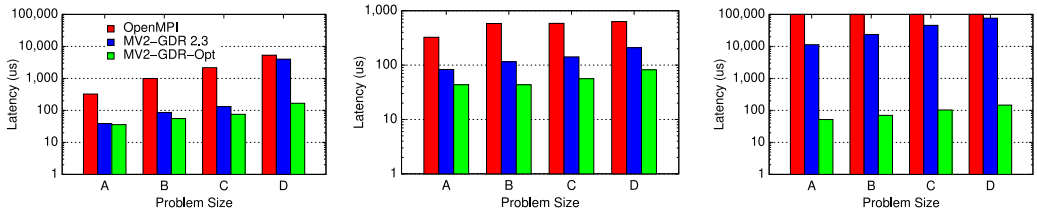
2D-Stencil: Finally, we conducted a two-dimensional and 9-point double-precision stencil computation test that consists of halo exchange using MPI vector datatype, i.e., for communicating data in 'East' and 'West' directions on the DGX-2 system. In this experiment, we report throughput in terms of GigaFlops(GFLOPS) over 1000 iterations while the grid size is kept constant at 1048 × 1048, resulting in a strong scaling trend.

Fig. 15 shows the performance benefits of the proposed design over baseline MVAPICH2-GDR. We also tried Open MPI for this experiment; however, the runs resulted in various runtime issues. As it can be seen from the results, that the proposed design is able to achieve up to 10% higher GFLOPS over default MVAPICH2-GDR by efficiently taking advantage of the zero-copy schemes over NVSwitch.

8. Related work

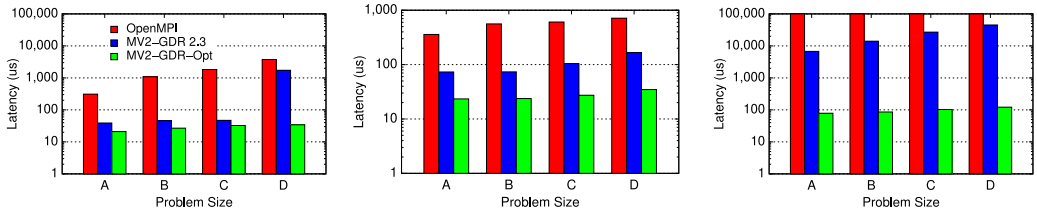
Researchers have explored network features to improve the performance of MPI datatype processing. Santhanaraman et al. [27] leveraged the Scatter/Gather List (SGL) feature to propose a zero-copy based scheme called SGRS. Li et al. [16] exploited User-mode Memory Registration (UMR) feature of Infiniband to remove the packing/unpacking overhead on the sender and receiver sides, which led to better performance. Their design also had lower memory utilization as it avoided the need for the intermediate staging of the data during pack/unpack operation. However, these designs rely on hardware features while our proposed designs emphasize intra-node communication. Perry et al. [25] proposed a compile-time transformation algorithm to reduce the non-contiguous datatype entities inside the application and avoid the copy overhead associated with pack/unpack operation. Schneider et al. [29] used several compilation techniques to generate an efficient and optimized packing code for MPI datatypes during the *commit* phase, which showed performance over manual packing code. Friedley et al. proposed different designs for shared memory-based communication in a multi-threaded MPI runtime [6,7], however, this work did not focus on MPI datatypes and its associated overheads. Gropp et al. [10] proposed concise datatype patterns and showed efficient implementation techniques for designing them. Ganian et al. [8] proposed a tree-like representation of MPI derived datatypes that are efficient in terms of space and processing cost and showed that their proposed design is capable of reconstructing a given datatype in polynomial time. In contrast, our proposed designs amortize or completely eliminate many of the layout translation and exchange overheads to improve the application performance transparently.

For the GPU-resident data, Wang et al. proposed a multi-stage pipeline of data transfer and offloading packing/unpacking processing from the host to the GPU to accelerate the non-contiguous data movement [36]. Jenkins et al. [13,14] presented means for converting conventional datatype representations into a GPU-amenable format and exploited fine-grained, element-level parallelism offered by a GPU kernel to perform in-device packing and unpacking of non-contiguous elements. However, this approach still remains focused on copy-based (packing/unpacking) designs while the proposed work completely eliminates extraneous copies involved in packing/unpacking by using efficient zero-copy mechanisms. Rong et al. [30] proposed a framework called *HAND* to employ datatype-specific GPU kernels to improve further the efficiency of packing and unpacking kernels. Chu et al. [4] achieved a higher overlap between CPU and GPU



(a) NAS_MG (b) MILC (c) Specfem3D_cm

Fig. 13. Performance comparison of non-contiguous data transfer when P2P access is available through PCIe on CS-Storm. Problem size for NAS_CG (A, B, C, D) = (3360, 62496, 256032, 1036320). Problems size for MILC (A, B, C, D) = 12288, 24576, 49152, 98304). Problem size for specfec3D_cm (A, B, C, D) = (26424, 55224, 106248, 177672).



(a) NAS_MG (b) MILC (c) Specfem3D_cm

Fig. 14. Performance comparison of non-contiguous data transfer when P2P access is available through NVLink on DGX-2. Problem sizes are the same as Fig. 13.

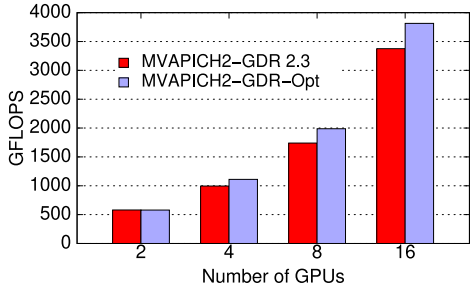


Fig. 15. Throughput of stencil computation on a DGX-2 machine. (Higher is better).

executions and eliminated the unnecessary synchronizations by using an asynchronous packing/unpacking solution. Wu et al. [39] proposed a different kernel-based design in Open MPI to offload unpacking and packing operations onto the GPU. However, their design fundamentally remained focused on optimizing the packing/unpacking by exploiting GPU resources. In contrast, our approach is fundamentally different since we use zero-copy mechanisms and address various limitations posed by zero-copy designs. Chu et al. [5] proposed adaptive selection strategies to choose between zero-copy and packing/unpacking schemes based on the system configurations and workload characteristics on the Dense-GPU systems. Their focus remained on the adaptive selection of various GPU-specific schemes to handle non-contiguous data-movement for GPU resident data. However, some of their designs still expose various overheads such as extra memory allocations for packing/unpacking, extraneous copies, and synchronization overheads. In contrast to their approach, our work mainly focuses on extending the ideas of host-based non-contiguous data-movement and demonstrates the applicability of our load/store based designs to dense GPU systems. As we have shown in this paper, while the GPU-based non-contiguous data-movement poses some additional challenges, the core ideas and designs for host-based data-movement are equally applicable to dense GPU systems as well when communication is carried out using the load/store semantics.

9. Conclusion and future work

In this paper, we identified the challenges involved in designing intra-node zero-copy communication schemes for MPI derived datatypes on modern multi-/many-core CPU and GPU architectures and proposed designs to address them efficiently. The proposed solutions, referred to as FALCON-X, reduced the cost of layout translation and exchange using novel designs based on pipelining and memoization. Finally, we propose enhancements to the MPI datatype creation semantics to enable future avenues for high-performance zero-copy datatype processing. We integrated our proposed FALCON-X designs in the variants of the MVAPICH2 library, namely MVAPICH2-X for CPU and MVAPICH2-GDR for GPU. We demonstrated their efficacy and performance improvement using various micro-benchmarks and applications on emerging multi-/many-core CPUs and modern GPU systems such as DGX-2. The proposed designs were able to reduce the intra-node point-to-point communication latency of MPI derived datatypes by up to 3×, and improved the communication performance of various application kernels such as 3D-Stencil, MILC, WRF, and NAS_MG by up to 5.5× on CPUs and up to 120× on DGX-2 system over state-of-the-art MPI libraries. Going forward, we plan to integrate our designs with inter-node zero-copy with multi-GPU clusters and evaluate their impact on applications at a larger scale.

CRediT authorship contribution statement

Jahanzeb Maqbool Hashmi: Conceptualization, Investigation, Methodology, Writing - original draft. **Ching-Hsiang Chu:** Conceptualization, Investigation. **Sourav Chakraborty:** Methodology, Writing - review & editing. **Mohammadreza Bayatpour:** Data curation, Methodology. **Hari Subramoni:** Supervision. **Dhabaleswar K. Panda:** Supervision, Validation, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

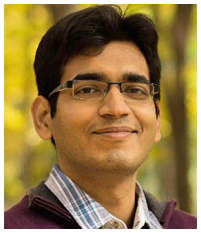
This research is supported in part by NSF grants #ACI-2007991, #CNS-1513120, #ACI-1450440, #CCF-1565414, #ACI-1664137, and #ACI-1931537. The authors would like to thank Dr. Sadaf Alam and Dr. Carlos Osuna for providing access to the CSCS testbed.

References

[1] Aurora supercomputer, http://aurora.alcf.anl.gov.
[2] M. Bayatpour, J.M. Hashmi, S. Chakraborty, H. Subramoni, P. Kousha, D.K. Panda, SALaR: Scalable and adaptive designs for large message reduction collectives, in: 2018 IEEE International Conference on Cluster Computing, CLUSTER, IEEE, 2018, pp. 12–23.
[3] S. Chakraborty, H. Subramoni, D. Panda, Contention aware kernel-assisted MPI collectives for multi/many-core systems, in: 2017 IEEE International Conference on Cluster Computing, 2017.
[4] C.-H. Chu, K. Hamidouche, A. Venkatesh, D.S. Banerjee, H. Subramoni, D.K. Panda, Exploiting maximal overlap for non-contiguous data movement processing on modern GPU-enabled systems, in: 2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2016, pp. 983–992.
[5] C.-H. Chu, J.M. Hashmi, K.S. Khorassani, H. Subramoni, D.K. Panda, High-performance adaptive MPI derived datatype communication for modern multi-GPU systems, in: 2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics, HiPC, IEEE, 2019, pp. 267–276.
[6] A. Friedley, G. Bronevetsky, T. Hoefler, A. Lumsdaine, Hybrid MPI: Efficient message passing for multi-core systems, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, ACM, 2013, p. 18.
[7] A. Friedley, T. Hoefler, G. Bronevetsky, A. Lumsdaine, C.-C. Ma, Ownership passing: Efficient distributed memory programming on multi-core systems, in: ACM SIGPLAN Notices, Vol. 48, (8) ACM, 2013, pp. 177–186.
[8] R. Ganian, M. Kalany, S. Szeider, J.L. Träff, Polynomial-time construction of optimal MPI derived datatype trees, in: 2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS, IEEE, 2016, pp. 638–647.
[9] B. Goglin, S. Moreaud, KNEM: A generic and scalable kernel-assisted intra-node MPI communication framework, J. Parallel Distrib. Comput. 73 (2) (2013) 176–188.
[10] W. Gropp, E. Lusk, D. Swider, Improving the performance of MPI derived datatypes, in: Proceedings of the Third MPI Developer's and User's Conference, MPI Software Technology Press, 1999, pp. 25–30.
[11] J.M. Hashmi, S. Chakraborty, M. Bayatpour, H. Subramoni, D.K. Panda, Designing efficient shared address space reduction collectives for multi-/many-cores, in: 2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS, IEEE, 2018, pp. 1020–1029.
[12] J.M. Hashmi, S. Chakraborty, M. Bayatpour, H. Subramoni, D.K. Panda, FALCON: Efficient designs for zero-copy MPI datatype processing on emerging architectures, in: 2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS, IEEE, 2019, pp. 355–364.
[13] J. Jenkins, J. Dinan, P. Balaji, T. Peterka, N.F. Samatova, R. Thakur, Processing MPI derived datatypes on noncontiguous GPU-resident data, IEEE Trans. Parallel Distrib. Syst. 25 (10) (2014) 2627–2637.
[14] J. Jenkins, J. Dinan, P. Balaji, N.F. Samatova, R. Thakur, Enabling fast, noncontiguous GPU data movement in hybrid MPI + GPU environments, in: 2012 IEEE International Conference on Cluster Computing, 2012, pp. 468–476.
[15] H.-W. Jin, S. Sur, L. Chai, D.K. Panda, Limic: Support for high-performance MPI intra-node communication on Linux cluster, in: 2005 International Conference on Parallel Processing, ICPP'05, IEEE, 2005, pp. 184–191.
[16] M. Li, H. Subramoni, K. Hamidouche, X. Lu, D.K. Panda, High performance MPI datatype support with user-mode memory registration: Challenges, designs, and benefits, in: Cluster Computing. CLUSTER, 2015 IEEE International Conference on, IEEE, 2015, pp. 226–235.
[17] Linux Kernel, Cross memory attach, https://lwn.net/Articles/405284/. (Online; Accessed April 21, 2020).
[18] MIMD lattice computation (MILC), http://physics.indiana.edu/~sg/milc.html. (Online; Accessed April 21, 2020).
[19] Message passing interface forum, MPI: A message-passing interface standard, 1994.
[20] MVAPICH2: MPI over infiniband, 10GigE/iWARP and RoCE, https://mvapich.cse.ohio-state.edu/. (Online; Accessed April 21, 2020).
[21] NAS parallel benchmarks, https://www.nas.nasa.gov/publications/npb.html. (Online; Accessed April 21, 2020).
[22] NVIDIA, NVIDIA GPUDirect, 2019, URL https://developer.nvidia.com/gpudirect. (Accessed: April 21, 2020).
[23] OSU micro-benchmarks, http://mvapich.cse.ohio-state.edu/benchmarks/. (Online; Accessed April 21, 2020).
[24] K. Pedretti, B. Barrett, XPMEM: Cross-process memory mapping, https://gitlab.com/hjelmn/xpmem. (Online; Accessed April 21, 2020).
[25] B. Perry, M. Swany, Improving MPI communication via data type fission, in: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC '10, ACM, New York, NY, USA, 2010, pp. 352–355, http://dx.doi.org/10.1145/1851476.1851528.
[26] S. Potluri, H. Wang, D. Bureddy, A.K. Singh, C. Rosales, D.K. Panda, Optimizing MPI communication on multi-GPU systems using CUDA inter-process communication, in: Parallel and Distributed Processing Symposium Workshops PhD Forum, IPDPSW, 2012 IEEE 26th International, 2012, pp. 1848–1857.
[27] G. Santhanaraman, J. Wu, D.K. Panda, Zero-copy MPI derived datatype communication over infiniband, in: European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting, Springer, 2004, pp. 47–56.
[28] T. Schneider, R. Gerstenberger, T. Hoefler, Micro-applications for communication data access patterns and MPI datatypes, in: European MPI Users' Group Meeting, Springer, 2012, pp. 121–131.
[29] T. Schneider, F. Kjolstad, T. Hoefler, MPI datatype processing using runtime compilation, in: Proceedings of the 20th European MPI Users' Group Meeting, ACM, 2013, pp. 19–24.
[30] R. Shi, X. Lu, S. Potluri, K. Hamidouche, J. Zhang, D.K. Panda, HAND: A hybrid approach to accelerate non-contiguous data movement using MPI datatypes on GPU clusters, in: 2014 43rd International Conference on Parallel Processing, 2014, pp. 221–230.
[31] R. Shi, S. Potluri, K. Hamidouche, J. Perkins, M. Li, D. Rossetti, D.K. Panda, Designing efficient small message transfer mechanism for inter-node MPI communication on infiniband GPU clusters, in: 2014 21st International Conference on High Performance Computing, HiPC, 2014, pp. 1–10.
[32] SPECFEM 3D, https://geodynamics.org/cig/software/specfem3d/. (Online; Accessed April 21, 2020).
[33] V. Tipparaju, G. Santhanaraman, J. Nieplocha, D. Panda, Host-assisted zero-copy remote memory access communication on infiniband, in: Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, IEEE, 2004, p. 31.
[34] J.L. Träff, R. Hempel, H. Ritzdorf, F. Zimmermann, Flattening on the fly: Efficient handling of mpi derived datatypes, in: European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting, Springer, 1999, pp. 109–116.
[35] H. Wang, S. Potluri, D. Bureddy, C. Rosales, D.K. Panda, GPU-Aware MPI on RDMA-enabled clusters: Design, implementation and evaluation, IEEE Trans. Parallel Distrib. Syst. 25 (10) (2014) 2595–2605.
[36] H. Wang, S. Potluri, M. Luo, A. Singh, X. Ouyang, S. Sur, D. Panda, Optimized non-contiguous MPI datatype communication for GPU clusters: design, implementation and evaluation with MVAPICH2, in: Cluster Computing, CLUSTER, 2011 IEEE International Conference on, 2011, pp. 308–316.
[37] H. Wang, S. Potluri, M. Luo, A.K. Singh, S. Sur, D.K. Panda, MVAPICH2-GPU: Optimized GPU to GPU communication for infiniband clusters, Comput. Sci.-Res. Dev. 26 (3–4) (2011) 257.
[38] WRF: Weather research and forecasting model, https://www.mmm.ucar.edu/weather-research-and-forecasting-model. (Online; Accessed April 21, 2020).
[39] W. Wu, G. Bosilca, R. vandeVaart, S. Jeaugey, J. Dongarra, GPU-Aware non-contiguous data movement in open MPI, in: Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, HPDC '16, ACM, New York, NY, USA, 2016, pp. 231–242.
[40] J. Wu, P. Wyckoff, D. Panda, High performance implementation of MPI derived datatype communication over infiniband, in: 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings, IEEE, 2004, p. 14.

Jahanzeb Maqbool Hashmi is a Ph.D. candidate at The Ohio State University where he works at Network-Based Computing Laboratory (NBCL). Before joining NBCL, he was a graduate fellow at the Department of Computer Science and Engineering, Ohio State University. His research is mainly targeted at enabling high-level programming models and runtime systems to achieve high-performance and scalability on dense CPU and GPU architectures. He works on the areas related to shared-memory and shared-address-space communication, topology-aware communication protocols, high-performance deep learning, and optimizations for MPI, PGAS, and MPI+X programming models on modern CPU, accelerators, and interconnects. Prior to joining OSU, he completed his MS in computer engineering at Ajou University, South Korea under the Korean Global IT fellowship where he worked on the performance evaluation and characterization of energy-efficient clusters for scientific workloads. He received his BS from National University of Science and Technology, Pakistan under the Prime Minister's ICT fellowship.

Ching-Hsiang Chu is a Ph.D. candidate in Computer Science and Engineering at The Ohio State University, Columbus, Ohio, U.S.A. He received B.S. and M.S. degrees in Computer Science and Information Engineering from National Changhua University of Education, Taiwan in 2010 and National Central University, Taiwan in 2012, respectively. His research interests include high-performance computing, GPU communication, and wireless networks. He is a student member of IEEE and ACM. More details are available at http://web.cse.ohiostate.edu/~chu.368.



Sourav Chakraborty graduated with a Ph.D. from Ohio State University where he worked at Network Based Computing Laboratory (NBCL). His research interests include Highperformance Computing, MPI and PGAS programming models, kernel assisted collective communication, and designing MPI protocols for emerging architectures. He had made contributions to the MVAPICH2 and MVAPICH2-X projects that are used by wider HPC community.

Mohammadreza Bayatpour is a 5th year Ph.D. student at Ohio State University in Computer Science and Engineering Department. His research interests are High-Performance Networking and Computing, Scalable Distributed Systems, Parallel Programming Models, and In-Network Computing.

Hari Subramoni received the Ph.D. degree in Computer Science from The Ohio State University, Columbus, OH, in 2013. He is a research scientist in the Department of Computer Science and Engineering at the Ohio State University, USA, since September 2015. His current research interests include high performance interconnects and protocols, parallel computer architecture, network-based computing, exascale computing, network topology aware computing, QoS, power-aware LAN-WAN communication, fault tolerance, virtualization, big data, and cloud computing. He has published over 50 papers in international journals and conferences related to these research areas. Recently, Dr. Subramoni is doing research and working on the design and development of MVAPICH2, MVAPICH2-GDR, and MVAPICH2-X software packages. He is a member of IEEE. More details about Dr. Subramoni are available from http://www.cse.ohio-state.edu/~subramon.

Dhabaleswar K. Panda is a Professor and University Distinguished Scholar of Computer Science and Engineering at The Ohio State University. He has published over 450 papers in major journals and international conferences. The MVAPICH2 (High Performance MPI over InfiniBand, iWARP and RoCE) open-source software package, developed by his research group (http://mvapich.cse.ohiostate.edu), are currently being used by more than 3,075 organizations worldwide (in 89 countries). This software has enabled several InfiniBand clusters to get into the latest TOP500 ranking during the last decade (including the current #3). More than 756,000 downloads of this software have taken place from the project's website alone. He is an IEEE Fellow and a member of ACM. More details about him are available from http://web.cse.ohio-state.edu/~panda.2/.