Network Assisted Non-Contiguous Transfers for GPU-Aware MPI Libraries

Kaushik Kandadi Suresh, Kawthar Shafie Khorassani, Chen Chun Chen, Bharath Ramesh,
Mustafa Abduljabbar, Aamir Shafi, Hari Subramoni, Dhabaleswar K. Panda
Department of Computer Science and Engineering
The Ohio State University
Columbus, USA
{kandadisuresh.1, shafiekhorassani.1, chen.10252, ramesh.113, abduljabbar.1, shafi.16, subramoni.1, panda.2}@osu.edu

***Abstract*—The importance of GPUs in accelerating HPC applications is evident by the fact that a large number of supercomputing clusters are GPU-enabled. Many of these HPC applications use MPI as their programming model. These MPI applications oftentimes exchange data that is non-contiguous in GPU memory. MPI provides Derived Datatypes(DDTs) to represent such data. In the past, researchers have proposed solutions to optimize these MPI DDT based inter-node GPU exchanges. All of these solutions are aimed at optimizing the overheads associated with pack-unpack kernels that facilitate the non-contiguous exchanges. Modern HCAs are capable of gathering/scattering data from/to non-contiguous GPU memory regions. In this work, we analyze the challenges in using HCA's scatter/gather mechanism for GPU-based HPC workloads. We propose a low-overhead HCA-assisted scheme to improve the performance of GPU-based non-contiguous exchanges. We show that the proposed scheme provides up to 2X benefits compared to existing pack-based schemes at the benchmark level. Furthermore, on the layouts used by MILC, NASMG, Specfem3D applications, we show that the proposed scheme outperforms the state-of-the MPI libraries such as MVAPICH2-GDR and OpenMPI+UCX.**

***Index Terms*—MPI, DDT, GPU**

I. Introduction

Graphics Processing Units (GPUs) have become ubiquitous in modern supercomputers due to their high compute capability and power efficiency. This is particularly evident in the growing number of top supercomputers on the Top500 [17] list deploying GPUs on their clusters.

In these super-computing clusters, Message Passing Interface (MPI) is a widely adopted programming model for several large-scale GPU-based applications. Oftentimes, applications are required to exchange data that is non-contiguous in memory. MPI provides Derived Data Types (DDTs) that allow an application to represent any non-contiguous layout in memory. Table I gives a summary of access patterns and possible data-types of different HPC applications. As shown in Table I, applications often involve a variety of complex data exchange patterns and use multiple types of MPI DDT layouts. This underscores the need for MPI libraries to optimize the exchange of such non-contiguous data layouts represented by DDTs.

*This research is supported in part by NSF grants #1818253, #1854828, #1931537, #2007991, #2018627, and XRAC grant #NCR-130002.
Thanks to ALCF for providing resources for this study.

TABLE I
Summary of Datatypes used in HPC applications

Applications	MPI DDTs used	Data Exchange Pattern
NAS	MPI_Type_Vector	2D,3D face exchange
MILC	MPI_Type_Vector, MPI_Type_Contiguous	4D face exchange
Specfem3D	MPI_Type_Vector, MPI_Type_Indexed	unstructured exchange

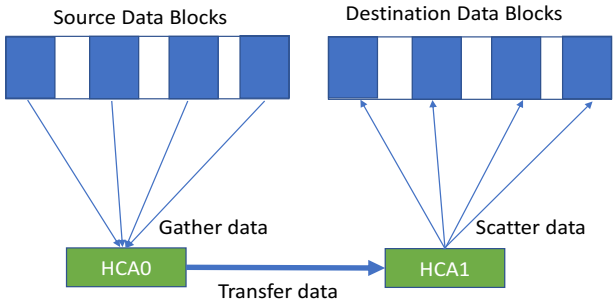


Fig. 1. HCA Assisted Exchange of Non-Contiguous Data. HCA0 gathers data blocks from source memory and sends it to HCA1. HCA1 scatters the received data to the destination memory at appropriate locations

MPI libraries typically use pack-unpack kernels for inter-node DDT based exchanges. All the preceding studies on DDT based inter-node optimizations have either optimized 1) pack-unpack kernels [5] or 2) overlapped kernels with transfers/other kernels for inter-node GPU-to-GPU transfers [22]. While pack/unpack kernels can be an effective approach to transfer DDT messages, they involve an additional step of packing/unpacking every buffer on the sender/receiver side. Modern HCAs provide the capability of transferring non-contiguous data directly from source buffers to the destination buffers without using the pack-unpack kernels. As shown in figure 1, given a list of source buffer addresses and destination buffer addresses, the source HCA can gather data from these memory regions and send them to the destination HCA. Once the destination HCA receives the data, it can scatter

these to their respective memory regions. None of the past research on GPU DDT optimization explored this possibility of doing non-contiguous transfers using HCA assisted scatter-gather mechanisms. There are overheads associated with such transfers such as the cost of registering the layouts with HCAs. In this work, we identify all the challenges in using the HCA assisted mechanisms for moving GPU resident non-contiguous data and design a low-overhead HCA assisted data-transfer scheme that performs better than pack based schemes for certain application layouts.

II. Background

A. MPI Derived Datatypes

Derived datatypes (DDT) are used in MPI in order to group and then communicate data that is either noncontiguous or data that are differing in type. MPI provides various functions for derived datatypes to represent different types including MPI_Type_Contiguous, MPI_Pack, MPI_Unpack, MPI_Type_Vector, and etc. For example, MPI_Type_Vector is a function that will take the following values as parameters and utilize this information to create a vector datatype: the number of blocks, the block length, the stride, a handle representing the old datatype, and a handle representing the new datatype.

B. UMR

User-Mode Memory Registration (UMR) is a registration mode introduced by Infiniband and utilized for communication of non-contiguous data. This feature can enable direct communication of MPI derived datatypes(II-A) without requiring any additional packing or unpacking schemes. A list of lkeys and rkeys are created through registering memory regions and the mkey is then mapped to this list. UMR uses send queues and enables changing the address translation of mKeys [18].

C. MPI Inter-Node Communication Protocols

Various communication protocols exist within MPI and are utilized for different configurations of communication. In particular, the rendezvous protocol involves a transfer of data that utilizes the Infiniband RDMA feature in order to directly transfer data from the buffer at the source to the destination. This eliminates the overhead of copying large regions of memory to the buffer involved in the communication. The sender process registers the buffer and sends a RTS (Request to Send) packet to the receiver process. Once the receiving process posts the process to the HCA, it will then initiate a CTS (Clear to Send) packet back to the sender, to post the send on the HCA. This will then trigger an RDMA transfer between the sender and receiver.

For GPU-based communications in a GPU-aware MPI library, GPUDirect RDMA is a common protocol for data transfer between GPUs. GPUDirect RDMA, introduced by NVIDIA, provides a direct path between the GPU and the network interface sharing the same PCIe root complex to communicate data. This eliminates large overhead involved in communication between two GPUs by removing the CPU from the critical path.

D. Commonly used pack based GPU schemes for DDT based transfers

Several different packing schemes currently exist in the state-of-the-art MPI libraries and the literature. The first scheme, referred to here as pack-host, involves the host in the packing process. The packing operation is done onto the host, where a data transfer then occurs between two CPUs before unpacking it to the GPU. The second scheme, referred to here as pack-stage, involves the packing happening to the GPU in the first step. The data is then moved from GPU to CPU and transferred between CPUs where it is then unpacked to the destination GPU. Finally, the third scheme, referred to here as pack-gdr utilizes GPUDirect RDMA to transfer between GPUs and does not involve the host in the data transfer path. The packing is done to the source GPU, GPUDirect RDMA is then utilized to transfer the data between GPUs, where it is then unpacked at the destination GPU.

III. Motivation

A. Pack Cost in Application Layouts

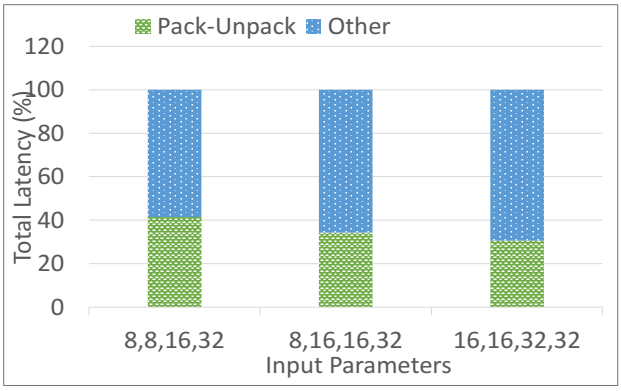


Fig. 2. Pack-Unpack Cost as a percentage of total latency in a ping-pong benchmark which exchanges non-contiguous layouts used in MILC application between source and destination GPU buffers. The data at the sender side packed and sent. The data at the receiver's side is unpacked to the final destination buffer. The input parameters represent the grid dimensions used by the MILC application.

In this section, we analyze the pack cost incurred in application layouts. We wrote an application kernel with MPI Derived Datatypes based on DDTBench which exchanges data from GPU resident buffers. We profiled the amount of time spent in pack-unpack routines inside an MPI library to exchange some non-contiguous datatype layouts used in the MILC application. Figure 2 shows the percentage of time spent in pack-unpack operations out of the total time to exchange data for three different input parameters. We observe that depending on the input parameters, the pack-unpack cost could be as high as 40% of the total exchange time. **Given this information, we strive to know if we can leverage the HCA's scatter/gather mechanism to exchange non-contiguous data between MPI ranks.**

B. Overheads in the Hardware Assisted Scheme

State-of-the-art NVIDIA HCAs support non-contiguous RDMA operations through the User Mode Memory Reg-

istration (UMR) feature. This feature allows a program to directly exchange non-contiguous data from a set of source buffers to a set of destination buffers using a single post operation. This is shown in figure 1 where the source HCA gathers the non-contiguous data blocks and transfers them to the destination HCA. Then, the destination HCA scatters the data to the destination memory addresses. However, this operation requires the user to create a mkey and map the set of non-contiguous buffers with the mkey and subsequently use that mkey for posting send operations. To understand the cost of these operations we wrote an IB level benchmark that exchanges non-contiguous layouts used in the MILC application. We used UMR to exchange the non-contiguous data. We observed that the creation of a single mkey takes about 200us. In figure 3 we show the time spent in mapping the UMR mkeys to the layout as a percentage of the total time taken to do RDMA-Write operation of non-contiguous data. We observe the percentages of UMR overhead are similar to the pack costs that we presented in the section III-A. However, applications tend to re-use a particular layout multiple times. **Can we leverage this information to amortize the overhead of mapping mkeys to a layout in UMR based transfers?**

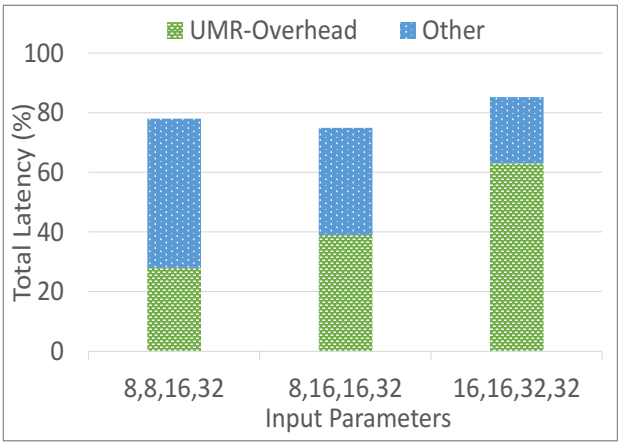


Fig. 3. UMR Overhead as a percentage of total latency in a ping-pong benchmark which exchanges non-contiguous layouts used in the MILC application between source and destination GPU buffers. The UMR overhead refers to the cost of mapping mkeys to a particular layout. The input parameters represent the grid dimensions used by the MILC application.

C. Amortizing the mkey exchange overhead

By mapping a single mkey to a set of buffers, we can exchange all the buffers using a single ibv_post operation. However, the number of buffers associated with a single mkey is limited. Therefore, one needs to use multiple mkeys depending on the number of blocks in a non-contiguous layout. For a process to do RDMA-Write to a remote process, the local process needs the list of remote process' mkeys. To understand the effect of this exchange, let us consider a layout with 4096 blocks. Assuming the HCA can support 4 blocks per mkey, this would require 1024 keys which can amount to 4KB of data exchange. This can have a significant impact on the performance of medium message transfers. However, the applications tend to re-use many layouts. This brings us to the next challenge where we ask: **Given the temporal repetition of layouts in the application how can we amortize this mkey exchange?**

D. Contributions

In this work, we motivate the need for a hardware-assisted inter-node transfer mechanism for GPU resident non-contiguous memory layouts by analyzing the pack costs of application layouts. Driven by this motivation, we identify the challenges with a hardware-assisted mechanism called UMR and propose a design that addresses the above challenges.

To summarize, this paper makes the following contributions:

1) Survey of the existing mechanisms for inter-node exchange of non-contiguous data using MPI Derived Datatypes.
2) Motivate the need for HCA-assisted non-contiguous data transfers by profiling the layout of the MILC application.
3) Propose a UMR-based design for exchanging non-contiguous data.
4) Enhance the proposed design by using caching mechanisms to amortize the overheads associated with the UMR scheme.
5) Demonstrate the usefulness of the proposed schemes by comparing the performance of the proposed designs on real application layouts in GPU-based HPC clusters.

IV. Design and Implementation

In this section, we discuss our network based non-contiguous transfer design (Proposed-UMR).

A. Rendezvous Protocol

We employ RPUT protocol for all our designs. In this protocol, the sender first sends Request-To-Send (RTS) packet to the receiver. This RTS packet may contain the sender's buffer information depending on the design. After receiving the RTS packet, the receiver sends a Clear-To-Send (CTS) packet to the sender. This packet may contain information that is useful for data exchange. After receiving the CTS packet, the sender transfers data and sends a FIN packet to signal the completion. Now, to transfer a contiguous buffer using the above protocol, the sender first registers the send GPU buffer with the HCA to obtain a lkey, and a rkey. The receiver also registers the receive buffer to obtain the lkey and the rkey. Once the receiver gets the RTS, it sends the receive buffer address, data-size, and rkey to the sender in the CTS packet. Now, the sender uses this buffer address, data-size, and rkey to post an RDMA-Write to the receiver.

B. DDT Processing in Rendezvous Protocol

When derived datatypes (DDT) are used in MPI calls, first the sender/receiver parses the DDT handle to get a list of IOVs. An IOV is a structure that contains the address and length of a contiguous memory block. The number of IOVs in a layout indicates the number of blocks in that layout. Once the sender gets the list of IOVs for a given DDT handle,

as a next step a pack kernel is launched which packs all the memory regions represented by the IOVs to a single contiguous memory region called pack-buffer. Next, the data from this pack-buffer is transferred to a remote pack-buffer using the RPUT mechanism described earlier (Section IV-A). Once the receiver receives data in the remote pack-buffer, it will launch an unpack kernel which transfers data from the pack-buffer to the memory regions pointed by individual IOVs.

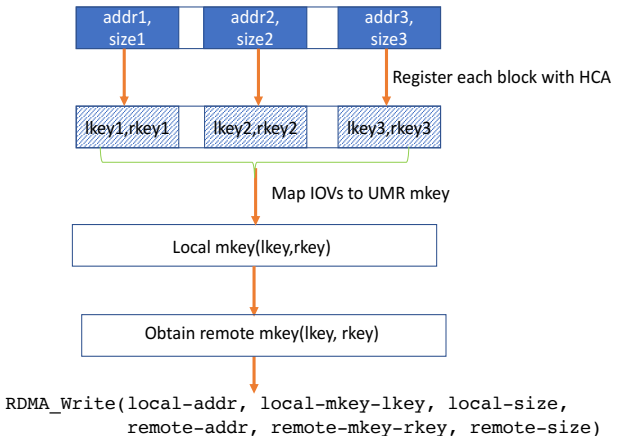


Fig. 4. Basic sender-side flow showing the steps involved in making a non-contiguous buffer ready for UMR-based HCA-assisted RDMA operations

C. Mkey Mapping and Exchange

Figure 4 depicts how UMR can be used for non-contiguous data exchange. Given a list of memory addresses and sizes, first, these memory regions are registered with the HCA which generates a list of lkeys and rkeys. Then UMR mkey is created and then mapped to the list of addresses. This mkey object contains one lkey and one rkey that refers to the entire memory region. This composite lkey and rkey can be used for posting RDMA operations provided we have the corresponding mkey based rkey of a destination buffer address. The HCA is responsible for gathering data referred to by the newly created mkey from the local node and scattering data to the remote node according to the mapping in the remote process's mkey.

As described in section III, we can only map a limited number of blocks/IOVs to a single mkey. Therefore, we use a list of mkeys to represent a single layout. To simplify the data exchange process, we first fix a chunk-size for a given layout. Then, we use a moving window-based approach to map mkeys with the IOVs. A window of size chunk-size starts at IOV-0, and spans all the IOVs whose collective sum of size is the chunk-size. The first mkey is mapped to the IOVs spanned by the window. After the first mkey is mapped, the window is moved by an offset of chunk-size. Now the next mkey is mapped to the IOVs under the current window. This process continues until we exhaust all the IOVs. Note that the window does not span the gaps between blocks, it operates at the IOV level. This process is shown in figure 5.

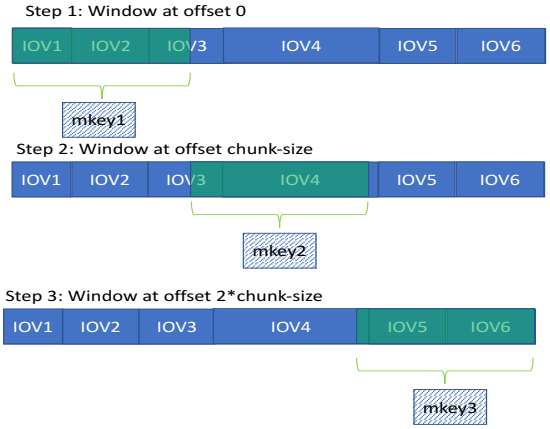


Fig. 5. Sliding window based approach to map mkeys to IOV list. At each step the IOVs covered by the window is mapped to a new mkey. This way the entire layout is mapped to a set of mkeys which represent the layout.

When we use the above approach to map a list of IOVs with a list of mkeys, we also add a constraint of the max number of IOVs per mkey to ensure that our RDMA operation does not fail.

D. UMR based Design

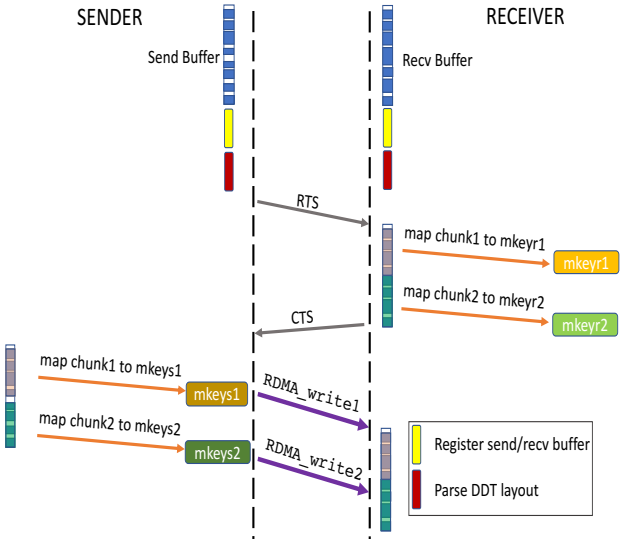


Fig. 6. Steps involved in the UMR design between a Sender and Receiver. The figure shows how one send/recv buffer can have multiple mkeys representing regions in memory, and the independent transfer of each region using RDMA operations.

Since the sender and receiver's layouts may not be identical it is necessary to create and map mkeys on the sender and receiver's side in a co-operative manner to ensure data validation. In our protocol, the sender first selects a chunk-size based on local layout and sends it to the receiver. The receiver then arrives at a chunk-size based on the sender's chunk-size and its layout. This agreed chunk-size is sent to the sender. Now, both sender and receiver will create mkeys based on the agreed chunk-size. This will result in the same number of

mkeys on the sender and receiver's sides. A single mkey in the sender side will be used to post an RDMA-Write operation using a corresponding remote mkey from the receiver side. This way the responsibility of gathering and scattering data from any type of source layout to any type of destination layout is given to the source and the destination HCAs.

Figure 6 illustrates the various steps involved in our UMR design. First, the sender/receiver registers the entire send/receive buffer to obtain the lkey and the rkey. Then the sender sends its chunk-size to the receiver. After receiving the sender's layout information, the receiver arrives at an agreed chunk-size and it creates and maps mkeys the receiver's IOVs with this agreed chunk-size. These mkeys and agreed chunk-size values are sent to the sender in the CTS packet. Then, the sender creates and maps a set of mkeys to its layout based on the agreed chunk-size value obtained in the CTS packet. Then, the sender uses the remote mkey's rkey to post RDMA-Write operations. The number of RDMA-Write operations is equal to the number of mkeys created.

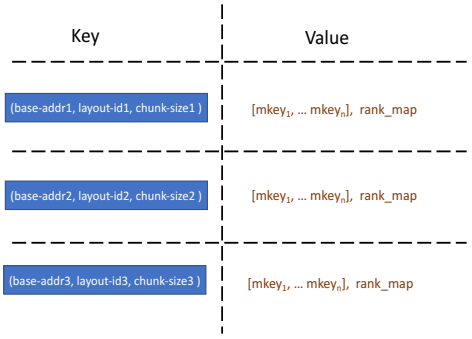


Fig. 7. UMR local mkey cache. This cache is a hash-table which has keys generated from the base address of the send/recv buffer, layout-ID, and chunk-size of each transfer. The value is a tuple with the list of mkeys which are mapped to the layout, and a rank_map which stores the remote ranks to which the mkeys were sent to (used by the receiver).

E. Enhancing the UMR design

As discussed in section III, mkey creation is an expensive operation. Therefore we create a pool of mkeys at the time of MPI_Init. During the run of the application, a sender/receiver obtains a mkey from the pool and uses it for mapping to IOVs. If at any point the size of the free pool is reduced to 50%, an auxiliary thread is signaled which creates and adds a fixed number of mkeys to the pool. This is done to ensure that the main thread does not get impacted if it runs out of mkeys.

We use a layout-cache [4] to amortize the layout parsing cost. In the above design, the sender and receiver use the same set of layouts multiple times, each time the receiver performs UMR based registrations and exchanges the list of mkeys with the sender. To amortize the UMR registration cost and the mkey exchange cost, we propose a UMR mkey cache.

We propose two kinds of mkey cache. First, is a local cache which is to avoid re-mapping of mkeys to the layouts. The second is a remote cache maintained at the sender side to cache the remote mkeys. This cache is used by the sender when the receiver's layout was already sent to it at an earlier exchange.

Figure 7 depicts the UMR mkey cache. It is implemented as a hash-table that is indexed by local layout-cache-id, local base address, and agreed chunk-size. Each entry of the cache stores a list of mkeys that are uniquely identified by the above three parameters. In addition to the mkey list, the local cache stores a bitmap. This bitmap is used by the receiver to store the ranks to which the mkey list was sent. On the sender's side, this bitmap is not used.

The remote mkey cache is implemented as an array of hash-tables where the array is indexed by remote-rank. Each entry of an array has a structure similar to the local cache which is a hash-table indexed by the remote layout-cache-id, remote base address and agreed-chunk-size. This is used at the sender's side when it does not receive the mkey list from the receiver.

F. Understanding the performance of non-contiguous exchange

We evaluate the performance of the above designs with a simple vector-based ping-pong latency benchmark. We use a block size of 1KB and a stride of 2KB between consecutive blocks. Figure 8 compares the performance of the basic UMR scheme and UMR with mkey map cache for a different number of blocks.

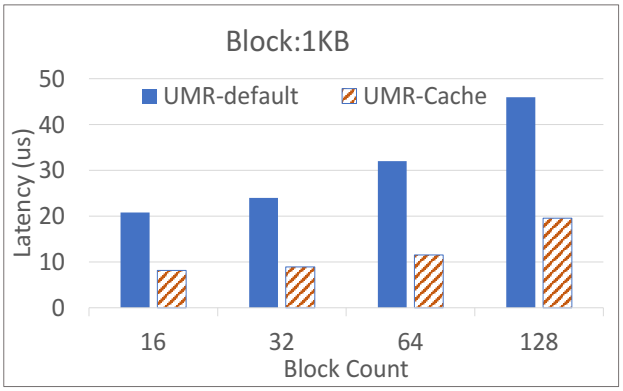


Fig. 8. Impact of UMR mkey cache on a vector of block size 1KB. The benchmark used is a modified version of OMB which exchanges vector layouts of a given block and count. The UMR-default is a basic design proposed in section IV. UMR-cache is the enhanced version of the design where UMR-pool, UMR local and remote cache are used to amortize mkey creation, mapping and exchange costs respectively.

We observe that as the block count increases the performance of default UMR becomes much worse, for instance, at 16 blocks default UMR is 2X worse compared to the cached UMR and at 64 blocks it becomes 3X worse. Both mkey mapping and mkey exchange contribute to the degradation, however, the mkey exchange cost increases with an increase in the number of blocks.

V. Performance Evaluation

In this section, we compare the performance of the proposed scheme against other existing pack-based schemes on GPU-based clusters. We also compare the proposed scheme with the state-of-the-art MPI libraries.

A. Experimental Platforms and Setup

We use MRI and ThetaGPU clusters for our evaluations. MRI is an in-house cluster of 8 nodes with AMD-EPYC processors and A100 NVIDIA GPU nodes. The ThetaGPU cluster, deployed at the Argonne Leadership Computing Facility (ALCF), contains 24 DGX-A100 nodes with AMD-EPYC processors. The NVIDIA DGX A100 GPU has 40GB HBM2. The GPUs are connected with the third generation NVIDIA NVLink and the second generation NVIDIA NVSwitch. The detailed hardware specifications of these clusters are shown in table II

We implemented the proposed scheme (UMR) in MVAPICH2-GDR using MPI derived datatypes (DDT) and we compare this with other pack based MPI-DDT schemes in our MPI library. We compare pack-gdr and pack-staged with our scheme. Details of these pack scheme are mentioned in section II-D.

TABLE II
HARDWARE SPECIFICATION OF DIFFERENT TEST-BED CLUSTERS

Specification	MRI	ThetaGPU
Processor Family	AMD EPYC	AMD EPYC
Processor Model	EPYC 7713	EPYC 7742
Clock Speed	2.0 GHz	3.4 GHz
Sockets	2	2
Cores Per socket	64	64
NUMA nodes	2	8
CCX Per NUMA	8	4
RAM (DDR4)	256 GB	1 TB
Interconnect	IB-HDR(200G) - 1 HCA	IB-HDR(200G) - 8 HCAs
GPU Processor	NVIDIA A100×4	NVIDIA A100×8
GPU Memory	40GB	40GB
Interconnects between GPUs	PCIe	NVLink-3 and NVSwitch
NVIDIA Driver Version	510.39.01	470.82.01

First, to understand the performance of a simple non-contiguous layouts, we modified the osu_latency test provided by the OSU Micro-Benchmarks (OMB) suite [13] to support MPI_Type_Vector datatype. In this benchmark, a simple vector layout of a given block length and count is exchanged in a ping-pong manner for a given number of iterations.

Then, we evaluate application kernels with representative application layouts. We implemented the GPU-enabled application kernels based on popular benchmarks including ddtbench [15] and a kernel of 3D domain decomposition [11]. For all our experiments, we report an average of 100 iterations, excluding the 10 warm-up iterations.

B. Microbenchmark Results

In this section, we first evaluate the modified osu_latency vector benchmark for block sizes of 1KB, 2KB, 4KB. For each of these block sizes, we vary the number of segments from 16 to 128. These block lengths are representative of some of the layouts in halo exchange based applications. Figures 10(a), 10(b), 10(c) show the results on the MRI cluster. We observe that UMR performs up-to 2X better than pack-GDR and UMR performs up to 3X better than pack-staged. The HCA's ability to scatter/gather data directly to GPU buffers coupled with our UMR cache design, which ensures that expensive operations like UMR-registration and mkey exchange happen only once, enable the UMR scheme to outperform pack-based schemes.

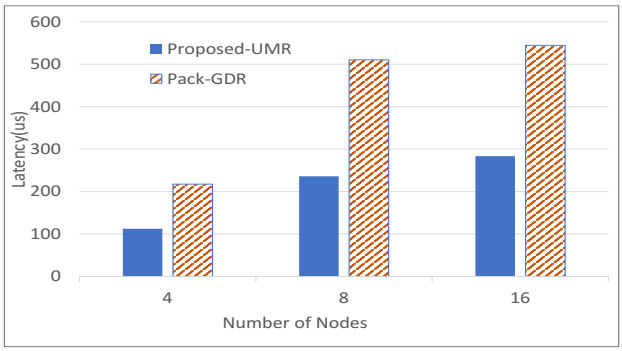


Fig. 9. 3D-Stencil benchmark performance on ThetaGPU nodes

C. 3D-Stencil Communication Benchmark

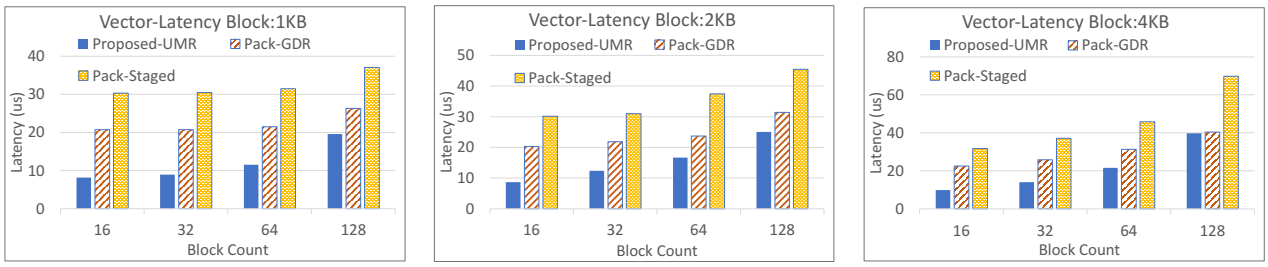
In this section, we evaluate the performance of a GPU based 3D stencil communication benchmark. 3D Stencil benchmark follows a near-neighbor communication pattern which is common in HPC applications. In this benchmark, each process sends and receives non-contiguous data buffers represented by MPI_Type_subarray derived datatype to/from at most 6 neighbors. We ran this benchmark for a problem size of 128X128X128 with 1 GPU per-node scaling from 4 to 16 nodes. For the above problem size, the block sizes used are 8 bytes in the X direction, 1KB in the Y direction. Z-direction datatype is a contiguous send of size 128KB. The results are shown in figure 9. We have not added results for pack-staged because it performed worse compared to pack-gdr. We observe that the proposed scheme is 2X better than pack-gdr.

D. Comparison of Application layouts with the State-of-the-Art MPI Libraries

In this section, we evaluate the proposed designs for the performance of various applications layouts using applications level kernels and compare the results with existing state-of-the-art GPU-aware MPI libraries. For our comparisons here, we utilize the MVAPICH2-GDR library [12] (version 2.3.6) and OpenMPI+UCX [1] (version 4.1.3 and ucx version 1.12.1). We would like to note that in the section V-B, we only show the comparison of basic pack schemes. Advanced pack schemes such as pipelined pack have been implemented in OpenMPI+UCX. Therefore we use OpenMPI+UCX numbers as reference for such schemes.

We utilize the following application layouts for our evaluation below:

MILC: MILC studies the integration of quarks and gluons using Quantum Chromodynamics (QCD). The *MILC_su3_zd* kernel in DDTbench models the z-direction of the *su3_rmd* application from the MILC code. It uses nested vector datatype for 4D face exchanges. Figure 11(a) shows that the Proposed-UMR scheme is at least 15% better than MVAPICH2-GDR and is at least 10X better than OpenMPI+UCX. The layout used for the inputs has block lengths varying from 768 bytes to 6144 bytes. The strides for these layouts are several orders of magnitude larger than the block length. The main reason we see such benefits with the proposed scheme is the mkey

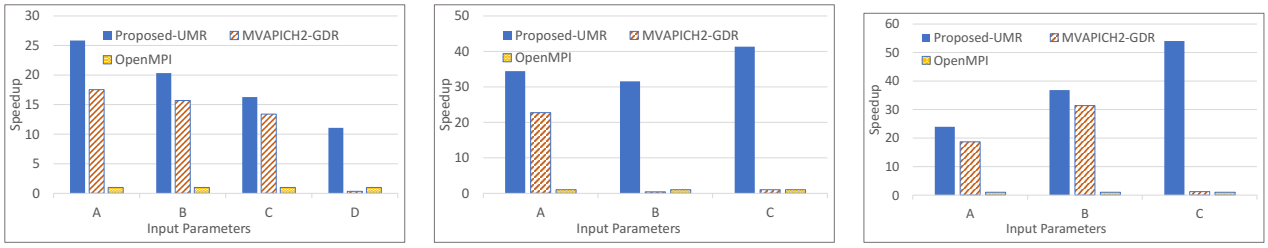


(a) Comparison of proposed schemes with state-of-the-art MPI libraries for 1KB block size.

(b) Comparison of proposed schemes with state-of-the-art MPI libraries for 2KB block size.

(c) Comparison of proposed schemes with state-of-the-art MPI libraries for 4KB block size.

Fig. 10. Performance comparison of schemes for representative layouts with the basic pack schemes



(a) **MILC** on ThetaGPU. Grid dimensions are A = (8,8,16,32), B = (8,16,16,32), C = (16,16,32,32)), D = (16,32,32,32).

(b) **NASMGY** on ThetaGPU. Grid dimensions are A = (512,66,66), B = (1024,66,66), C = (2048,66,120).

(c) **SPECFEM3D_mt** on ThetaGPU. Grid dimensions are A = (1024,2,32), B=(1024,2,64), C=(1024,2,128)

Fig. 11. Normalized performance comparison of Proposed-UMR with state-of-the-art solutions using application kernels for different input sizes. Latencies are normalized with OpenMPI+UCX. Higher is better.

cache that amortizes the overhead caused by mkey mapping. This coupled with the absence of pack-kernel overheads in the proposed scheme make it viable for these layouts.

NAS_MG: It is a fluid dynamics application that does 3d face exchanges in x,y, and z directions with vector and nested vector datatypes. For inputs shown in the graphs, the block lengths go to 6KB and similar to MILC strides several orders of magnitude more than the block lengths. The counts used for these layouts are around 60 elements. The proposed scheme does at least 50% better than MVAPICH2-GDR. The proposed scheme does about 30X better than OpenMPI+UCX. This again demonstrates the efficacy of HCA-assisted designs that avoid the usage of pack-unpack kernels.

SPECFEM3D_GLOBE: Specfem3d_Globe is a spectral-element an application that can simulate global seismic wave propagation through the earth model. We used the *SPECFEM3D_mt* kernel, which uses vector and contiguous data types for data exchange. In Figure 11(c), we compare the performance of the proposed scheme with MVAPICH2-GDR and OpenMPI. The block length used is 4KB and the counts vary from 32 to 128. Our proposed scheme is nearly 20% better than MVAPCIH2-GDR and performs approximately 20X better than OpenMPI+UCX.

VI. Related Work

For optimizing GPU based DDT exchange, the first study provides a significant speedup over CPU-based design for datatype processing was done by Wang et al. [20]. They process non-contiguous datatypes by leveraging a multi-stage pipeline of data transfer and offloading packing/unpacking processing from the host to the device. Jenkins et al. concluded that non-contiguous data transfer could improve performance by kernelizing the packing operations into the GPU [9], [10]. Rong et al. propose a novel packing framework, called HAND, to efficiently pack and unpack non-contiguous data on GPU directly [16]. To obtain a higher overlap between CPU and GPU executions and eliminate unnecessary synchronizations, [3] propose an asynchronous design by taking advantage of several CUDA features. Wu et al. implement a different way to offload the packing and unpacking operations onto the GPU and seamlessly integrate with RDMA networks [22]. However, none of these approaches use HCA assisted scheme to avoid pack-unpack kernels.

To efficiently mitigate performance penalties caused by transferring non-contiguous data, extensive research has been explored with MPI derived datatypes processing. Traff et al. propose ”flattening on the fly” scheme to optimize the parse of MPI DDT layout [19]. Gropp et al. provide a guideline for using various aspects of datatype [7] based on the performance evaluation. Byna et al. propose packing algorithms that take advantage of memory-optimization techniques, which improves the performance of derived datatypes [2]. There are other approaches for MPI datatype communication over the InfiniBand network such as pack/unpack-based, and copy-reduced approaches [21]. To support processing MPI datatype routines efficiently outside of the MPI implementations, Ross et al. propose an open-source library, MPITypes [14]. In [8], the MPI DDT layout extraction and caching are analyzed thoroughly. A new zero-copy scheme for MPI DDT is proposed

by leveraging inter-process load-store operations on CPU and GPU memory within the node. These approaches are aimed at optimizing the DDT processing cost whereas we optimize the exchange of data.

Girolamo et al. [6] implement full non-contiguous memory transfer processing and work with sPIN, a packet streaming processor, to develop scheduling strategies that enhance datatype processing. Our work is not at the HCA level, rather we leverage existing HCA's feature to optimize GPU workloads. Chu et al. [4] propose a zero-copy scheme to exploit load-store semantics over NVLink/PCIe and achieve pack-free mechanism. Moreover, they implement an adaptive scheme on selecting the optimal CPU- or GPU-driven packing/unpacking scheme if NVLink/PCIe is not available between GPUs. However, our work is focused on optimizing the inter-node transfers. Our work could be added to their adaptive scheme.

VII. Conclusion

The deployment of GPUs to accelerate many modern Supercomputers has created a need for optimized communication patterns that adhere to the needs of these applications. In particular, applications that are utilizing GPU-aware MPI may require exchanging data that is non-contiguous in GPU memory. While MPI Derived Datatypes have been used in the past and extensive work has elaborated on the usage of datatypes for non-contiguous data movement, most of this work focuses on optimizing packing and unpacking schemes. In this work, we proposed an efficient mechanism to handle non-contiguous data on GPUs being communicated across the network for inter-node communication. Through these designs, we utilize the features provided by modern HCAs in order to gather then scatter data between non-contiguous GPU memory regions. We provide an extensive evaluation of our proposed schemes against existing approaches. At the benchmark layer, we are able to present approximately 2X improvement with our HCA assisted schemes compared to approaches currently utilized in various state-of-the-art GPU-aware MPI libraries. We also utilize the layouts provided by MILC, NASMG, and Specfem3D to show improvement against various libraries including MVAPICH2-GDR and OpenMPI+UCX.

References

[1] Open MPI: Open Source High Performance Computing. http://www.open-mpi.org.

[2] Byna, Gropp, X.-H. Sun, and Thakur. Improving the performance of MPI derived datatypes by optimizing memory-access cost. In *2003 Proceedings IEEE International Conference on Cluster Computing*, pages 412–419, Dec 2003.

[3] C.-H. Chu, K. Hamidouche, A. Venkatesh, D. S. Banerjee, H. Subramoni, and 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)*, pages 983–992, May 2016.

[4] C.-H. Chu, J. M. Hashmi, K. S. Khorassani, H. Subramoni, and 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)*, pages 267–276, 2019.

[5] C.-H. Chu, K. S. Khorassani, Q. Zhou, H. Subramoni, and D. K. Panda. Dynamic kernel fusion for bulk non-contiguous data transfer on gpu clusters. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 130–141, 2020.

[6] S. D. Girolamo, K. Taranov, A. Kurth, M. Schaffner, T. Schneider, J. Beránek, M. Besta, L. Benini, D. Roweth, and T. Hoefler. Network-Accelerated Non-Contiguous Memory Transfers, 2019.

[7] W. Gropp, T. Hoefler, R. Thakur, and J. L. Träff. Performance Expectations and Guidelines for MPI Derived Datatypes. In Y. Cotronis, A. Danalis, D. S. Nikolopoulos, and J. Dongarra, editors, *Recent Advances in the Message Passing Interface*, pages 150–159, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[8] J. M. Hashmi, C.-H. Chu, S. Chakraborty, M. Bayatpour, H. Subramoni, and D. K. Panda. FALCON-X: Zero-copy MPI Derived Datatype Processing on Modern CPU and GPU Architectures. *Journal of Parallel and Distributed Computing (JPDC)*, 2020.

[9] J. Jenkins, J. Dinan, P. Balaji, T. Peterka, N. F. Samatova, and R. Thakur. Processing MPI Derived Datatypes on Noncontiguous GPU-Resident Data. *IEEE Transactions on Parallel and Distributed Systems*, 25(10):2627–2637, Oct 2014.

[10] J. Jenkins, J. Dinan, P. Balaji, N. F. Samatova, and R. Thakur. Enabling Fast, Noncontiguous GPU Data Movement in Hybrid MPI+GPU Environments. In *2012 IEEE International Conference on Cluster Computing*, pages 468–476, Sept 2012.

[11] W. Lawry, C. Wilson, A. B. Maccabe, and R. Brightwell. COMB: A Portable Benchmark Suite for Assessing MPI Overlap. In *IEEE Cluster*, pages 23–26, 2002.

[12] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. Design and Implementation of MPICH2 over InfiniBand with RDMA Support. In *Proceedings of Int'l Parallel and Distributed Processing Symposium (IPDPS '04)*, April 2004.

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

[14] R. Ross, R. Latham, W. Gropp, E. Lusk, and R. Thakur. Processing MPI Datatypes Outside MPI. In M. Ropo, J. Westerholm, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 42–53, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[15] T. Schneider, R. Gerstenberger, and T. Hoefler. Micro-Applications for Communication Data Access Patterns and MPI Datatypes. In *Recent Advances in the Message Passing Interface - 19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria, September 23-26, 2012. Proceedings*, volume 7490, pages 121–131. Springer, Sep. 2012.

[16] R. Shi, X. Lu, S. Potluri, K. Hamidouche, J. Zhang, and D. Panda. HAND: A Hybrid Approach to Accelerate Non-contiguous Data Movement Using MPI Datatypes on GPU Clusters. In *43rd International Conference on Parallel Processing (ICPP)*, pages 221–230, Sept 2014.

[17] T. . Supercomputers. http://www.top500.org/statistics/list/.

[18] M. Technologies. Mellanox Adapters Programmer's Reference Manual (PRM). https://network.nvidia.com/files/doc-2020/ethernet-adapters-programming-manual.pdf.

[19] J. L. Träff, R. Hempel, H. Ritzdorf, and F. Zimmermann. Flattening on the Fly: efficient handling of MPI derived datatypes. In J. Dongarra, E. Luque, and T. Margalef, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 109–116, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[20] H. Wang, S. Potluri, M. Luo, A. Singh, X. Ouyang, S. Sur, and 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*, pages 308–316, Sept 2011.

[21] J. Wu, P. Wyckoff, and D. Panda. High performance implementation of MPI derived datatype communication over InfiniBand. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pages 14–, April 2004.

[22] W. Wu, G. Bosilca, R. vandeVaart, S. Jeaugey, and 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, pages 231–242, New York, NY, USA, 2016. ACM.