Received: 16 March 2017 | Revised: 2 October 2017 | Accepted: 22 April 2018
DOI: 10.1002/cpe.4667

WILEY

SPECIAL ISSUE PAPER

Design considerations for GPU-aware collective communications in MPI

Iman Faraji | **Ahmad Afsahi**

Department of Electrical and Computer Engineering, Queen's University, Kingston, ON K7L 3N6, Canada

Correspondence
Iman Faraji, Department of Electrical and Computer Engineering, Queen's University, Kingston, ON K7L 3N6, Canada.
Email: i.faraji@queensu.ca

Funding information
Natural Sciences and Engineering Research Council of Canada, Grant/Award Number: RGPIN-2016-05389; Canada Foundation for Innovation, Grant/Award Number: 7154

Summary

GPU accelerators have established themselves in the state-of-the-art clusters by offering high performance and energy efficiency. In such systems, efficient inter-process GPU communication is of paramount importance to application performance. This paper investigates various algorithms in conjunction with the latest GPU features to improve GPU collective operations. First, we propose a GPU Shared Buffer-aware (GSB) algorithm and a Binomial Tree Based (BTB) algorithm for GPU collectives on single-GPU nodes. We then propose a hierarchical framework for clusters with multi-GPU nodes. By studying various combinations of algorithms, we highlight the importance of choosing the right algorithm within each level. The evaluation of our framework on MPI_Allreduce shows promising performance results for large message sizes.

To address the shortcoming for small and medium messages, we present the benefit of using the Hyper-Q feature and the MPS service in jointly using CUDA IPC and host-staged copy types to perform multiple inter-process communications. However, we argue that efficient designs are still required to further harness this potential. Accordingly, we propose a static and a dynamic algorithm for MPI_Allgather and MPI_Allreduce and present their effectiveness on various message sizes. Our profiling results indicate that the achieved performance is mainly rooted in overlapping different copy types.

KEYWORDS

collectives, GPU, hierarchical framework, inter-process communications, MPI, MPS

1 | INTRODUCTION

Accelerators and co-processors have reshaped the high-performance computing (HPC) landscape in recent years. Among them, GPU accelerators have gained popularity within the HPC community by offering high performance and energy efficiency. In fact, GPUs have been deployed in many supercomputers according to the Top500 list.[1] In these systems, to achieve higher computational power, the nodes consist of multi-core processors and multiple GPU devices and are interconnected by a fast network.

The Message Passing Interface (MPI[2]) is the de facto standard for parallel programming in HPC systems. HPC applications typically consist of computation phases followed by communication phases. In GPU clusters, MPI application processes running on the host nodes accelerate the computation by launching a kernel to offload the compute-intensive portion of the application onto the GPUs. They would then communicate or exchange the outcome of their GPU computation with the other GPUs/MPI processes residing on the same or different nodes. Consequently, efficient use of GPUs in GPU clusters demands for both efficient GPU computations and GPU inter-process communications. Both intranode and internode GPU-to-GPU communications play a crucial role in the performance of MPI applications,[3] as the overhead imposed by the inter-process GPU communications in applications with high data dependency may wipe out the potential benefits of GPU offloading. To achieve efficient inter-process GPU communication, MPI libraries should be tuned and become GPU-aware to be able to efficiently communicate the data residing on the GPU memory. In this regard, researchers have started looking into incorporating GPU-awareness into the MPI library, targeting both point-to-point and collective communications.[4-11]

In this paper, we take on the challenge to incorporate GPU-awareness into the MPI collective communication operations and propose a number of GPU-aware algorithms for efficient collective operations. We first focus our study on collective operations targeting systems with single-GPU nodes. For such systems, we study how the choice of using different algorithms could affect the performance of our collective designs. For that, we propose and evaluate two algorithms in our designs: (1) a GPU Shared Buffer-aware (*GSB*) algorithm and (2) a Binomial Tree Based (*BTB*) algorithm. In *GSB*, we utilize the fan-in/fan-out algorithm to minimize the synchronization among different inter-process communications. In the *BTB* design, we use the binomial tree algorithm to minimize the number of communication steps involved in the collective operation. Both designs utilize GPU shared buffers in conjunction with CUDA interprocess-communication (IPC) to support GPU communication between MPI processes. We also exploit in-GPU kernel functions to perform the computational component of the collective operation on the pertinent data that reside on the GPU global memory.

In MPI, multiple processes typically run on a single node. Nodes in HPC machines are usually equipped with multiple GPUs in order to achieve higher computational power and memory capacity. Consequently, MPI processes may be assigned to different GPUs or share a single GPU. In such a setting, data communication between the GPU buffers of different pair of processes would have to traverse different communication channels with different characteristics. Taking this into account, we propose a hierarchical framework for GPU collective communication operations targeting multi-GPU nodes. Using our framework, we break up the collective operation into three distinct phases: (1) Intranode intra-GPU phase, (2) Intranode inter-GPU phase, and (3) Internode inter-GPU phase. The inter-process communications within each phase is restricted to use the communication channel that is associated with that phase. We evaluate the efficiency of our hierarchical framework by utilizing the proposed *GSB* and *BTB* collective algorithms within different phases.

The evaluation of our proposed designs on MPI_Allreduce shows promising performance improvement for large message sizes. The poor performance for small and medium message sizes lies in the high overhead associated with the use of CUDA IPC in our collective designs. CUDA IPC and the host-staged copy are two different data copy types that may be used to perform GPU communication between processes residing on a single node. Previous research studies, however, have used only one of these copy types to perform the GPU inter-process communication.[6,8,10,11] Therefore, utilizing the alternative copy type, the host-staged copy, for small and medium-size messages could be a viable solution. This paper investigates how we can potentially use different copy types in conjunction with each other to improve the GPU collective operation performance on a wide range of message sizes. Accordingly, based on the analysis of CUDA IPC and host-staged copy types, we propose a *Static* algorithm and a *Dynamic* algorithm for GPU collective operations that jointly use both copy types. Our designs make informed decisions on how to efficiently use different copy types for performance. The *Static* algorithm makes this decision based on a tuning table that is provided to it prior to the runtime. The *Dynamic* algorithm, on the other hand, makes this decision dynamically at runtime. Our experimental results indicate that our designs in most cases outperform other collective designs.

The work in this paper extends our prior study[12,13] in different ways. While our collective designs in our other work[12] target a single node with a single GPU, in this paper, we extend our work and propose a three-level hierarchical framework for GPU collectives for clusters with multi-GPU nodes. The intention of this framework is to highlight the importance of selecting the right algorithm at each hierarchy level in performing the GPU collective operations. We evaluate different combinations of our algorithms in the proposed framework and discuss our findings. In addition, this paper extends the proposed algorithms in our other work[13] from a single GPU to across the clusters and provides an extended evaluation of using different copy types for collective operations against a wider set of alternative designs. Our experimental results highlight the importance of efficiently using the right copy type in GPU collective operations; this observation is further investigated and discussed in this paper by providing some profiling results.

The rest of this paper is organized as follows. Section 2 presents the background material. We discuss the related work in Section 3. We then present and evaluate our GPU-aware collective designs and the hierarchical framework in Section 4. In Section 5, we present our *Static* and *Dynamic* algorithms and evaluate their performance. Finally, in Section 6, we make concluding remarks and outline our future work.

2 | BACKGROUND

2.1 | Intranode inter-process GPU communication

MPI processes have different address spaces. Therefore, processes require to use an inter-process communication mechanism to exchange data with each other. Similarly, for an inter-process GPU communication, the GPU IPC feature is used to transfer data into or from the GPU address space of MPI processes. The GPU IPC can be performed using different data copy types. Such data copy types may use different communication channels and GPU features to facilitate GPU inter-process communications. The host-staged and the CUDA IPC are two data copy types that can be used to perform GPU IPC between processes residing on a single node. The host-staged data copy type is performed by first staging the data from the GPU buffer into the host memory buffer of the source process. Next, an inter-process copy is performed between the host buffers of the source and target processes (if the host memory is not shared). Finally, the staged data are copied from the host buffer to the GPU buffer of the target process.

With the NVIDIA CUDA IPC, data can be directly copied (without host intervention) from the GPU address space of one process to the GPU address space of another process within the same root complex. The CUDA IPC copy requires a process to expose a portion of its address space to the remote processes. In this regard, a memory handle of the shared address is created and passed to the remote processes. The remote processes can then access and modify the shared remote address space; however, synchronization between the involved processes is required to guarantee

the completion of the copy. This synchronization is performed using a shared CUDA IPC event, by one process recording it after initiating its IPC copy and the other process querying its completion. Depending on the inter-process communication characteristics such as the message size, one copy type may be favored over the other. In this paper, we seek to investigate the performance of these copy types and decide how to use them efficiently in collective operations.

2.2 | Hyper-Q and Multi-Process Service

Hyper-Q[14] is an NVIDIA feature that provides potential concurrency among CUDA tasks from a single process. However, Hyper-Q by itself cannot provide concurrency among CUDA requests from multiple processes to the GPU compute and memory engine; thus, these tasks would have to serialize. In order to provide such concurrency across multiple processes, NVIDIA has introduced the Multi-Process Service (MPS[15]) for GPUs with compute capability of 3.5 and above. The MPS service acts as a funnel to collect CUDA tasks from multiple intranode processes and issues them to the GPU as if coming from a single process so that the Hyper-Q feature can take effect. Without this service, each of the MPI processes has to allocate storage and scheduling resources on the GPU, and only work from a single context can be launched on the GPU engines at a time. In contrast, with the MPS service enabled, there is only a single context, known as MPS context, present on the GPU. This allows all processes to share the GPU storage and scheduling resources, eliminating the overhead of the context switching. In this paper, we use this feature to enhance the overlap between different GPU data copy types in collective communications.

2.3 | GPU-aware MPI

The prevailing use of GPU accelerators in HPC clusters has lead the application developers to adapt many of the existing MPI applications to be able to use the GPU resources in such heterogeneous systems. In such applications, the compute-intensive portion of the application is offloaded and accelerated on the GPU, whereas MPI is used to communicate the data that reside on the GPU buffers. To perform such communications efficiently, GPU-awareness has been added to some MPI libraries to remove the burden of learning a new programming language from the programmer to use MPI efficiently in conjunction with the GPU. The GPU-aware MPI libraries would help the programmer to develop a more concise, readable, and even efficient application to run on the GPU clusters. MVAPICH2[16] and Open MPI[17] are two well-known open-source MPI libraries supporting GPU-awareness.

The GPU collective communication in GPU-aware MPI libraries may follow a general approach, which involves staging the GPU data into the host buffer and leveraging the CPU-based MPI routines. It may also involve further tunings by pipelining the transfers and using specifically designed algorithms for some MPI routines. The first step in the general approach involves copying the pertinent data from the GPU global memory into the host memory buffers; next, the MPI operation is performed on data that reside on the host buffers, and finally, the result is written back to the GPU memory. Depending on the message size, some MPI implementations such as MVAPICH2 may leverage a host-based pipelining design to hide the CUDA memory copy latency or use a more advanced design, such as the Fine Grained Pipeline (FGP) algorithm[18] that is proposed for MPI_Allgather. The FGP algorithm exploits simultaneous asynchronous network transfers and CUDA copies in a store and forward fashion.

MVAPICH2-GDR 2.0 is a closed-source MPI implementation that leverages the GPUDirect RDMA technology to achieve significant improvement for small message GPU-to-GPU communication. MVAPICH2-GDR takes advantage of the loopback and gdrcopy features in the intranode point-to-point and collective operations for small messages. The loop-back design replaces the CUDA memory copy that has an initial calling overhead that is not negligible for short messages. The GDR copy has a non-blocking nature, allowing the transfer to progress in parallel and thus incurring a lower latency compared with the CUDA memory copy used in MVAPICH2.

Current GPU-aware collective operations neither utilize efficient GPU-aware algorithms nor fully exploit modern GPU features. Moreover, while different data copy types have been proposed for GPU inter-process communications, efficiently leveraging them in collective operations has not been investigated. The existing GPU-aware collective operations leverage flat designs, which is inefficient given the hierarchical structure of multi-GPU nodes and GPU clusters. This paper proposes efficient GPU-aware algorithms and hierarchical solutions for GPU collectives that also utilize modern GPU features. It also proposes algorithms that wisely use different copy types in a concurrent fashion.

3 | RELATED WORK

In traditional clusters, using an intranode collective stage in a hierarchical design for a collective operation is a well studied problem.[19-22] In the work of Tipparaju et al,[19] the authors leveraged remote memory operations across the cluster and shared memory within the cluster to implement collective operations. Shared memory is also used in the work of Graham and Shipman[20,21] to perform data transfer; while the former paper considered various intranode collective operations, the latter proposed a multi-threading based MPI_Allreduce. Mamidala et al[22] proposed SMP-based collective designs that is aware of the multi-core aspect of the clusters. This work studies the performance impact of different hierarchical architectures on the collective communication performance. While these works study the impact of hierarchical algorithms on homogeneous CPU clusters, in this paper, we propose a hierarchical framework that targets GPU clusters with multi-GPU nodes. Our hierarchical framework for GPU collectives is composed of three phases: (1) intranode intra-GPU, (2) intranode inter-GPU, and (3) internode inter-GPU.

In GPU clusters, various non-hierarchical GPU-aware collective operations have been proposed.[6,11] Singh et al[6] optimized MPI_Alltoall by leveraging a pipelining mechanism to overlap device-to-host and host-to-device CUDA memory copies with the network communications. In their work, host-staged copy type was used for inter-process communications. Chu et al[11] investigated various algorithms for GPU-aware MPI_Allreduce across the node. However, none of these work considered hierarchical collective designs. The CUDA IPC copy type was studied in the work of Ji et al[4,5] for one-sided and point-to-point communications as well. In this paper, unlike previous studies that either use CUDA IPC or host-staged copy, we leverage a combination of both copy types for intranode inter-process communications. We propose algorithms for GPU collective operations that are capable of deciding the number and type of the inter-process copies. Depending on the algorithm, we opt to make this decision based on the information that is gathered either during or prior to the runtime. The NVIDIA Collective Communications Library (NCCL[23]) implements collective communication primitives similar to MPI routines and is optimized to achieve high bandwidth. This library requires MPI applications to be modified in order to integrate NCCL primitives, such as NCCL communicator initializer. On the other hand, our proposed designs in this paper can be directly applied into the MPI collective routines, and no application modification is required. The effect of the MPS service is evaluated in the work of Wende et al[24] on the offloaded computational kernels from multiple processes to the GPU. The MPS service in general targets to reduce the overhead of launching multiple tasks from different processes on intranode GPUs. LeBeane et al[25] introduced XTQ, an RDMA-based active messaging mechanism for accelerators, to issue light-weight active messages across the network. The authors showed 10%-15% improvement for small and medium sized messages on MPI Accumulate, Reduce, and Allreduce operations by using direct NIC-to-accelerator communication. The implementation of the XTQ reduction operations is built on the LibNBC library,[26] which is designed to support non-blocking collectives on generic architectures. In this paper, we evaluate the effect of the MPS service and the Hyper-Q feature on GPU inter-process communications and investigate its impact on our proposed designs for GPU collective operations.

4 | MPI COLLECTIVES WITH EFFICIENT GPU-AWARE HIERARCHICAL ALGORITHMS

In this section, we first propose two different algorithms to implement collective communication operations on a single-GPU node. Next, we propose a hierarchical framework for collective operations that target GPU clusters with multi-GPU nodes. Our hierarchical framework provides the capability to select different algorithms for different hierarchy levels of the GPU cluster. As a case study, we will focus on MPI_Allreduce; however, our designs can be adapted to other collective operations with minimal changes.

4.1 | Design considerations for single-GPU nodes

For collective operations targeting a single-GPU node, we propose to utilize a GPU shared buffer region to hold the collective pertinent data. The GPU shared buffer in our designs is a pre-allocated area in the address space of the GPU global memory. This address space is accessible by all intranode processes and can be used as a shared medium for inter-process communications and storing the pertinent data for efficient implementation of collectives. Our designs also exploit CUDA kernels to speed up collective computations on the GPU. Taking these into account, we propose two designs for GPU collective operations that leverage different algorithms: (1) a GPU Shared-Buffer (*GSB*) aware approach and (2) a Binomial Tree Based (*BTB*) approach.

4.1.1 | GPU shared buffer-aware design

In the GPU Shared Buffer-Aware (*GSB*) design for MPI_Allreduce, we use an aggregated GPU shared buffer area to gather the pertinent collective data, manipulate the data, and make the collective result available to the designated processes. The GPU shared buffer in our design is an aggregated space that is allocated in its entirety in the address space of a predefined process. Below, we discuss the general stages involved in designing MPI_Allreduce using the *GSB* approach.

GSB Allreduce

The *GSB* Allreduce is implemented by following two general stages, Stage1: *GSB* Reduce and Stage2: *GSB* Broadcast. In the following, we first introduce the *GSB* Reduce and Broadcast components of the *GSB* Allreduce and then further discuss the implementation details of the *GSB* Allreduce design.

Stage1: GSB Reduce

The *GSB* Reduce is used to implement the reduce component of the *GSB* Allreduce. The *GSB* Reduce uses the fan-in algorithm to gather the collective pertinent data into the GPU shared buffer. The gathered data is then reduced inside the GPU shared buffer using a GPU reduction kernel. The reduced data is then read by the root process. It should be mentioned that the *GSB* Reduce can be used to implement MPI_Reduce.

Stage2: GSB Broadcast

The *GSB* Broadcast is used to implement the broadcast component of the *GSB* Allreduce. In the *GSB* Broadcast stage, all processes using the fan-out algorithm read the collective data from the GPU shared buffer. With the root process first copying the collective data into the GPU shared buffer, it is evident that the *GSB* Broadcast can also be used to implement MPI_Broadcast operation.

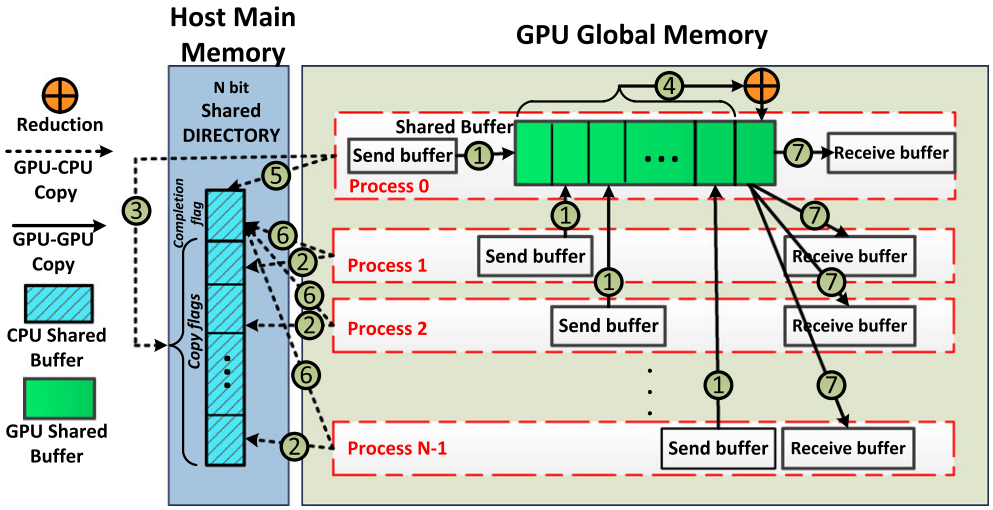


FIGURE 1 Steps of the GPU shared-buffer aware approach for MPI_Allreduce

In Figure 1, we show the different components of the *GSB* design in implementing the MPI_Allreduce operation. All intranode processes copy their pertinent data from their GPU send buffers to the GPU shared buffer. Once all data are available in the GPU shared buffer, the reduction operation takes place on the aggregated data, and the result is stored back into the GPU shared buffer. Upon availability of the result, all processes copy the results into their GPU receive buffers.

According to the Figure 1, the GPU shared buffer in its entirety is allocated in the address space of a predefined process (without loss of generality, we assume the process with `rank 0` as the predefined process). Processes exploit CUDA IPC to communicate through the GPU shared buffer region. Processes on the node also have access to a shared directory that keeps track of the IPC copies into the GPU shared buffer and report their completion. This buffer can be allocated either on the GPU global memory or on the host memory. We have evaluated both design alternatives and decided to keep the directory on the host memory. We provide some justifications for this decision later in Section 4.1.3.

The size of the shared directory is `intra_comm_size` bits, in which `intra_comm_size` represents the number of processes on the node. Each of the first `intra_comm_size - 1` bits (`copy flags` - see Figure 1) is associated with one process rank and is set once this process initiates its IPC copy into the GPU shared buffer. The last bit in the directory (`completion flag`) indicates the completion of the collective operation and the availability of the results in the GPU shared buffer. This bit is set by the pre-defined process with `rank 0`.

The GPU shared buffer area should be sufficiently large to hold the gathered dataset from all intranode processes (256 MB is used in our experiments). This is directly related to the number of processes/GPUs per node, as well as the size of the dataset that are typically in-use in applications leveraging MPI_Allreduce. As such, the size of the allocated GPU shared buffer is much less than the amount of global memory available in modern GPUs. Therefore, we believe this is not a scalability concern in our design.

Figure 1 shows the general steps of the GPU shared-buffer aware MPI_Allreduce design as follows:

`Step 1.` All processes copy their share of data from their send buffers into their associated addresses in the GPU shared buffer area.
`Step 2.` All processes (except `process 0`) set their associated `copy flags` in the shared directory after initiating their IPC copies.
`Step 3.` `Process 0` waits on all `copy flags` to be set (this step can overlap with `Step 2`).
`Step 4.` Once all `copy flags` are set, all pertinent data are available in the GPU shared buffer. `Process 0` then performs an in-GPU element-wise reduction on the aggregated data and stores the result in a predefined location inside the GPU shared buffer.
`Step 5.` Once the collective result becomes available in the GPU shared buffer, `process 0` toggles the `completion flag` to indicate the completion of the collective operation.
`Step 6.` All processes (except `process 0`) query the `completion flag` (this step can overlap with `Steps 3, 4,` and `5`).
`Step 7.` Once the `completion flag` is toggled, all processes copy their share of collective result into their respective receive buffers.

Note that in `Step 5`, the `completion flag` has to be toggled in each instance of MPI_Allreduce call; otherwise, successive MPI_Allreduce calls may end up reading stale data.

Implementation Details: The GPU shared-buffer aware approach leverages pre-allocated CPU and GPU shared buffers. The CPU shared memory region is allocated during MPI initialization stage, `MPI_Init()`, and is attached to the address space of the intranode processes. The memory handle of the GPU shared buffer is also created, broadcast, and mapped to the address space of other processes during this stage. This is performed only once to mitigate the costly operation of exposing the GPU shared buffer address space.

The IPC copies in CUDA have an asynchronous behavior, even if synchronous CUDA memory copy operations are issued on the shared region. Therefore, the `copy flags` in the shared directory only indicate the initiation of the IPC copies but not their completion. To guarantee the completion of the IPC copies, we leverage the IPC events. The IPC event can be shared and used among processes residing on the same node. To share an

IPC event, the handle of the allocated event is created and passed on by the predefined process to the other processes. To guarantee the completion of the IPC copies, an IPC event using `cudaEventRecord()` command is immediately recorded after each IPC copy (`Step 1` in Figure 1). Recording an event is then followed by setting the associated `copy flag` in the shared directory in `Step 2`. Once each `copy flags` is set, `process 0` issues `cudaStreamWaitEvent()` on the associated event until the inter-process copy completes. The `process 0` will wait on all IPC events in a first-come first-served fashion. This way, the completion of all IPC copies can be guaranteed. Finally, all allocated shared buffers and events are freed and destroyed within the `MPI_Finalize()` call.

4.1.2 | GPU-aware binomial tree based design

The *BTB* approach uses the binomial algorithm in conjunction with GPU shared buffer to perform collective operations. The GPU shared buffer in the *BTB* approach (unlike the black*GSB* approach) is distributed among processes involved in the collective operation. Similar to the black*GSB* approach, in *BTB*, we use both the GPU and host shared buffers. The IPC events are also similarly used in conjunction with a directory to indicate the completion of the IPC copies.

BTB Allreduce

The *BTB* Allreduce follows two stages to implement MPI_Allreduce using the *BTB* design, Stage1: *BTB* Reduce and Stage2: *BTB* Broadcast. In the following, we first introduce the *BTB* Reduce and Broadcast components of the *BTB* Allreduce and then further discuss the implementation details of the *BTB* Allreduce design.

Stage 1: BTB Reduce

The *BTB* Reduce implements the reduction operation by following the binomial algorithm. In each level of the algorithm, processes copy their collective pertinent data into the GPU shared buffer of their peer process. The received and the peer process data are then reduced inside the GPU shared buffer using a GPU reduction kernel and the algorithm proceeds to the next level. In the *BTB* Reduce, the binomial tree is traversed from the leaf to the root process, and the distance between communicating processes is doubled after each step. It takes $\lceil log(N) \rceil$ steps (N is the number of intranode processes) to reduce the data. *BTB* Reduce can be used to implement the MPI_Reduce operation.

Stage 2: BTB Broadcast

To implement the broadcast component of the *BTB* Allreduce, all processes, using the binomial tree algorithm, read their collective data from the GPU shared buffer. The binomial tree in *BTB* broadcast is traversed from the root to the leaf, and the distance between communicating processes is halved after each step. It takes $\lceil log(N) \rceil$ steps to broadcast the data. With the root process first copying the collective data into the GPU shared buffer, the *BTB* Broadcast can also be used to implement the MPI_Broadcast operation.

Figure 2 shows the detailed steps involved in the *BTB* Reduce stage followed by the *BTB* Broadcast stage of the *BTB* Allreduce. Considering that the steps involved in different levels of the collective operation are similar, we only discuss the first and last levels of the reduce part of our design.

BTB Reduce Stage - Level 1:

`Step 1.` First, each odd-ranked process uses IPC to copy the contents of its send buffer to the GPU shared buffer of its adjacent, even-ranked peer process.
`Step 2.` Odd-ranked processes set their associated flag in the directory (allocated on the CPU shared buffer) after initiating their IPC copies.
`Step 3.` Even-ranked processes query the flag in the directory to check the initiation of the IPC copy from their peer process.
`Step 4.` Once the flag is set, each even-ranked process performs an element-wise reduction between the data in its GPU send buffer and the pertinent data in its GPU shared buffer. The reduced result is stored back into the GPU shared buffer.

This algorithm then proceeds to the next levels of the binomial tree. In each level, the distance between the peer processes is doubled compared to the previous level. This algorithm terminates when the last binomial tree level (ie, *Level log(N)*) is processed. The steps of the final level is as follows.

BTB Reduce Stage - Level log(N):

`Step 1.` `Process N/2` IPC copies the content of its GPU shared buffer into the GPU shared buffer of its peer process (ie, `Process 0`).
`Step 2.` `Process N/2` sets its corresponding flag in the directory after initiating its IPC copy.
`Step 3.` `Process 0` queries the directory flag.
`Step 4.` Once the flag is set, `process 0` performs the final reduction on the pertinent data into the GPU shared buffer.

Broadcast Stage: Next, `process 0` broadcasts the reduced data from its GPU shared buffer to the rest of the processes. In the `broadcast stage`, the IPC copy and setting/querying the directory are performed similar to the `reduce stage`. However, unlike the `reduce stage`, the distance between the peer processes is halved in each step.

Implementation Details: The *BTB* design, similar to the design, utilizes both GPU and host shared buffers. The GPU shared buffer is allocated by only half of the participating processes. Thus, the size of the shared buffer in the *BTB* approach is half the size of the shared buffer in the approach. The directory is allocated on the host memory, and its size is also half the size of the directory in the approach. The memory and event handles are created and passed along during the MPI initialization time. In the *BTB* design, we guarantee the order of the communications and computations by enforcing processes on the same tree level to only communicate with each other. To this aim, we store the tree level of the sending process in its associated entry in the shared directory. This way, the sending and receiving processes can check and match their tree levels before initiating the

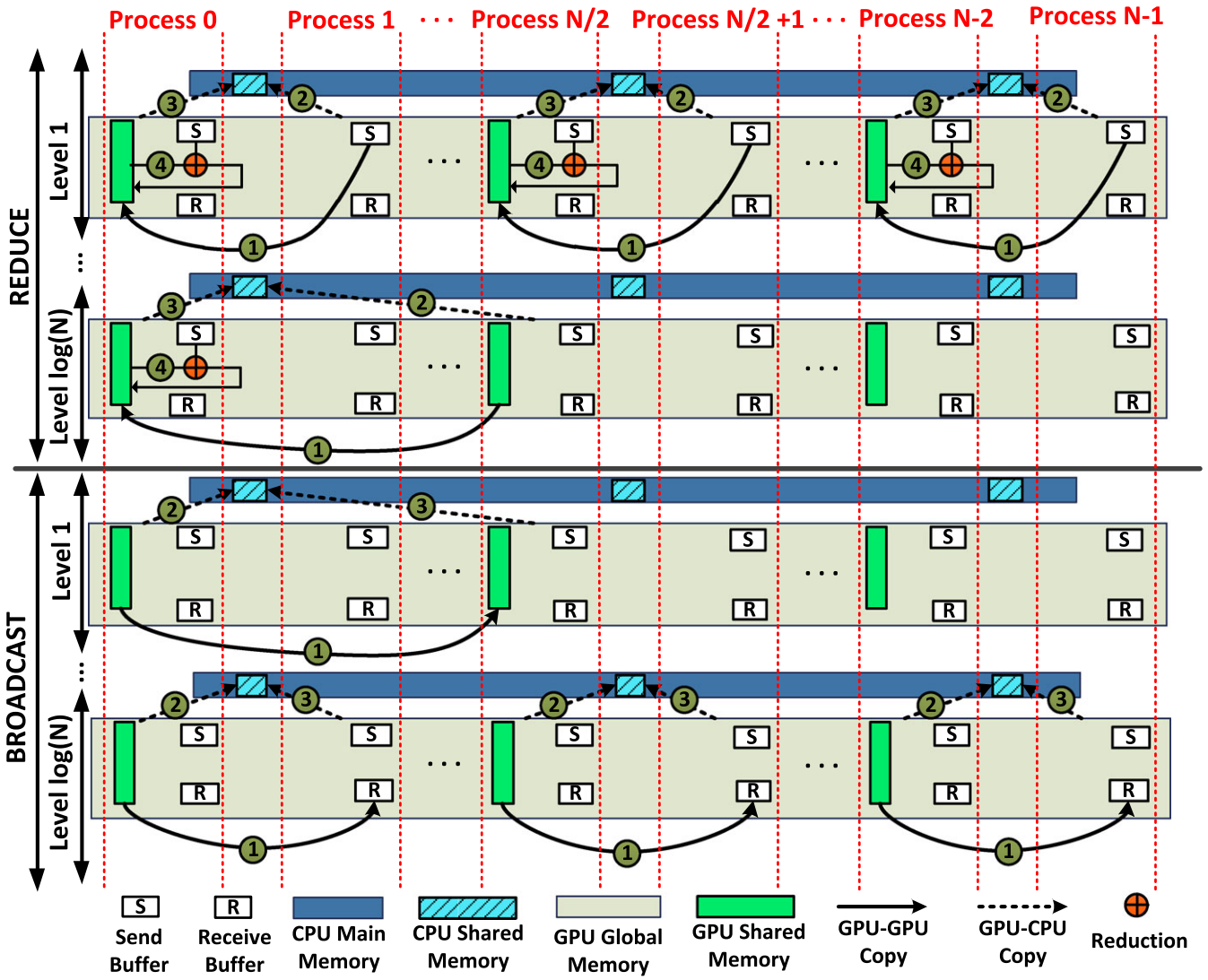


FIGURE 2 Steps of the GPU-aware MPI_Allreduce using the BTB design

IPC copy. Matching the tree level indicates that all of the copies in previous tree levels have been already completed. Consequently, the ordered communication/computation can be guaranteed, preventing any potential race condition.

4.1.3 | Other design considerations

Both *GSB* and *BTB* designs allocate the shared buffer on the GPU global memory while the directory is kept on the host main memory. In the first glance, having the directory on the GPU memory with a kernel function querying its entries seems to be justified; however, this can potentially lead to spin-waiting on the directory forever, as the process querying the directory will take over the GPU resources and would prohibit other processes to access them. We tried to address this issue by forcing the querying process to release the GPU in time-steps. However, the performance results were not promising and selecting the appropriate value for the time-step was dependent on many factors such as message size and process count.

We also tried to query the directory using CUDA asynchronous copy operations. Though this approach was feasible, it had high detrimental effect on the performance. The performance slowdown is basically due to the high number of asynchronous copy calls issued by the querying processes. These calls have to be synchronized at the beginning of each MPI_Allreduce invocation. Synchronization calls are costly, as they require waiting on all previously issued copies on the directory to complete. Avoiding synchronization calls can result in accessing stale data on the directory, which were stored in the previous invocation of MPI_Allreduce. This can ultimately result in inaccurate directory checking. Taking everything into consideration, the best way to exploit this feature in our designs is to allocate the directory on the host main memory, while keeping the shared buffer on the GPU.

4.2 | Design considerations for GPU clusters with multi-GPU nodes

4.2.1 | Hierarchical collective design

In HPC clusters, multi-GPU nodes are commonly used to provide higher aggregated GPU computational power and memory capacity. In such nodes, the number of CPU cores is usually larger than the number of GPU devices. As a result, each GPU can be potentially oversubscribed by

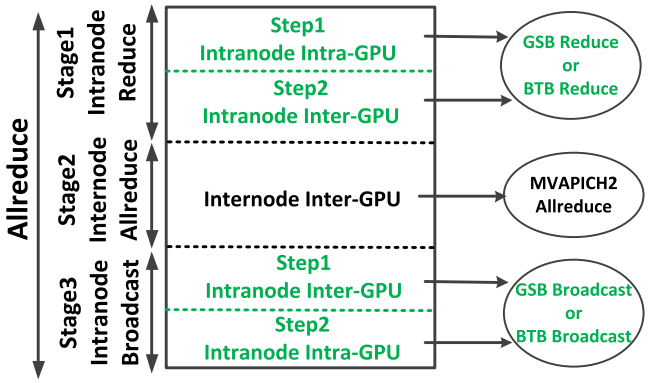


FIGURE 3 Steps of the hierarchical GPU-Aware MPI_Allreduce on GPU clusters with multi-GPU nodes

multiple processes. In such GPU clusters, inter-process communications may go through different communication channels with different bandwidth capacities. Taking this hierarchical architecture into account, we propose a three-level hierarchical framework to extend our black*GSB* and *BTB* algorithmic designs across the clusters with multi-GPU nodes. Our proposed framework facilitates the important aspect of the selection of different algorithms in each hierarchy level. In the following, as a test case scenario, we discuss how our proposed framework can be applied to MPI_Allreduce.

MPI_Allreduce

To apply our proposed hierarchical framework to MPI_Allreduce, we perform this operation in three stages: Stage 1, Intranode Reduce; Stage 2, Internode AllReduce; and Stage3, Intranode Broadcast. Figure 3 illustrates these general stages.

Stage1: Intranode Reduce

Step1: Intranode Intra-GPU Reduce - Intranode intra-GPU processes reduce their pertinent data and store it in the GPU shared buffer of their pre-defined *GPU leader process*; this step is performed using the *GSB* Reduce (or *BTB* Reduce) algorithm.

Step2: Intranode Inter-GPU Reduce - On each node, using the *GSB* Reduce (or *BTB* Reduce) algorithm, the reduce operation is performed among the GPU leader processes and the result is stored in a pre-defined *node leader process*.

It should be mentioned that by adding an Internode Inter-GPU Reduce stage after the above two steps, MPI_Reduce can be can be implemented across the multi-GPU cluster.

Stage 2: Internode AllReduce

The node leader processes engage in an Internode Inter-GPU Allreduce stage. We use the existing internode MVAPICH2 algorithm to perform the MPI_Allreduce operation in this step. At this point, the final reduced result is available at the node leader processes.

Stage 3:Intranode Broadcast

Step1: Intranode Inter - GPU Broadcast - The node leader process in each node uses the *GSB* Broadcast (or *BTB* Broadcast) algorithm to broadcast the reduced data to the GPU leader processes.

Step2: Intranode Intra-GPU Broadcast - Each GPU leader process uses the *GSB* Broadcast (or *BTB* Broadcast) algorithm to broadcast the reduced data to its intranode intra-GPU processes.

It should be mentioned that by adding an Internode Inter-GPU Broadcast stage before the above two steps, MPI_Bcast can be implemented across the multi-GPU cluster.

As shown in Figure 3, unlike the internode phase, the intranode phase is composed of two steps (ie, Intranode Intra-GPU and Intranode Inter-GPU). As discussed, the rationale behind this is that the intranode intra-GPU communication performance is at least an order of magnitude higher than the intranode inter-GPU. Moreover, while intranode intra-GPU communications share the same traversal path, the intranode inter-GPU communications may traverse different paths. Therefore, by separating the two phases, the number of intranode inter-GPU communications can be substantially reduced.

Considering the availability of the proposed *GSB* and *BTB* algorithms in our framework for the intranode stages, four design alternatives exist on a multi-GPU node: (1) GSB-GSB, (2) GSB-BTB, (3) BTB-GSB, and (4) BTB-BTB. In each scenario, the first term determines the algorithm for the Intranode Intra-GPU Step and the second term determines the algorithm for the Intranode Inter-GPU Step. As an example, the GSB-BTB design selects the *GSB* and the *BTB* algorithms for the Intra-node Intra-GPU and the Intranode Inter-GPU Steps, respectively. More specifically, GSB-BTB means that *GSB* Reduce is followed by *BTB* Reduce for the Intranode Reduce stage and *BTB* Broadcast is followed by *GSB* Broadcast for the Intranode Broadcast stage.

Figures 4 and 5 illustrate the general steps involved in the Reduce Stage of the MPI_Allreduce, as shown in Figure 3, using the GSB-GSB and the GSB-BTB designs, respectively. According to these figures, both designs perform a *GSB* Reduce in their intranode intra-GPU level. For the intranode inter-GPU level, the *GSB* Reduce algorithm is used in the GSB-GSB design (Figure 4), whereas the *BTB* Reduce is used in the GSB-BTB design (Figure 5).

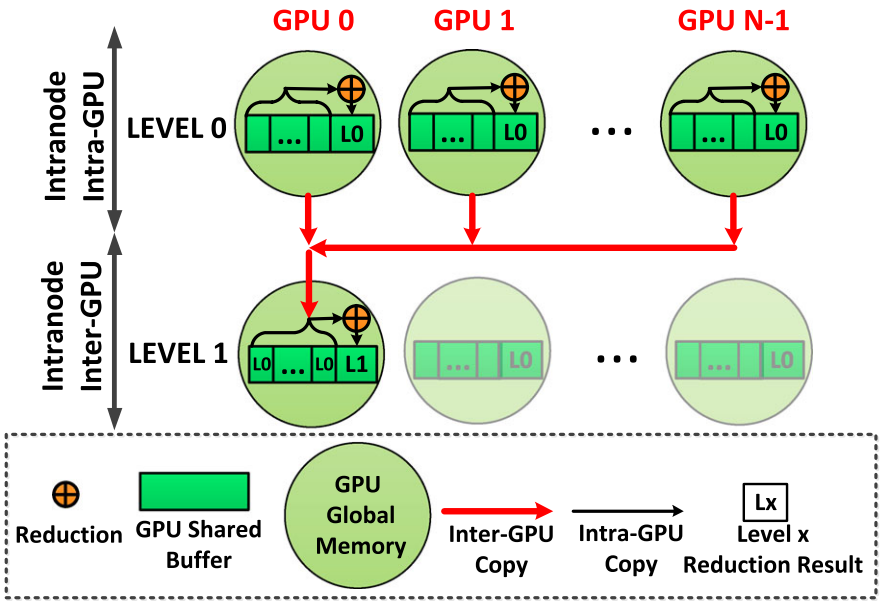


FIGURE 4 Hierarchical MPI_Allreduce utilizing Intranode Intra-GPU *GSB* Reduce and Intranode Inter-GPU *GSB* Reduce algorithms - Reduce stage

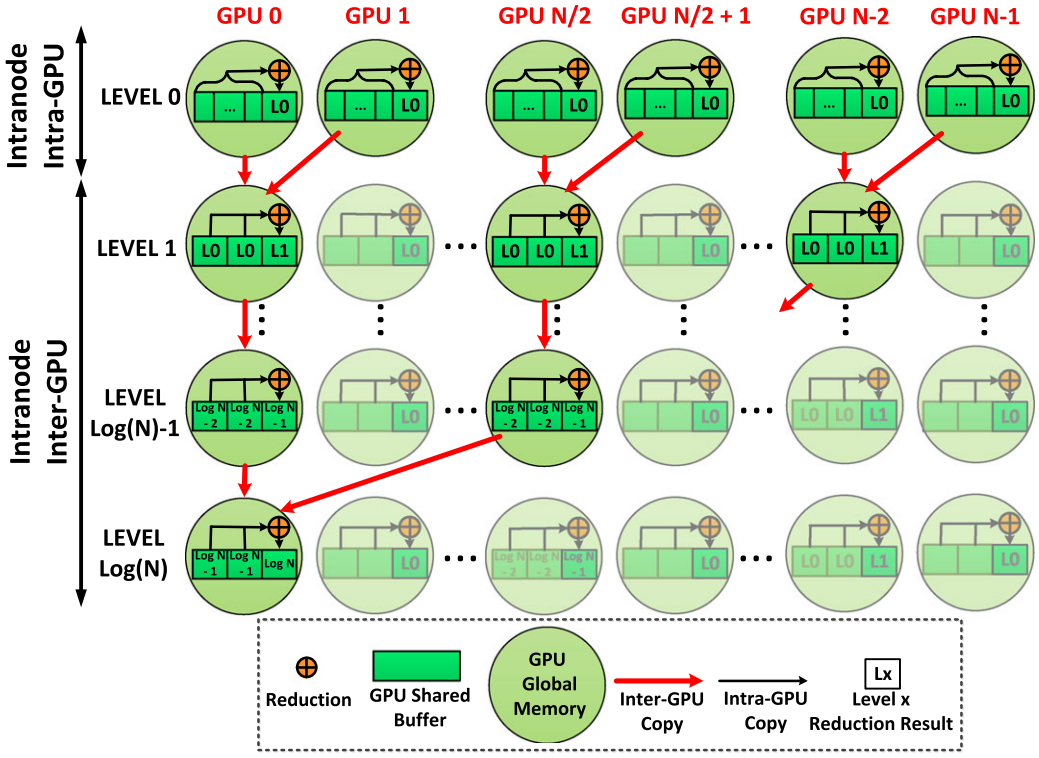


FIGURE 5 Hierarchical MPI_Allreduce utilizing Intranode Intra-GPU *GSB* Reduce and Intranode Inter-GPU *BTB* Reduce algorithms - Reduce stage

4.3 | Performance evaluation of the GPU-aware and hierarchical designs

4.3.1 | Experimental platform

Our experimental evaluations in this section are conducted on System A, which is a 4-node GPU cluster called Helios, installed at Université Laval. Each of the Helios nodes is equipped with 8 K80 GPUs, 256 GB of memory, and two Intel Xeon Ivy Bridge E5-2697 processors. Each Xeon processor provides 12 cores and operates at 2.7 GHz clock speed; thus, there exist a total of 24 cores per node. Each node runs a 64-bit CentOS 6.7 and CUDA Toolkit 7.5. The Helios nodes use QDR InfiniBand as their interconnect. We evaluate our algorithmic designs on System A that has more GPUs than System B in Section 5.3.1. We compare our proposed designs against the existing design in MVAPICH2-2.1 and present the results for various message sizes (4 B to 16 MB). We consider up to 24 processes per node (a total of 96 processes) in our tests that are uniformly distributed among different GPUs. We evaluate various configurations in our experiments by varying the number of nodes, GPUs per node, and processes per GPU.

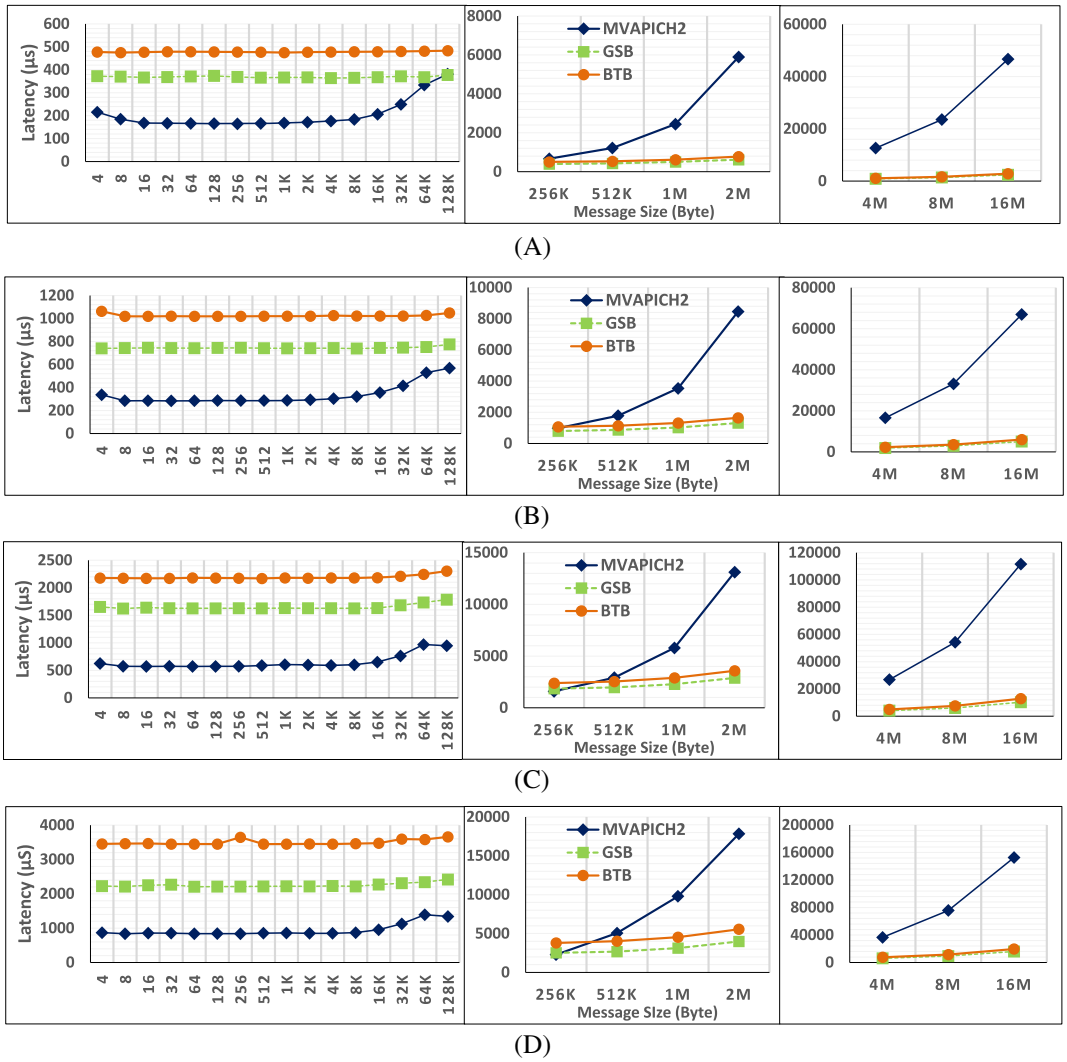


FIGURE 6 MVAPICH2 vs. *GSB* vs. *BTB* MPI_Allreduce on System A using a single node with a single GPU per node. A, 4 processes; B, 8 processes; C, 16 processes; D, 24 processes

We perform our experiments using the OSU microbenchmark that is configured to support GPUs.[27] This benchmark suite provides tests for measuring the performance of collective and point-to-point operations. The MPI_Allreduce test reports the latency of reduction sum on various message sizes. Our experiments in this section are performed on up to four nodes of System A, each having 8 GPUs. The GPUs on each node share the same root complex, and thus, all 8 GPUs can directly exchange data with each other over the PCIe topology Tree. We observed consistent microbenchmark results across different runs.

4.3.2 | Performance results for single-GPU nodes

Figure 6 compares MPI_Allreduce using our proposed *GSB* and *BTB* approaches against the MVAPICH2 design on a single GPU of System A. According to the figure, the benefit of our proposed designs starts at 256 KB message sizes. Using our proposed designs, we can observe up to 19.5 times performance improvement for large messages over the MVAIPCH2 MPI_Allreduce. This is because MVAPICH2 uses host-based data staging and reduction operation, which are costly specifically for large message sizes. On the other hand, our designs utilize a GPU shared buffer to perform direct IPC copies and in-GPU reductions. The startup and the peer synchronization of the CUDA IPC copies impose high overhead in copying short message sizes; however, as the message size increases, the startup overhead becomes negligible compared with the data transfer time.

As shown in Figure 6, the performance of the *GSB* approach is superior or at least similar to the *BTB* approach. This indicates that on a single-GPU platform, the logarithmic nature of the Binomial algorithm cannot provide any improvement over the GPU Shared Buffer Aware approach with linear complexity. The performance results indicate a consistent behavior in the *GSB* and *BTB* approaches for message sizes up to 128KB. In this range, the startup latency for the IPC copy and the in-GPU reduction kernel is almost oblivious to the message size and mainly dominates the collective runtime. However for larger message sizes, the reduction and the IPC copy time become mainly dependent on the message size. More specifically, the message size and the number of IPC calls to/from the shared buffer mainly determine the total execution time in our designs. These IPC calls cannot overlap; thus, increasing the number of processes in our design increases the total execution time accordingly.

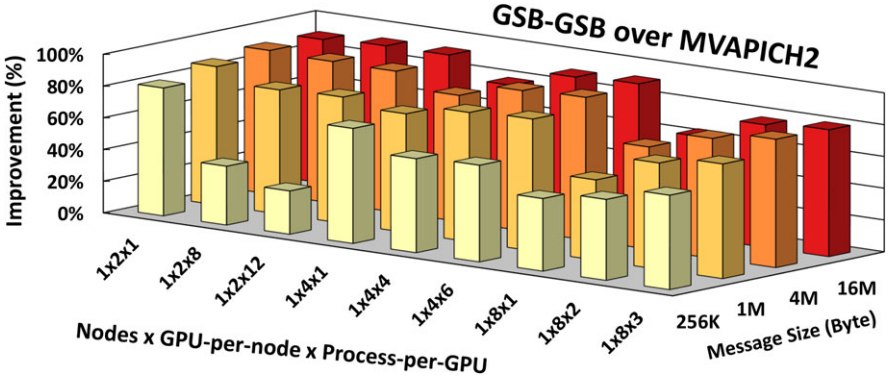


FIGURE 7 GPU Hierarchical MPI_Allreduce with GSB for intra-GPU and GSB for inter-GPU steps over MVAPCIH2 MPI_Allreduce on System A using a single node with multiple GPUs per node

4.3.3 | Performance results for GPU clusters with multi-GPU nodes

For the proposed hierarchical framework for a multi-GPU cluster, we consider the *GSB* and *BTB* designs in different intranode hierarchy levels. Therefore, we have the following design scenarios: (1) GSB-GSB, (2) GSB-BTB, (3) BTB-GSB, and (4) BTB-BTB. In each scenario, the first and the second names determine the algorithm to be used in the intranode intra-GPU and intranode inter-GPU phases, respectively.

Figures 7 and 8 evaluate our designs on System A using a single node with multiple GPUs per node. We provide cluster-wide results in Figures 9 and 10 on four 8-GPU nodes. In all of these figures, we compare two different design scenarios and provide the performance improvement percentage. Our results are reported for 256 KB to 16 MB message sizes; as for smaller message sizes, our designs provide limited or no improvement. Figure 7 compares the GSB-GSB design with the MVAPICH2 MPI_Allreduce. According to the figure, the GSB-GSB design is superior over MVAPICH2 in all test cases. We can also observe that the GSB-GSB design provide higher improvement for cases with larger message sizes and a higher number of processes per GPU.

To investigate the choice of algorithm in our framework, we compare the GSB-GSB case against the BTB-GSB, BTB-BTB, and GSB-BTB cases in Figures 8A, 8B, and 8C, respectively. According to Figure 8A, the BTB-GSB design underperforms the GSB-GSB design in all test cases. These results comply with the results in Figure 6, showing that the *BTB* design is not favored in the intra-GPU phase. Using the BTB-BTB design (Figure 8B) can lead to some performance improvement in cases with 4 and 8 GPUs. Interestingly, with the BTB-BTB design, performance improves as the number of GPUs per node increases and downgrades as the number of processes per GPUs increases. This implies that despite the inefficiency of using the *BTB* design in the intranode intra-GPU phase, it can lead to performance improvement when used in the intranode inter-GPU phase. Finally, the GSB-BTB design outperforms the GSB-GSB design in all test cases (Figure 8C). This verifies that the *GSB* and the *BTB* approach should be the algorithm of choice for the intranode intra-GPU and inter-GPU phase, respectively. Similar to our single-node analysis, our cluster-wide experiments indicate that the GSB-GSB design can outperform the MVAPICH2 MPI_Allreduce in all test cases (Figure 9), although with a lower improvement compared with the single-node results. This is an expected behavior as we are using a fixed internode algorithm for the internode phase of our framework. Thus, the higher the share of the intranode phase in the collective operation, the higher the improvement potential in our proposals.

In Figure 10, we also evaluate the choice of different algorithms for the intranode phase of a cluster-wide MPI_Allreduce operation. According to Figure 10, similar to our intranode results (Figure 8), the GSB-BTB approach is the algorithm of choice for the intranode phase of the cluster-wide MPI_Allreduce operation. In general, we conclude that at the intranode intra-GPU level, where GPU operations (such as kernel computations and inter-process communications) from different processes cannot overlap with each other, collective algorithms with lower GPU operations from different processes and lower syncrhonization among them (the *GSB* algorithm in our case) provide superior performance compared with alternative designs. On the other hand, at the the intranode inter-GPU level, where different processes are assigned to different GPUs, the GPU operations (such as kernel computations and inter-process communications) from different processes can overlap with each other; thus, collective algorithms with fewer steps and higher concurrent operations among different processes (the *BTB* algorithm in our case) can provide higher performance compared with alternative designs.

5 | MPI COLLECTIVES WITH EFFICIENT GPU DATA COPY TYPE SELECTION

In this section, we first provide some motivational results that serve as the backbone of our designs. These results also show the benefit of using the NVIDIA Multi-Process Service (MPS[15]) and Hyper-Q on inter-process GPU communications using different copy types. We then exploit the NVIDIA MPS and the Hyper-Q features and propose a *Static* algorithm and an alternative *Dynamic* algorithm for intranode GPU collectives.

5.1 | Impact of MPS and Hyper-Q features on inter-process communication

The benefit of using the Hyper-Q feature through the MPS service is already evaluated on various applications and offloaded computational kernels.[24,28-30] However, in this section, we evaluate the impact of the Hyper-Q feature and the MPS service on the point-to-point intranode

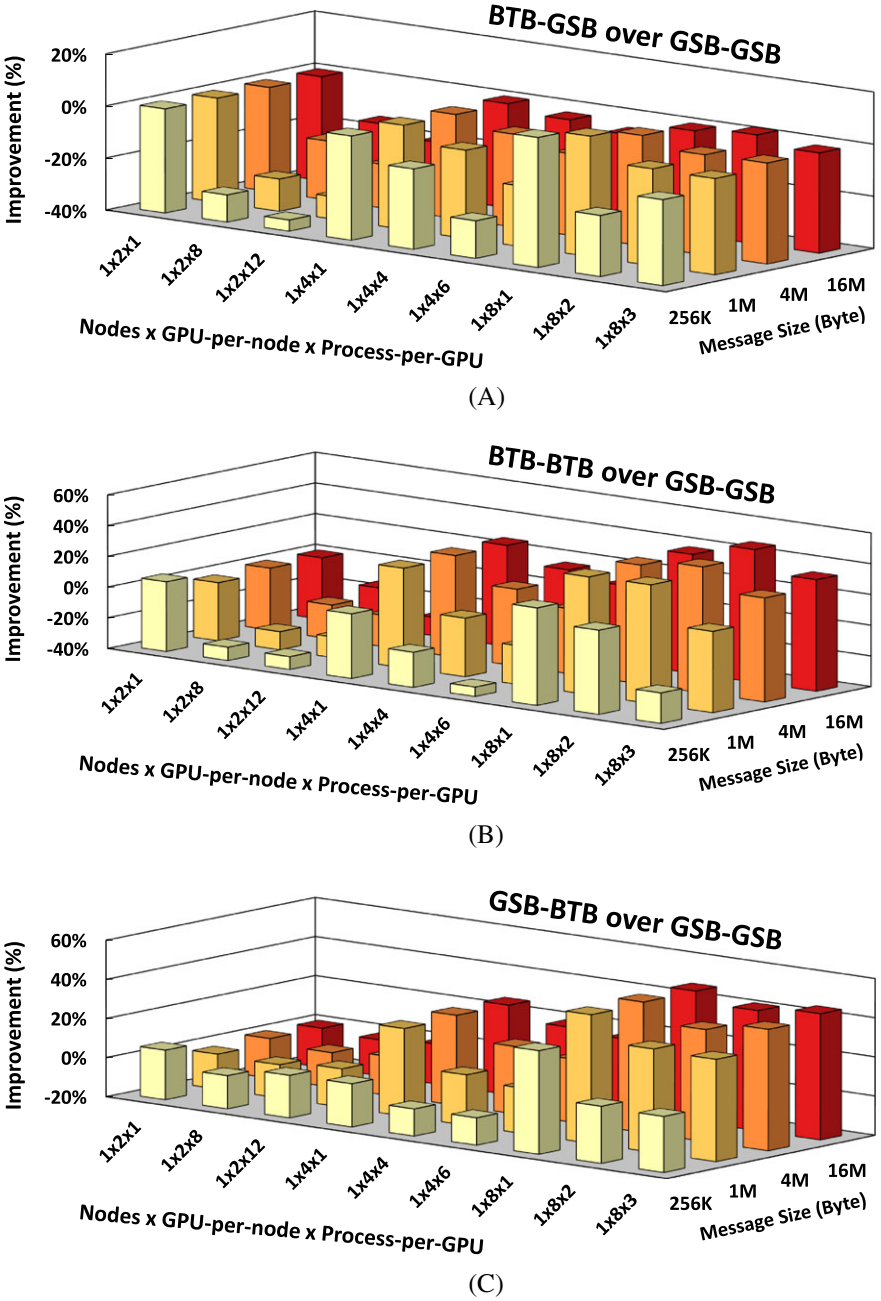


FIGURE 8 Evaluating the effect of using different algorithms in the GPU Hierarchical MPI_Allreduce on System A using a single node with multiple GPUs per node. A, Case1: Hierarchical design - Intra-GPU: *GSB*, Inter-GPU: *GSB* over Case2: Hierarchical design - Intra- GPU: *BTB*, Inter-GPU: *GSB*; B, Case1: Hierarchical design - Intra-GPU: *GSB*, Inter-GPU: *GSB* over Case2: Hierarchical design - Intra-GPU: *BTB*, Inter-GPU: *BTB*; C, Case1: Hierarchical design - Intra-GPU: *GSB*, Inter-GPU: *GSB* over Case2: Hierarchical design - Intra-GPU: *GSB*, Inter-GPU: *BTB*

communications. We show that there are certain message ranges in which a single copy type is most favored for inter-process communications. With the MPS service, this trend will not change, even though the communication performance can potentially improve. We also provide evidence that for some message sizes, utilizing multiple copy types in conjunction with the MPS service is the best way to improve multiple inter-process communications.

We have developed a microbenchmark that considers four point-to-point communicating pairs that are first synchronized and then perform pair-wise communications with either the host-staged (HS) or CUDA IPC data copy methods. In Figure 11, we consider and evaluate six different cases to perform these four point-to-point communications. In case1 (HS-only), all four point-to-point communications are performed using host-staged copies. In case2 (2HS-2IPC), two of the point-to-point communications are performed with host-staged copies, and the other two are performed using CUDA IPC copies. In case3 (IPC-only), we perform all four point-to-point communications only using CUDA IPC copies. The point-to-point communications in case4 (HS-Only-MPS), case5 (2HS-2IPC-MPS), and case6 (IPC-Only-MPS) use the same copy types as in case1, case2, and case3, respectively; however, the MPS service is disabled for case1, case2, case3, and is enabled for case4, case5, and case6. According to Figure 11, three main observations can be made. The most apparent observation is that both host-staged and IPC copy types can benefit from

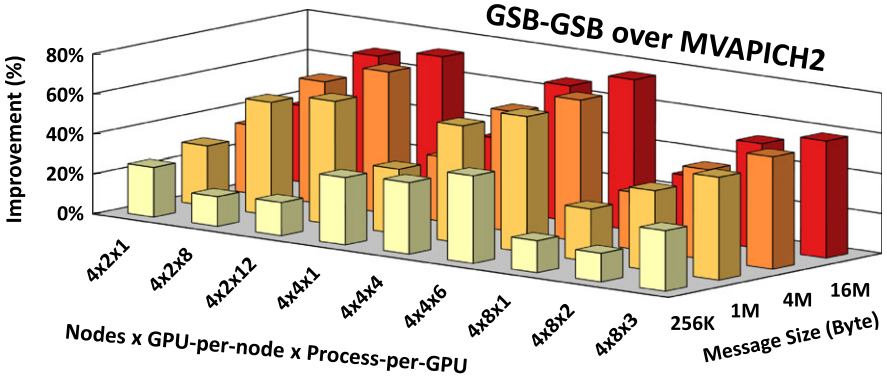


FIGURE 9 GPU Hierarchical MPI_Allreduce with *GSB* for intra-GPU and *GSB* for inter-GPU steps over MVAPCIH2 MPI_Allreduce on System A using 4 nodes with multiple GPUs per node

the MPS service mainly for small and medium message sizes. For 128KB messages and above, however, the MPS service imposes overhead on both copy types, with a larger impact on the host-staged copy. Secondly, it can be seen that the host-staged copies (with or without MPS) are faster than the CUDA IPC copies for small and medium size messages. The opposite trend can be seen for large message sizes, in which the CUDA IPC copies (with or without MPS) are superior to the host-staged copies. Finally, we can also observe that in some message sizes (32KB and 64KB) with MPS enabled, the communication using both copy types is superior to the communication with a single copy type. This implies that the Hyper-Q feature through the MPS service is providing some overlap between the host-staged and CUDA IPC communications.

While Figure 11 shows the potential benefit of the Hyper-Q feature in intranode communications with different copy types, it only considers three out of five possible combinations. For further investigations, we repeated the tests in Figure 12 with the MPS service enabled but, this time, considering all five possible cases for performing the point-to-point communications. In case1 (HS-Only–MPS), all four point-to points are performed only with the host-staged copies. In case2 (3HS-1IPS–MPS), three point-to-points are performed with host-staged copies and one with the CUDA IPC. In case3 (2HS-2IPC–MPS), two point-to-points are performed with host-staged copies and the other two with the CUDA IPC copies. In case4 (1HS-3IPC–MPS), one point-to-point is performed with host-staged copy and three with the CUDA IPC copies. In case5 (IPC-Only–MPS), all four point-to points are performed only with the IPC copies. As shown in the figure, the Hyper-Q feature through the MPS service provides faster communication when using both copy types for a larger message range (8 to 256 KB). It can be argued that there is no silver bullet combination working efficiently across all message sizes; thus, the best approach is to leverage different combinations of copy types across different message sizes.

Considering our findings, we raise the following question: how can different GPU data copy types be used in conjunction with each other to improve the performance of GPU collective communication operation? In the following, we propose two algorithms that could use the two data copy types together for performance reasons.

5.2 | Incorporating MPS and Hyper-Q features into the GPU collective design

In this section, we exploit the NVIDIA MPS and the Hyper-Q feature and propose a *Static* algorithm and an alternative *Dynamic* algorithm for intranode GPU collectives. Both of the *Static* and *Dynamic* approaches decide the number and type (host-staged or IPC) of the GPU inter-process communications that are involved in the collective operation. The *Static* algorithm makes this decision based on a priori information that it extracts from a tuning table. The *Dynamic* algorithm, on the other hand, dynamically decides the number and type of the copies at runtime. Our proposed algorithms can be applied to any collective operation; however, in this paper, we will consider MPI_Allgather and MPI_Allreduce operations as our case studies. While both the *Static* and *Dynamic* algorithms go through the same general steps (ie, Gather, Kernel Function, and Broadcast), each step has a different algorithmic design.

5.2.1 | Static Hyper-Q aware algorithm

Tuning collective operations can be performed by conducting experiments on the underlying system and exploiting the gathered information; this has been extensively studied and shown to highly improve the collective performance.[31] In our work, the tuning table for each collective associates the most efficient combination of the copy types to each message size and process count. For example, the configuration of using 10 host-staged and 5 CUDA IPC copies has shown to be the most efficient combination to perform MPI_Allgather on 16 processes and 16 KB of data.* The main steps of the *Static* algorithm are described below.

*Note that the leader process in our algorithms does not use any of the host-staged or IPC copy types.

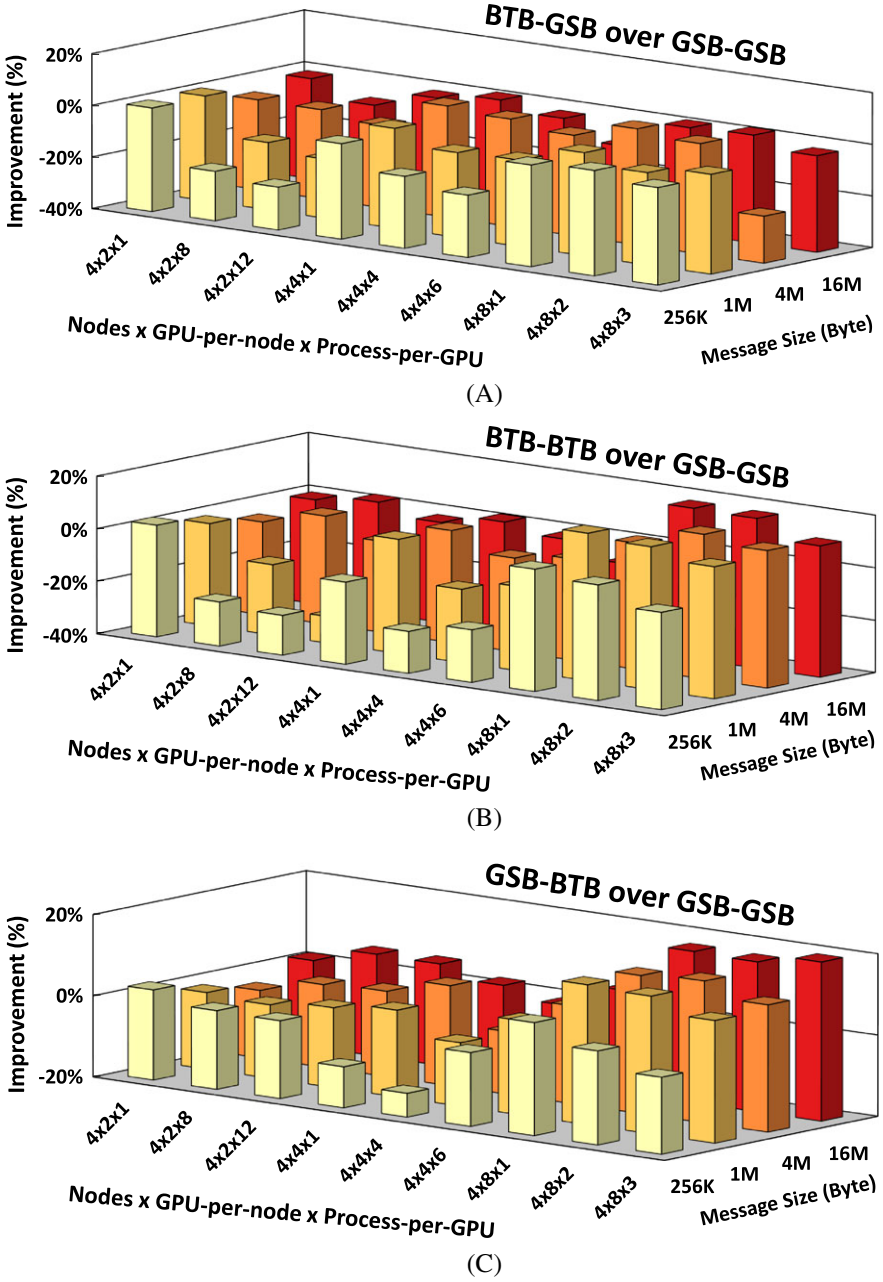


FIGURE 10 Evaluating the effect of using different algorithms in the GPU Hierarchical MPI_Allreduce on System A using 4 nodes with multiple GPUs per node. A, Case1: Hierarchical design - Intra-GPU: *GSB*, Inter-GPU: *GSB* over Case2: Hierarchical design - Intra-GPU: *BTB*, Inter-GPU: *GSB*; B, Case1: Hierarchical design - Intra-GPU: *GSB*, Inter-GPU: *GSB* over Case2: Hierarchical design - Intra-GPU: *BTB*, Inter-GPU: *BTB*; C, Case1: Hierarchical design - Intra-GPU: *GSB*, Inter-GPU: *GSB* over Case2: Hierarchical design - Intra-GPU: *GSB*, Inter-GPU: *BTB*

Stage1: Intranode Intra-GPU Gather

All processes copy their share of data into the GPU shared buffer area in a first-come first-serve order using a copy type that is assigned to them by the leader process (without loss of generality, a process with `rank 0` is considered as the leader process). The leader retrieves the most efficient combination of the copy types from the tuning table, assigns a particular copy type to each process, and then queries their completion.

Stage2: Kernel Function

In this step, a kernel function is called by the leader process on the aggregated data in its GPU shared buffer. This step is only required for some collective operations. In our test cases, only MPI_Allreduce goes through this step and performs an element-wise reduction on the aggregated data in the GPU shared buffer.

Stage 3: Intranode Intra-GPU Broadcast

The collective result is now available in the GPU shared buffer and is copied out into the destination buffer of the participating processes. Similar to the `Gather` step, the leader process assigns a particular copy type for each of the inter-process copies on the information that is extracted from the tuning table.

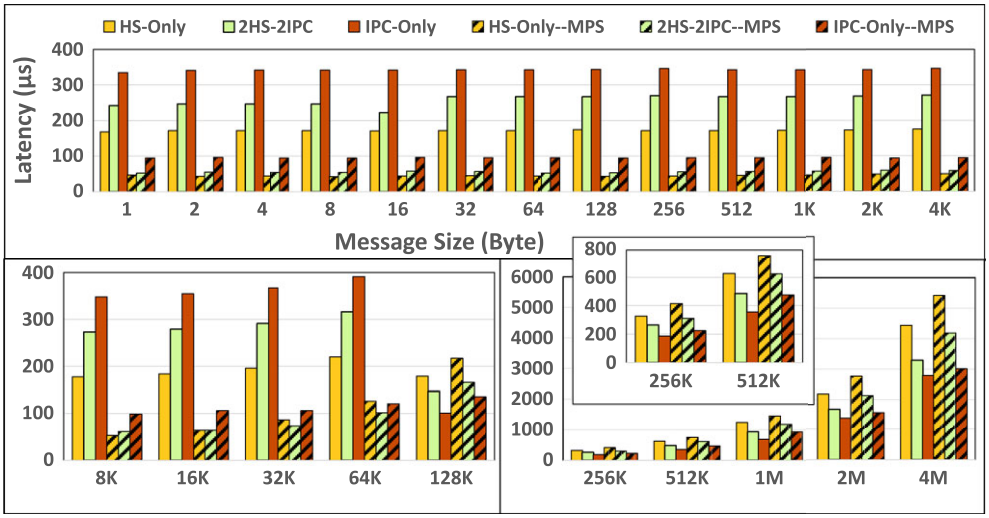


FIGURE 11 Hyper-Q effect on intranode point-to-point communication with and without MPS

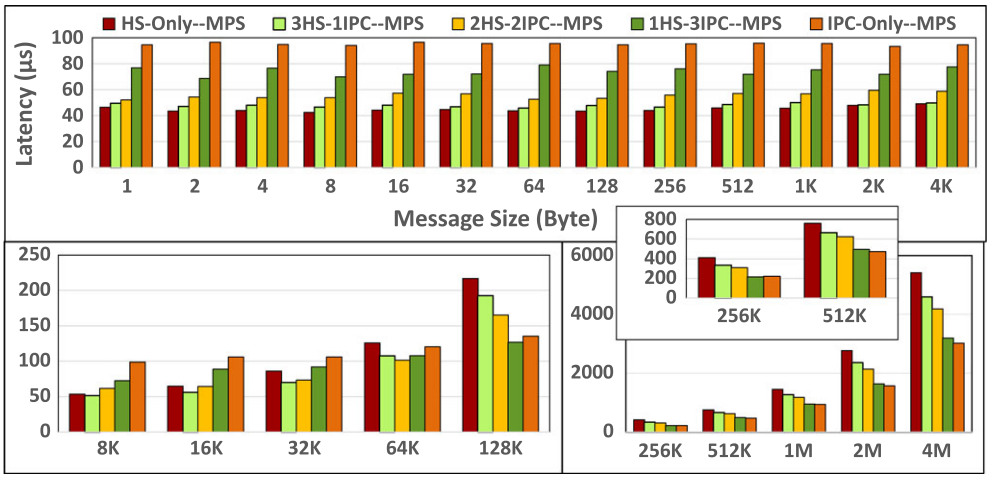


FIGURE 12 Hyper-Q effect on intranode point-to-point communication with MPS enabled

5.2.2 | Dynamic Hyper-Q aware algorithm

The *Static* algorithm is dependent on tuning parameters that must be available prior to the runtime. On top of that, the tuning parameters for a particular platform may not necessarily be useful on another platform. We propose a *Dynamic* algorithm that is independent of any tuning parameters and is capable of determining the copy type for each process by solely exploiting the runtime information. The idea behind this approach is to decide the copy types based on their availability and efficiency. We acquire this information by querying the responsiveness of these copy types. In other words, the *Dynamic* algorithm tends to choose the slower and less available copy type less frequently, whereas the faster and more responsive copy type is more frequently selected. The main steps of this algorithm are discussed below.

Stage 1: Intranode Intra-GPU Gather

All of the participating processes copy their share of data into the GPU shared buffer of the leader process. In this step, the leader process uses an algorithm that decides the type of its inter-process copy based on the responsiveness of the copy types; this algorithm goes through the following phases.

Phase1: Initialization - Considering that at this point, no prior knowledge about the copy types exists, assessing the responsiveness of a copy type can only be done by actually assigning a host-staged copy type and a CUDA IPC copy type to the first two (non-leader) processes arriving at the collective call and then querying their completion.

Phase2: Progress - The leader process queries the pending copies and waits until one completes. Our intuition is that it is more efficient to issue multiple copies on a faster copy type all at once. In this regard, once an inter-process communication using a specific copy type completes, we calculate the difference between the number of completed communications using the host-staged and the CUDA IPC copy types. A zero or a negative difference indicates that the currently completed copy type is not as fast as the other copy type; thus, only a single copy is issued with this slow but yet available copy type. A positive difference, on the other hand, indicates that the completed copy type is faster than the other type, and thus, multiple copies should be assigned to the next available processes. We determine the number of the issued copies to be two to the power of this

difference; this way, we ensure quick assignment of the inter-process communications to the faster copy type and thus using it more frequently. This procedure continues until the last copy is issued.

Phase3: Final Copy Completion - For the final pending copy, the leader takes a different approach. The rationale behind this is that the last pending copy could potentially linger for a long time and there is no pending copy using the other type. At this point, the leader checks if there has been any successful completion of this copy type before. If this is not the case, the leader considers this copy type to be extremely slow and assigns the final copy to be re-sent with the other copy type. If the slow copy turns out to be of the host-staged type, the remaining portion of the host-staged copy (if any) will be discarded. This can be supported by packetizing the host-staged copies into large chunks and sending them back to back. Once a process receives a re-send assignment with the IPC copy type, the remaining packets of the host-staged copy will be discarded. However, this approach cannot be applied to the slow IPC copy type; therefore, we overlap the slow IPC copy with the next steps of the collective operation.

Stage2: Kernel Function

This step is similar to the *Static* algorithm.

Stage 3: Intranode Intra-GPU Broadcast

Both copy types can be potentially used to broadcast the available result in the GPU shared buffer among the participating processes. However, unlike `Step 1`, the leader now has some knowledge about the efficiency of the copy types. Based on this information, this step goes through the following phases:

Phase1: Initialization - If there has been no successful completion of a copy type since the beginning of the collective operation, the leader tags it as a slow copy type and avoids using it in the broadcast step. Otherwise, both copy types can potentially be used in this step.

Phase2: Progress - If all copies are initiated using a single type, the completion of that type is only required to be queried. Otherwise, the leader monitors the progress of both copy types and uses the faster copy type more frequently and waits for all of the pending copies in the current or previous step(s) to complete before returning form the collective operation.

5.2.3 | Implementation details

Both *Static* and *Dynamic* algorithms use pre-allocated CPU and GPU shared buffers. These buffers are allocated during `MPI_Init()`, with the CPU shared buffer also being registered to prevent it from being swapped out. During `MPI_Init()`, we also check the status of the MPS service in order to choose the right tuning parameters. After the leader allocates its GPU shared buffer, it broadcasts its memory handle to the other processes. Buffer allocation and broadcasting the handle are expensive operations and thus are only performed once to mitigate their high cost.

Figure 13 illustrates different components of the Dynamic algorithm and shows how processes can communicate with each other through the CPU and GPU shared buffers. The GPU shared buffer is used to gather the pertinent data from all participating processes. The CPU shared buffer is used for staging the data in the host-staged type of copy; it also serves as a directory to track the communications between the processes. The directory is composed of two parts, the `Completion Flag` and the `Copy Status Flags`; setting the `Completion Flag` indicates that the result of the collective operation is available in the GPU shared buffer and can be copied out. The `Copy Status Flags` show the status of the copy operations. In the *Static* algorithm, these flags indicate the initiation and completion of the copies; in the *Dynamic* algorithm, these flags are also used by the leader process to assign the copy type to the other processes. Given that the use of directory in the *Static* algorithm is a simplified version of the *Dynamic* algorithm, in the following, we will only discuss the implementation details of the *Dynamic* algorithm.

The `Copy Status Flags` in the directory of the *Dynamic* algorithm for MPI_Allreduce can take one of the following eight states: `INIT`, `RTS`, `IPC_ASGN`, `HS_ASGN`, `IPC_INIT`, `HS_INIT`, `IPC_CMP`, and `HS_CMP`. The `INIT` state represents the initial state, and the `Copy Status Flags` are set to this state before entering and exiting the collective operation. The `RTS` (Ready To Send) flag indicates a process arrival to the collective operation. The `IPC_ASGN` and the `HS_ASGN` are the assignment flags, set by the leader process to assign IPC and host-staged copy types for inter-process communication, respectively. The `IPC_INIT` and `HS_INIT` indicate the initiation of the IPC and host-staged copy, respectively. The `IPC_CMP` and the `HS_CMP` indicate completion of the IPC and host-staged copy, respectively.

Figure 13 illustrates the different steps of the *Dynamic* algorithm and shows how processes can communicate with each other through the directory. Figure 13A shows the initial state, in which the `Copy Status Flags` are all set to the `INIT` state and the `Completion Flag` is set to zero. Figure 13B shows a snapshot of the gather step of the *Dynamic* algorithm. In this step, the leader process first queries the `Copy Status Flags`, looking for an `RTS` state. As can be seen in the figure, P_1 (process with `rank 1`) has arrived to the collective operation and is ready to send its data, whereas P_{n-3} has not yet arrived at the operation. All processes (except the leader) initiate their copy once their copy type is determined. In Figure 13B, the leader has assigned the host-staged copy type to P_{n-1}. The status of the P_{n-4} indicates that it has initiated its IPC copy. Depending on the initiated copy type, a different course of actions is required to guarantee its completion. The completion of the CUDA IPC copy requires synchronization between the sender and the receiver sides. In this regard, a CUDA inter-process event is recorded right after the sender process starts its CUDA IPC copy and before it sets its associated flag in the directory to `IPC_INIT`. The leader (the receiving process), once observes this flag, queries the inter-process event to check the IPC copy completion; once the copy completes, the leader sets the associated entry in the `Copy Status Flags` to the `IPC_CMP` state. In the case of the host-staged copy, the process initiating the copy is also responsible for querying its completion and setting its associated flag to the `HS_CMP`. Note that IPC and host-staged copies are initiated asynchronously, thus allowing the sending process to query the potential re-send assignment from the leader. According to the figure, P_2 and P_{n-2} have completed their host-staged copies and their share of data are available in the CPU shared buffer. The leader is also copying the staged data from P_2 and has already completed the copy

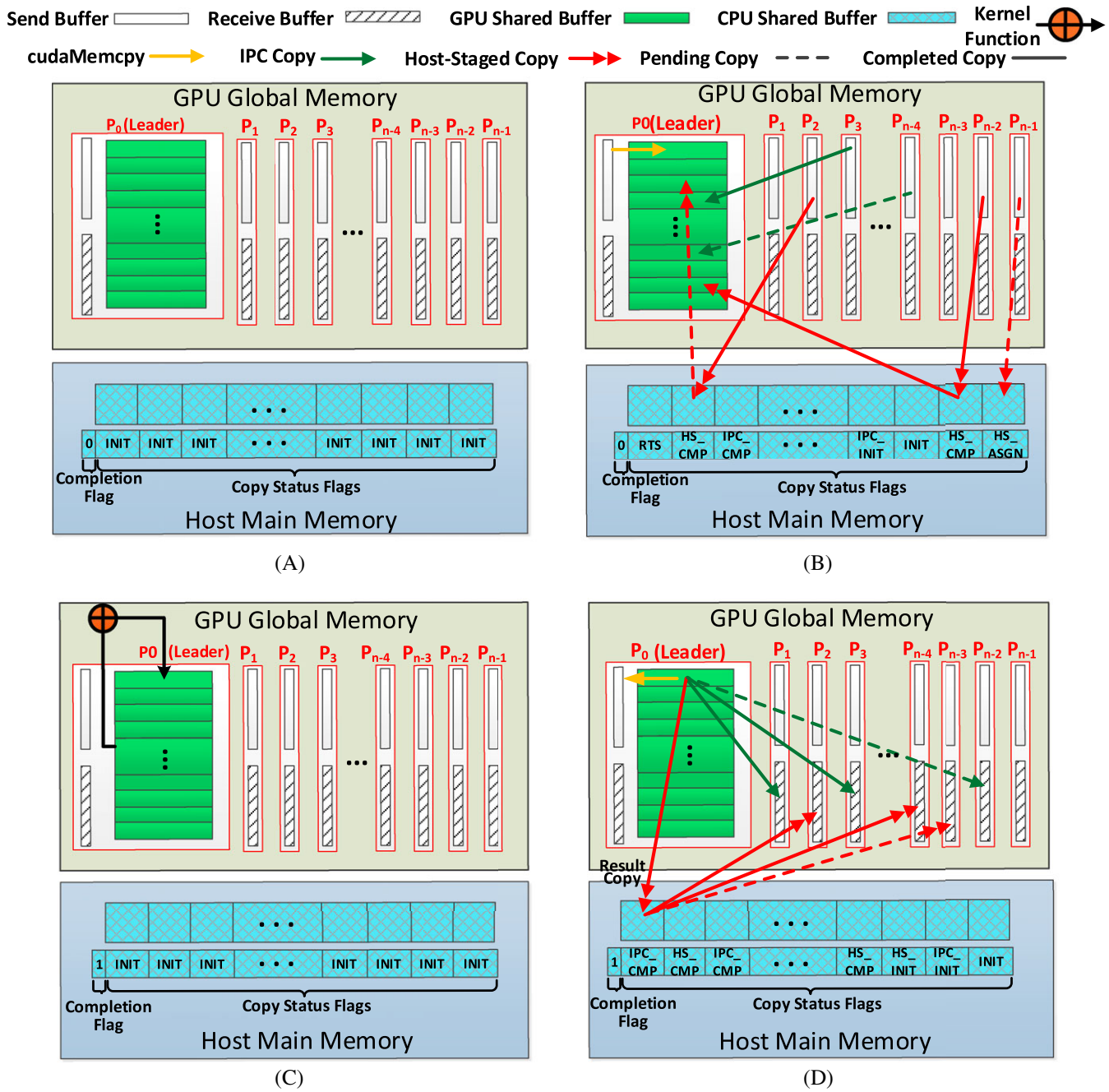


FIGURE 13 Different steps of the node-wide *Dynamic* algorithm for MPI_Allreduce. A, Initial state; B, Step 1 - Gather; C, Step 2 - Kernel function; D, Step 3 - Broadcast

from P_{n-2}. The state of P_3 shows that its IPC copy is completed. The dotted copies in the figure resemble pending copies and have the potential to be overlapped with each other using the Hyper-Q feature.

Figure 13C shows the kernel function step. This step can be skipped in some collectives (MPI_Allgather in our case) by immediately setting the `Completion Flag` and resetting all of the `Copy Status Flags` back to the `INIT` state once the gather step completes. In some collectives (MPI_Allreduce in our case) a kernel function is called on the aggregated data and the result is stored in the GPU shared buffer. Once the kernel function completes, the leader sets the `Completion Flag` to inform other processes that the result is available.

Figure 13D shows a snapshot of the Broadcast step of the *Dynamic* algorithm. In this step, the collective result is broadcast to all participating processes. The steps shown in the figure applies to MPI_Allreduce in which the reduced result is broadcast from the shared buffer to all processes. For other collective operations, this step may require some modifications; for example, in MPI_Allgather, the entire gathered data in the GPU shared buffer is broadcast to all participating processes. According to Figure 13D, using the *Dynamic* algorithm, the leader assigns IPC or host-staged copy type to other processes through the `Copy Status Flags` to notify them how and from which shared buffer (CPU or GPU) they can read their collective results. If the *Dynamic* algorithm opts to use the host-staged copy type, the leader requires to first copy the result from the GPU shared buffer to the CPU shared buffer.

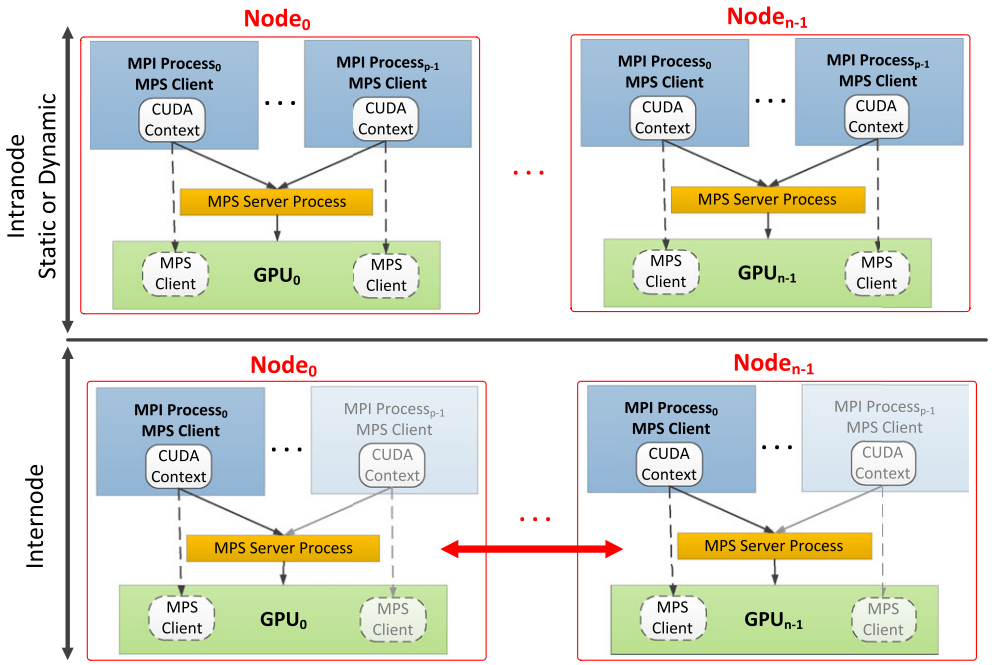


FIGURE 14 *Static* and *Dynamic* algorithms across the cluster

5.2.4 | Cluster-wide extension of the static and the dynamic algorithms

To extend the node-wide Static and Dynamic algorithms across the cluster, we propose a general three-level hierarchical framework similar with those presented in Section 4.2.1. A node-wide collective algorithm, reduce for MPI_Allreduce or gather for MPI_Allgather, is first performed among processes that share the same GPU using the static or dynamic algorithm. A cluster-wide collective operation, allreduce for MPI_Allreduce or allgather for MPI_Allgather, is then performed among the GPU leader processes on each node using the MVAPICH library. Finally, a node-wide collective algorithm, broadcast for both MPI_Allreduce and MPI_Allgather, is performed among processes that share the same GPU using the static or dynamic algorithm. In the following, we discuss the general steps involved in extending the *Static* and *Dynamic* approaches across the cluster for MPI_Allreduce and MPI_Allgather.

MPI_Allreduce

The MPI_Allreduce with the *Static* and *Dynamic* approaches is performed in three stages as follows:

Stage 1: Intranode Intra-GPU Reduce

Intranode intra-GPU processes use the *Static* or *Dynamic* approach to reduce the data into their predefined GPU leader process.

Stage 2: Internode Inter-GPU Allreduce

The GPU leader processes call MPI_Allreduce using the existing internode MVPAICH2 algorithm.

Stage 3: Intranode Intra-GPU Broadcast

The GPU leader processes use the *Static* or *Dynamic* approach to broadcast the data among the intranode intra-GPU processes.

In order to extend and use the MPS service across the cluster, an instance of this service is required to be running on each node of the cluster. The MPS server on each node allocates one instance of the GPU storage and scheduling resources that can be shared by all intranode MPI processes, which are also called MPS clients (Figure 14). This way, all intranode processes can use their own instance of the MPS service to share the GPU within their node; therefore, they can concurrently access and share the GPU resources.

MPI_Allgather

To extend the *Static* and *Dynamic* algorithms across the cluster for MPI_Allgather, this operation is performed in three stages as follows:

Stage1: Intranode Intra-GPU Gather

Intranode intra-GPU processes use the *Static* or *Dynamic* approach to gather the data into their predefined GPU leader process.

Stage2: Intranode Intra-GPU AllGather

The GPU leader processes call MPI_Allgather using the default MVPIACH2 algorithm.

Stage3: Intranode Intra-GPU Broadcast

The GPU leader processes use the *Static* or *Dynamic* approach to broadcast the data among the intranode intra-GPU processes.

5.3 | Performance evaluation of the Hyper-Q aware GPU collectives

In this section, we compare our Hyper-Q aware collective designs against MVAPICH2 and MVAPICH2-GDR and evaluate the effect of the MPS service on them. While our proposed algorithms can be applied to all collective operations, we consider MPI_Allgather and MPI_Allreduce as our

test cases. It is worth noting that the MVAPICH2 and MVAPICH2-GDR fail to use the Nvidia MPS to perform MPI_Allgather for some message sizes and process counts. We speculate this failure to be rooted in using the FGP algorithm[18] in MVAPICH2. So for these test cases, we do not have any results to report. In the rest of this section, we first provide our experimental platform; then, we discuss our results on a single-GPU platform and provide some profiling results to show how our Hyper-Q aware algorithms can successfully overlap different copy types and improve the total communication performance. Finally, we will discuss the results on the GPU cluster.

5.3.1 | Experimental platform

Our experiments in this section are conducted on a 4-node GPU cluster (System B), called Odin, at the HPC Advisory Council. Each of the Odin nodes is equipped with 3 (or 4) K80 GPUs, 64 GB of memory, and two Intel Xeon E5-2697 processors. Each Xeon processor operates at 2.6 GHz and provides 14 cores; thus, each Odin node has 28 cores. Each node runs a 64-bit RHEL 7.2 as the operating system and utilizes the CUDA Toolkit 7.5. We use System B to evaluate our Hyper-Q aware designs, which allows us to change the GPU mode and stop/start the MPS frequently. These actions require an interactive access to the GPU cluster and cannot be scheduled in a PBS script on System A in Section 4.3.1. A single GPU per node is used in our experiments on System B.

In our experiments, we compare our collective designs with the existing collectives in MVAPICH2-2.1 and the MVAPICH2-GDR-2.0. We conduct our experiments using the OSU microbenchmark that is configured to support GPUs.[27] We use this microbenchmark to measure the latency of the MPI_Allreduce and MPI_Allgather operations. For MPI_Allreduce tests, the OSU microbenchmark uses the sum reduction operation. The gathered results are consistent across multiple runs. We also provide some profiling results for our Hyper-Q aware designs with the OSU microbenchmark. We use the Nvidia Profiler (`nvprof`[32]) in conjunction with the Nvidia Tools Extension (`NVTX`) as our profiling tool. The `nvprof` presents an overview of the instructions launched by the CUDA runtime or driver APIs, whereas we used `NVTX` to annotate MPI routines and assign MPI ranks to their associated process ids and GPU contexts on the profiler timeline. Note that while we use the modified version of the OSU benchmark to get our profiling results, the original (unmodified) version of this benchmark is used to report the performance results.

5.3.2 | Performance results for single-GPU nodes

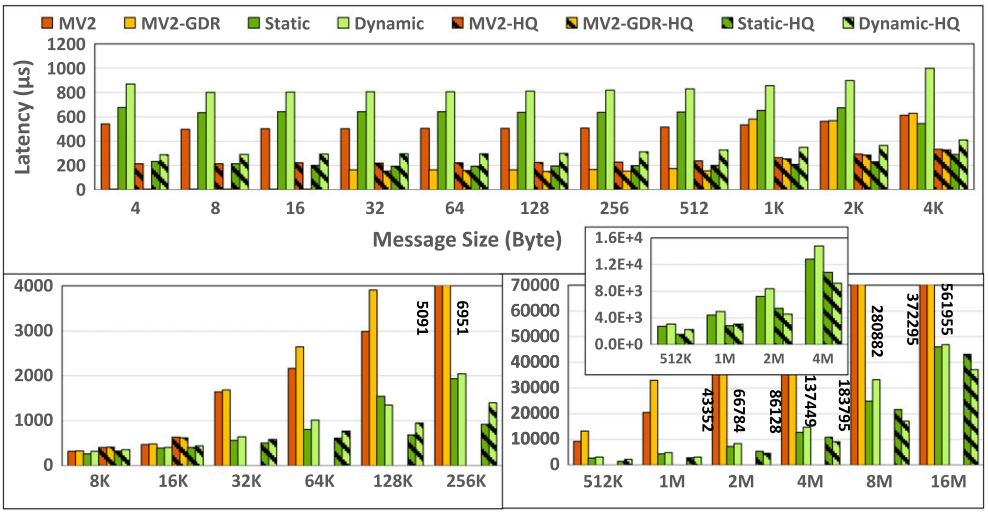
In Figure 15A, we compare the *Static* and *Dynamic* approaches against MVAPICH2 and MVAPICH2-GDR on MPI_Allgather. We provide the same comparison for MPI_Allreduce in Figure 15B. For MPI_Allgather (Figure 15A), the benefit of the *Static* and *Dynamic* approach mainly starts at 4 KB and 8 KB, respectively. For MPI_Allreduce (Figure 15B), the benefit of the *Static* and *Dynamic* approaches starts at 32 Bytes and 4 KB, respectively. For message sizes below these thresholds, both the *Static* and *Dynamic* approaches provide competitive results with the MVAPICH2 and MVAPICH2-GDR. The only exception to this is on the short message sizes (less than 16 KB) of the MVAPICH2-GDR. For these message sizes, we speculate that some features, such as gdrcopy, are the reasons behind the better performance of the MVAPICH2-GDR. In Figure 15, we can also observe that the performance of the *Dynamic* algorithm in most cases is comparable with the *Static* algorithm and that there are a few cases in which the *Dynamic* algorithm can outperform the *Static* algorithm. We associate this to the way that the tuning table is constructed for the *Static* algorithm. This table stores integer values for different configurations of the collective operation. These integer values represent the number of copy types to be used in a collective operation and are the rounded average of a thousand runs. For instance, we use 11 (10.8 rounded up) host-staged copies and 4 (4.2 rounded down) CUDA IPC copies for 64-KB message size and 16 processes. While the *Static* algorithm in different runs always stick to these rounded numbers, the *Dynamic* algorithm decides the number and type of the copy within each run. Consequently, the *Dynamic* approach has the potential to be more accurate in choosing the right number and type of the copies across multiple runs.

As shown in Figure 15, all approaches in most cases are benefiting from the MPS service. For MPI_Allgather, the *Static* and *Dynamic* approaches benefit the most from the MPS service and achieve 2.17 and 2 times speedup,† respectively. For MPI_Allreduce, on the other hand, the *Dynamic* approach benefits more from the MPS service (2.62 times), compared with the *Static* approach (2.49 times). There are two factors that are mainly contributing to the performance boost of the *Static* and *Dynamic* algorithms when the MPS service is enabled: (1) the MPS service avoids the context-switching overhead between multiple processes that are sharing the same GPU; thus, multiple inter-process communications can be issued faster with the MPS service; and (2) with the MPS service, there is an increased overlap between different inter-process copies that have the same or different copy types. We further elaborate on this in Section 5.3.3. It is noteworthy to mention that, as shown in Figure 15, the MPS service can also improve the performance of the MVAPCIH2 designs. However, the extent of the improvement provided by MPS is lower on MVAPCIH2 compared with the Hyper-Q aware *Static* and *Dynamic* designs. This is because MVAPICH2 uses a single copy type for inter-process communications in MPI_Allgather and MPI_Allreduce. The *Static* and *Dynamic* algorithms, on the other hand, make an informed decision on the number and type of the copy types, thus getting more benefit from the MPS service compared with MVAPICH2.

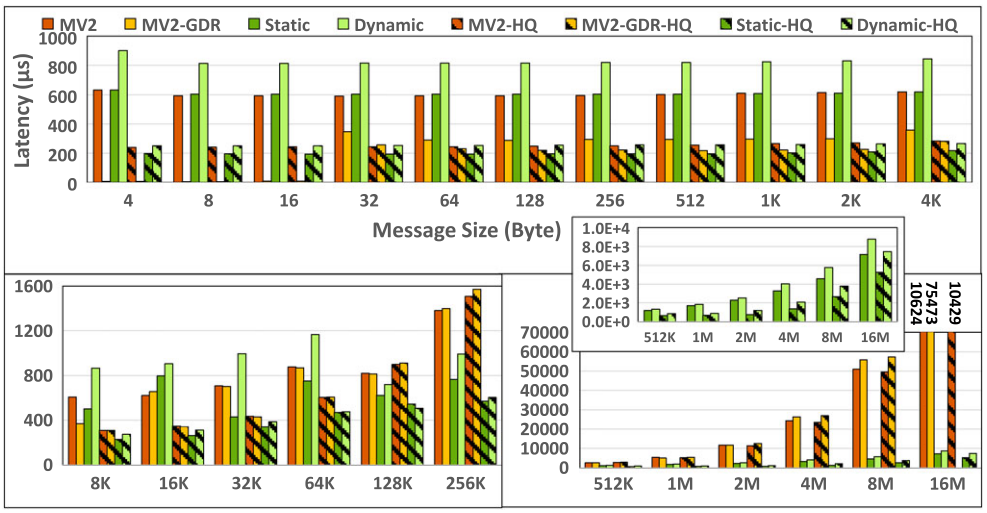
5.3.3 | Profiling results

In this section, we discuss the main reasons behind the benefit of our Hyper-Q aware algorithms. In this regard, we sheds some light on how our Hyper-Q aware algorithms work with the MPS service by providing some profiling results.

†Speed up results are averaged across all message sizes.



(A)



(B)

FIGURE 15 *Static* and *Dynamic* vs. MVAPICH2 and MVAPICH2-GDR with and without the MPS on System B using a single node with a single GPU per node. A, MPI_Allgather on 16 processes; B, MPI_Allreduce on 16 processes

The GPU computational kernels and memory operations are performed using the compute and memory engines, respectively. Without the MPS service, each engine can be assigned to a single process at a time and cannot be shared among them. A time sliced scheduler is used on the GPU to handle the requests from different processes to these engines. With the MPS service, however, requests for accessing the GPU engines are funneled through the MPS server through the only available context on the GPU (ie, the MPS context); consequently, there will be no need for any context switching.

Profiling MPS service on multiple processes communicating with the host-staged copies reveals that not only is the context switching overhead eliminated, device-to-host and host-to-device copies from different processes can also further overlap with each other. While the MPS server allows computational kernels from different MPI processes to share the compute engine on the GPU and overlap, we observed that with this service, among the three copy types (ie, host-to-device, device-to-host, and device-to-device), device-to-host and host-to-device engines cannot be shared among different MPI processes.

The MPS service highly improves the IPC copies by reducing the context-switching overhead. It also allows various CUDA IPC copies to share their local memory bandwidth and overlap their device-to-device communications. The IPC copies can also overlap with the host-staged copies. Our Hyper-Q aware designs select the right number and type of the inter-process copies to provide the maximal overlap among them through the MPS service. In Figure 16, we reflect this behavior by profiling the MPI_Allreduce operation (on 16 processes and 128 KB of data) that is implemented by our *Dynamic* algorithm. This figure is the output of the `nvvp` visual profiler that provides a runtime snapshot of the MPI_Allreduce operation with the MPS service. The profiling information in this figure is gathered using `nvprof` and `nvtx` profiling tools. According to the figure, the *Dynamic* algorithm selects 2 host-staged and 13 IPC copies in the Gather step, and 3 host-staged and 12 IPC copies in the Broadcast step of the MPI_Allreduce operation, respectively. We can also observe that with the MPS service, different inter-process copies with the same or different copy types can overlap with each other.

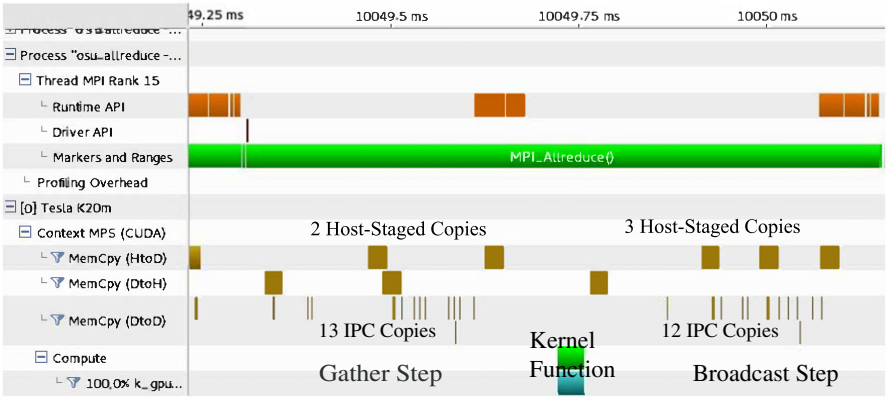
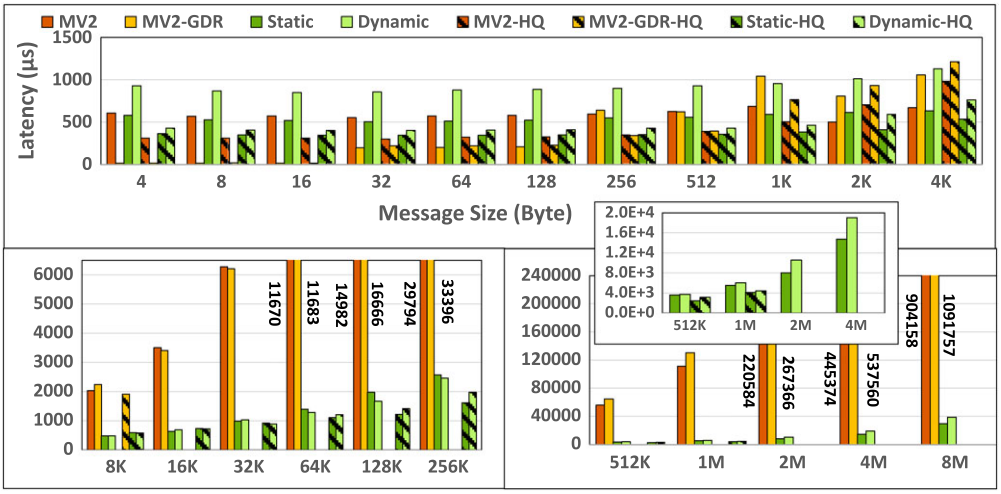
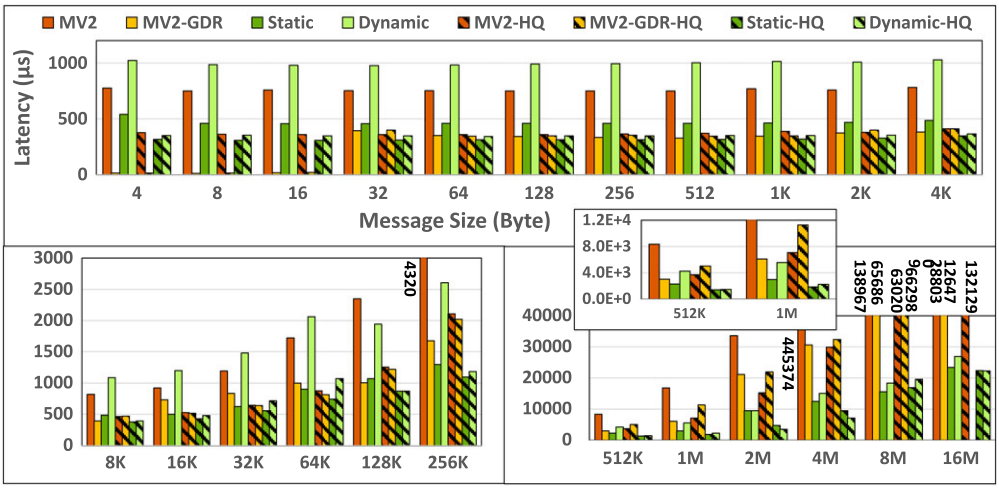


FIGURE 16 Profiling snapshot of the *Dynamic* algorithm in MPI_Allreduce with MPS



(A)



(B)

FIGURE 17 *Static* and *Dynamic* vs. MVAPICH2 and MVAPICH2-GDR with and without the MPS on System B using 4 nodes with a single GPU per node. A, MPI_Allgather on 64 processes - 16 processes per node; B, MPI_Allreduce on 64 processes - 16 processes per node

5.3.4 | Performance results for GPU clusters with multi-GPU nodes

Figure 17A provides cluster-wide comparison of the *Static* and *Dynamic* designs against the MVAPICH2 and MVAPICH2-GDR for MPI_Allgather. Figure 17B provides similar comparison for MPI_Allreduce operation. According to both figures, the *Static* approach in most cases outperforms the MVAPICH2 and MVPAICH2-GDR designs (except for message sizes less than 16 bytes on MVAPICH2-GDR). The *Dynamic* approach also in most cases provides a comparable result with the *Static* approach.

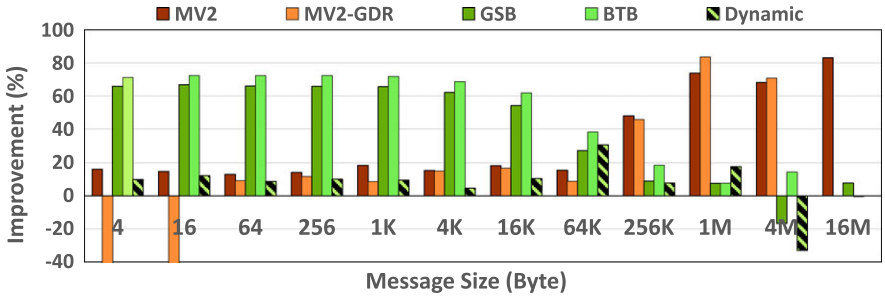


FIGURE 18 Improvement percentage of the *Static* approach over MVAPCIH2, MVAPICH2-GDR, *GSB*, *BTB*, and Dynamic for MPI_Allreduce with 64 processes using MPS - System B with 4 nodes and a single GPU per node

5.4 | Comparative analysis of Hyper-Q aware algorithms against GSB/BTB algorithms

To further investigate our Hyper-Q aware proposals and evaluate their efficiency with the presence of the MPS service, we compare them against our Hyper-Q agnostic algorithmic designs (*GSB* and *BTB*) in Section 4. Figure 18 provides the speedup achieved by using the *Static* design over MVAPICH2, MVAPICH2-GDR, *GSB*, *BTB*, and *Dynamic* designs in MPI_Allreduce. This experiment is conducted on 64 processes that are evenly distributed across 4 single-GPU nodes of the Odin cluster. As shown in the figure, the *Static* approach in most cases outperforms the rest of the designs, with few exceptions.

In general, we can conclude that while our algorithmic designs (*GSB* and *BTB*) are capable of outperforming the conventional MVAPICH2 design for large message sizes, they fall behind for small and medium message sizes. We attribute this to the high startup overhead of the IPC copies for small and medium message sizes. However, our Hyper-Q aware designs can rectify this problem by selecting the right number and type of the copies for different message sizes and process counts. In addition, the *Dynamic* approach, by dynamically selecting the right number and type of the copies, is capable of providing comparable results with the *Static* approach in most cases. The *Dynamic* approach has also the advantage of being independent to any tuning table and thus having the portability advantage.

6 | CONCLUSION AND FUTURE WORK

In GPU clusters, efficient communication plays a crucial role in the performance of MPI applications. In this paper, we propose various designs to improve the performance of GPU collective operations. In our designs, we leverage efficient algorithms in conjunction with the latest features that are available in modern GPUs.

For GPU clusters with multi-GPU nodes, we proposed a three-level hierarchical framework for GPU collective operations. We analyzed the performance of our framework by applying different algorithms into different hierarchy levels. We highlighted the importance of our hierarchical framework by showing that different hierarchy levels favor different algorithms. We evaluated our framework using MPI_Allreduce as an example of MPI collective operations. Our experiments showed promising performance results, specifically for large message sizes.

For GPU inter-process communications, different copy types with different performance characteristics can be used. Different copy types are usually favored for different message sizes. However, we observed the benefit of jointly using them when performing multiple inter-process communications. This way, different copy types can overlap with each other and speed up the total inter-process communications. Accordingly, we proposed Hyper-Q aware algorithms for GPU collectives that jointly utilize CUDA IPC and Host-Staged copy types. Our algorithms decide how to efficiently utilize these copy types in conjunction with each other to perform the inter-process communications of the collective operations. We showed the benefit of our Hyper-Q aware algorithms for different GPU collective operations. Our profiling results also verified that the overlap between different copy types is indeed the main reason behind the improvement.

We evaluated the effect of the MPS service on our proposed algorithms. The MPS service can allow different MPI processes to further overlap with each other and more efficiently share single GPU resources. Our algorithms in most test cases showed to benefit from this service; however, we achieved the highest improvement with our Hyper-Q aware algorithms.

As for future work, we would like to evaluate our designs in larger clusters. We also intend to evaluate the impact of our algorithms on real applications and compare them against the recently introduced Nvidia NCCL library.[23] We would like to extend our framework by studying other algorithms within different hierarchy levels. More specifically, we would like to investigate hierarchical collectives that are tuned for certain message sizes.

ACKNOWLEDGMENTS

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (grant RGPIN-2016-05389) and the Canada Foundation for Innovation (grant 7154). The computations in this work were performed on the Helios and Odin clusters. The Helios cluster is installed at Université Laval, which is managed by Calcul Québec and Compute Canada. The operation of this supercomputer is funded by the

Canada Foundation for Innovation (CFI), the ministére de l'Économie, de la science et de l'innovation du Québec (MESI), and the Fonds de recherche du Québec - Nature et technologies (FRQ-NT). We especially thank Maxime Boissonneault for his technical support regarding our experiments on the Helios cluster. We also would like to thank the HPC Advisory Council for providing the Odin cluster for our study. We especially thank Pak Lui for his technical assistance with conducting our experimental study on this cluster.

ORCID

Iman Faraji http://orcid.org/0000-0001-9442-7424

REFERENCES

1. The TOP500. https://www.top500.org/lists/2017/06/. Accessed September 30, 2017.
2. MPI3.1. http://www.mpi-forum.org/docs/mpi-3.1/. Accessed September 30, 2017.
3. Pena AJ, Alam SR. Evaluation of inter-and intra-node data transfer efficiencies between GPU devices and their impact on scalable applications. Paper presented at: 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing; 2013; Delft, The Netherlands.
4. Ji F, Aji AM, Dinan J, et al. DMA-assisted, intranode communication in GPU accelerated systems. In: Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems; 2012; Liverpool, UK.
5. Potluri S, Wang H, Bureddy D, Singh AK, Rosales C, Panda DK. Optimizing MPI communication on multi-GPU systems using CUDA inter-process communication. In: Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum; 2012; Shanghai, China.
6. Singh AK, Potluri S, Wang H, Kandalla K, Sur S, Panda DK. MPI alltoall personalized exchange on GPGPU clusters: Design alternatives and benefit. Paper presented at: 2011 IEEE International Conference on Cluster Computing; 2011; Austin, TX.
7. Jenkins J, Dinan J, Balaji P, Samatova NF, Thakur R. Enabling fast, noncontiguous GPU data movement in hybrid MPI+GPU environments. Paper presented at: 2012 IEEE International Conference on Cluster Computing; 2012; Beijing, China.
8. Potluri S, Hamidouche K, Venkatesh A, Bureddy D, Panda DK. Efficient inter-node MPI communication using GPUDirect RDMA for InfiniBand clusters with NVIDIA GPUs. Paper presented at: 2013 42nd International Conference on Parallel Processing; 2013; Lyon, France.
9. Wang H, Potluri S, Bureddy D, Rosales C, Panda DK. GPU-aware MPI on RDMA-enabled clusters: design, implementation and evaluation. *IEEE Trans Parallel and Distrib Sys*. 2014;25(10):2595-2605.
10. Shi R, Potluri S, Hamidouche K, et al. Designing efficient small message transfer mechanism for inter-node MPI communication on InfiniBand GPU clusters. Paper presented at: 2014 21st International Conference on High Performance Computing (HiPC); 2014; Dona Paula, India.
11. Chu C-H, Hamidouche K, Venkatesh A, Awan AA, Panda DK. CUDA kernel based collective reduction operations on large-scale GPU clusters. Paper presented at: 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid); 2016; Cartagena, Colombia.
12. Faraji I, Afsahi A. GPU-aware intranode MPI_Allreduce. In: Proceedings of the 21st European MPI Users' Group Meeting EuroMPI; 2014; Kyoto, Japan.
13. Faraji I, Afsahi A. Hyper-Q Aware intranode MPI collectives on the GPU. In: Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware, ESPM2 '15; 2015; Austin, TX.
14. NVIDIA Corporation. http://www.nvidia.com
15. NVIDIA, "MPS", Sharing a GPU between MPI processes: Multi-Process Service - vR352; 2015.
16. MVAPICH2. http://mvapich.cse.ohio-state.edu. Accessed October 14, 2016.
17. Open MPI. http://www.open-mpi.org/. Accessed October 14, 2016.
18. Singh AK. Optimizing All-to-All and Allgather Communications on GPGPU Clusters [PhD thesis]. Columbus, OH: The Ohio State University; 2012.
19. Tipparaju V, Nieplocha J, Panda D. Fast collective operations using shared and remote memory access protocols on clusters. In: Proceedings International Parallel and Distributed Processing Symposium; 2003; Nice, France.
20. Graham RL, Shipman G. MPI support for multi-core architectures: Optimized shared memory collectives. Paper presented at: European Parallel Virtual Machine/Message Passing Interface (PVM/MPI) Users Group Meeting; 2008; Dublin, Ireland.
21. Li S, Hoefler T, Snir M. NUMA-aware shared memory collective communication for MPI. In: Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing HPDC; 2013; New York, NY.
22. Mamidala AR, Kumar R, De D, Panda DK. MPI collectives on modern multicore clusters: Performance optimizations and communication characteristics. Paper presented at: 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID); 2008; Lyon, France.
23. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl. Accessed September 30, 2017.
24. Wende F, Steinke T, Cordes F. Multi-Threaded Kernel Offloading to GPGPU using Hyper-Q on Kepler Architecture [Technical Report]. Berlin, Germany: Zuse Institute Berlin; 2014.
25. LeBeane M, Potter B, Pan A, et al. Extended task queuing: Active messages for heterogeneous systems. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis; 2016; Salt Lake City, UT.
26. Hoefler T, Lumsdaine A, Rehm W. Implementation and performance analysis of non-blocking collective operations for MPI. In: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing; 2007; Reno, NV.
27. Bureddy D, Wang H, Venkatesh A, Potluri S, Panda DK. OMB-GPU: A Micro-benchmark Suite for Evaluating MPI Libraries on GPU Clusters. Paper presented at: European MPI Users' Group Meeting (EuroMPI); 2012; Vienna, Austria.
28. Chen Z, Xu J, Tang J, Kwiat K, Kamhoua C, Wang C. GPU-accelerated high-throughput online stream data processing. *IEEE Trans Big Data*.
29. Pompili A, Di Florio A, CMS Collaboration. GPUs for statistical data analysis in HEP: a performance study of GooFit on GPUs vs. RooFit on CPUs. *J Phys Conf Ser*. 2016;762:012-044.

30. Brown WM, Carrillo JMY, Gavhane N, Thakkar FM, Plimpton SJ. Optimizing legacy molecular dynamics software with directive-based offload. *Comput Phys Commun*. 2015;195:95-101.
31. Vadhiyar SS, Fagg GE, Dongarra J. Automatically tuned collective communications. In: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing; 2000; Dallas, TX.
32. NVIDIA profiler user's guide. http://docs.nvidia.com/cuda/profiler-users-guide/. Accessed March 3, 2017.

How to cite this article: Faraji I, Afsahi A. Design considerations for GPU-aware collective communications in MPI. *Concurrency Computat Pract Exper*. 2018;e4667. https://doi.org/10.1002/cpe.4667