Design and Implementation of an IPC-based Collective MPI Library for Intel GPUs

Chen-Chun Chen
chen.10252@osu.edu
The Ohio State University
Columbus, Ohio, USA

Goutham Kalikrishna Reddy Kuncham
kuncham.2@osu.edu
The Ohio State University
Columbus, Ohio, USA

Pouya Kousha
kousha.2@osu.edu
The Ohio State University
Columbus, Ohio, USA

Hari Subramoni
subramoni.1@osu.edu
The Ohio State University
Columbus, Ohio, USA

Dhabaleswar K. Panda
panda@cse.ohio-state.edu
The Ohio State University
Columbus, Ohio, USA

ABSTRACT

With the rising demand for computing power in High-Performance Computing and Deep Learning applications, there is a noticeable trend in outfitting modern exascale clusters with accelerators. In recent years, Intel has been designing and developing GPU products and their associated ecosystems. Concurrently, application developers are transitioning their programs to Intel GPUs, seeking to maximize the computational capabilities of multi-GPU systems by utilizing efficient communication facilitated by modern GPU-aware MPI libraries. Hence, it is critical to design an efficient MPI collective library specifically tailored for Intel GPUs to optimize communication performance. In this paper, we proposed hybrid and IPC-based designs for data movement collective MPI operations on contemporary Intel GPU systems. For large message communication, we developed a comprehensive design for data movement collectives that surpasses reliance on basic send/recv pairs, effectively minimizing overheads. For small messages, we employ CPU staging techniques and compare various underlying libraries to ensure optimal performance. We evaluate the benefits of our designs at both the benchmark and application layers on the Intel DevCloud, utilizing 4 Intel GPUs connected with X^e Links. In benchmark-level evaluations, our Alltoall and Allgather implementations show a constant 100 µs improvement for large messages, while other operations like Bcast achieve a 72x performance enhancement compared to MPICH at 32MB. In application-level evaluations, our proposed designs demonstrate up to a 30% improvement for the HPC application heFFTe compared to the second-best solution using MPICH.

KEYWORDS

Intel GPUs, GPU-aware MPI, IPC

PEARC '24, July 21–25, 2024, Providence, RI, USA

ACM ISBN 979-8-4007-0419-2/24/07
https://doi.org/10.1145/3626203.3670549

ACM Reference Format:
Chen-Chun Chen, Goutham Kalikrishna Reddy Kuncham, Pouya Kousha, Hari Subramoni, and Dhabaleswar K. Panda. 2024. Design and Implementation of an IPC-based Collective MPI Library for Intel GPUs. In *Practice and Experience in Advanced Research Computing (PEARC '24), July 21–25, 2024, Providence, RI, USA*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3626203.3670549

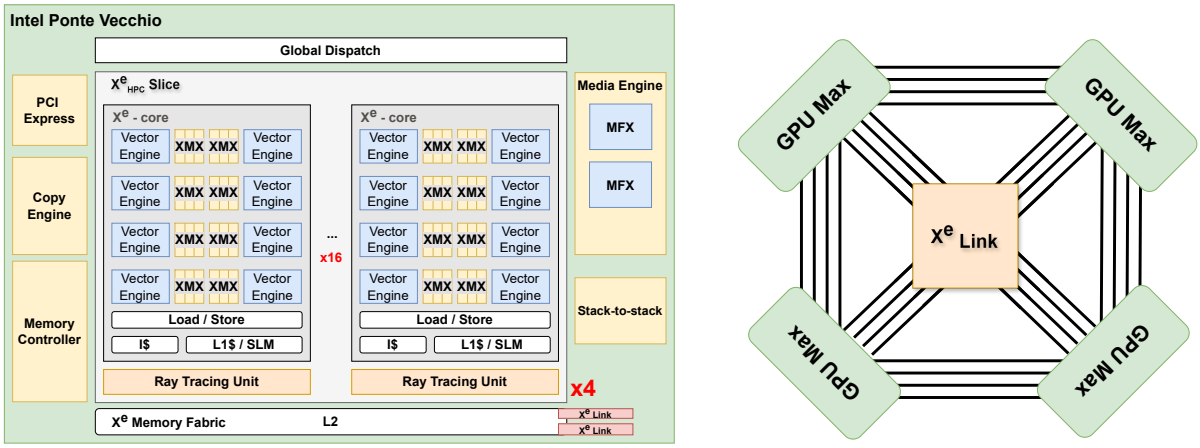
1 INTRODUCTION

Accelerators, such as Graphics Processing Units (GPUs), are pivotal in contemporary High-Performance Computing (HPC) ecosystems. Both HPC and Deep Learning (DL) applications heavily depend on the computational power provided by GPUs. The emergence of generative AI has underscored the significance of GPU performance, with leading systems in the TOP500 rankings utilizing these processors to enhance the execution of intricate HPC/DL tasks. To facilitate the optimized scaling of GPU-based applications, an efficient communication library is essential for enabling streamlined data movement among GPUs.

Despite entering the GPU market later, Intel is actively involved in the design and development of various GPU products and their associated ecosystems. In 2020, Intel introduced the Iris X^e Max, followed by the Intel Data Center GPU Max series. Figures 1(a) depict the micro-architecture of Intel X^e HPC stack used in Intel Data Center GPU Max series. Each stack contains 4 slices and each slice contains 16 X^e cores, computing units of the Intel GPUs. Each X^e core consists of Vector Engines and Matrix Engines, referred to as Intel X^e Matrix Extensions (Intel XMX). These cores are equipped with Vector Engines are designed to speed up traditional graphics, computing, and HPC tasks. Additionally, they include eight 4096-bit Intel XMX units optimized for accelerating AI workloads.

Furthermore, Intel GPUs are integrated into the Aurora supercomputer, securing the second position in the TOP500 list as of November 2023. It's important to note that Aurora's performance metrics were submitted with a measurement on only half of the planned final system. The success of Aurora underscores the effectiveness of Intel GPUs, emphasizing the necessity of developing an efficient MPI communication library tailored for HPC systems with Intel GPUs.

To enhance the usability of Intel GPUs, Intel introduced the oneAPI framework, encompassing the SYCL backend for high-level



(a) The micro-architecture of the Intel X^e HPC Stack comprises four slices, each containing 16 X^e cores. It is equipped with 48GB of HBM2e memory.

(b) 4-Way Intel X^e link configuration, with each X^e link capable of 26.5 GB/s of bandwidth in each direction.

Figure 1: The example overview of Intel Data Center GPU Max and the intro-node connection via X^e links.

programming on these platforms. Additionally, the oneAPI suite incorporates Level Zero library, providing lower-level options that afford developers the flexibility to optimize data movement strategies according to their specific requirements. Like NVLink for NVIDIA GPUs, Intel offers Intel X^e links for high-performance interconnectivity among its GPUs. It facilitates load and store operations, bulk data transfers, and synchronization of semantics. This technology serves as a scalability element capable of enabling communication for up to eight X^e HPC Stacks through an integrated switch. Figure 1(b) exhibits four-way scalable configurations, representing one of the potential X^e configurations.

1.1 Motivation

In the context of emerging dense GPU systems with multiple Intel GPUs, designing efficient GPU collective operations is crucial to fully leverage the potential of interconnected GPU architectures. Modern high-performance MPI solutions increasingly prioritize low latency and high bandwidth communication. Low latency is essential for efficiently transmitting smaller message sizes due to their frequent exchange. Conversely, the focus shifts to achieving high bandwidth for larger messages, particularly in applications with substantial communication payloads. Optimizing for high bandwidth becomes essential to meet the needs of these data-intensive applications, ensuring that larger messages are transmitted with maximum efficiency and minimal delay.

Over recent years, Intel has expanded its GPU ecosystem from a single GPU to multiple GPUs, connecting them with X^e links. This evolution encourages application developers to transition their programs to the Intel system, taking advantage of multi-GPU computation facilitated by GPU-aware MPI libraries. heFFTe serves as an exemplary application harnessing the power of multiple Intel GPUs through MPI collective operations. It has crafted GPU kernels using oneAPI and the oneMKL library, strategically employing Alltoall(v) communication patterns to facilitate efficient data transfer between GPU memories. This emphasizes the need for creating an efficient collective library specifically designed for Intel GPUs to optimize the performance of these applications.

GPU inter-process communication (IPC) optimizes data movement among GPUs within a node. Hence, MPI library developers leverage this technique to transfer large amounts of data between GPUs, particularly in point-to-point communications. While the IPC protocol provides high-bandwidth communication performance, there are currently limited optimized designs for collective operations, especially for Intel GPUs. Existing Intel GPU-aware MPI libraries, such as MPICH, showcase robust implementations for point-to-point communication using IPC techniques for large messages. However, these libraries often reuse point-to-point calls in collective operations without a comprehensive and optimized design. This practice introduces additional overhead, highlighting the need for more tailored and efficient designs in the collective communication layer to enhance overall performance.

1.2 Challenges

We address the following challenges to design and implement a hybrid data movement collective MPI library for intra-node communication:

- What strategies and techniques should be employed to design and implement a high-performance MPI library for data movement collective operations?
- How can we design a library for intra-node communication across multiple GPUs, leveraging the high speed and bandwidth of Intel X^e links?
- How do we determine the optimal choice of libraries (e.g., oneAPI/SYCL, Level Zero) for memory allocation and copying in the CPU staging approach?
- How can we devise a comprehensive design for data movement collectives, moving beyond the reliance on basic send/recv pairs and minimising overheads?

- Can we enhance the performance of HPC and DL applications relying on MPI collective operations, particularly on modern dense Intel GPU systems?

1.3 Contributions

This paper makes the following contributions:

- Implement and optimize collective operations for the small message, employing the CPU staging approach alongside SYCL APIs and fast memcpy techniques. (Section 3.2, Figure 4)
- Design and implement a comprehensive IPC solution specifically crafted for managing large messages, ensuring minimal introduction of unnecessary synchronization overheads. (Section 3.3, 3.4, and 3.5)
- Analyze various protocols and implement a hybrid design to deliver optimal performance across all message sizes. (Figure 4)
- Evaluate MPI collective operations and compare our proposed designs with Intel MPI and MPICH. Our Alltoall implementation exhibits a constant 100 µs improvement for large messages, while Bcast achieves a 72x performance enhancement compared to MPICH at 32 MB. (Figure 6 to 11)
- Demonstrate that our proposed designs yield up to a 30% improvement for heFFTe in the application-level evaluation compared to the second-best solution using MPICH. (Figure 12)

To the best of our knowledge, our proposed design is the first to outperform Intel MPI and MPICH libraries for intra-node data movement collective operations on Intel GPUs.

2 BACKGROUND

2.1 Intel GPUs

The Intel Data Center GPU Max series [5], also referred to as Ponte Vecchio or PVC is a powerful accelerator that's specifically designed to handle demanding workloads in deep learning, artificial intelligence, and HPC. This new GPU product is based on the Intel X^e-HPC micro-architecture with a compute-focused, programable and scalable element called the X^e-core. X^e-core includes specialized matrix engines also referred to as Intel X^e Matrix Extensions (Intel XMX) for accelerating tasks such as matrix multiplication, which is commonly used in AI training and inference. Intel Data Center GPU Max 1100 is one of the products from the Intel Data Center GPU Max series that was launched in early 2023 and comes equipped with 56 X^e cores and 48GB of HBM2e memory.

2.2 GPU-aware MPI

In traditional MPI implementations, when communicating between two GPUs on different nodes, developers have to manage the transfer of data between them using memcpy. However, with GPU-aware MPI, the users can provide either a host or device buffer directly to the MPI library. During data transfer initiation, the MPI implementation internally identifies the type of buffer. Based on the size and type of buffer, MPI implementation employs optimized algorithms and selects the most efficient pathway for communication. This intelligent approach ensures that data transfers are executed with minimal latency and maximal throughput, enhancing the overall performance.

2.3 Level-Zero Library

The Intel Level Zero (L0, ZE) [6] is a low-level API that is designed to facilitate interaction with accelerator devices, with the added benefit of flexibility through support for a broad set of features, such as unified shared memory, synchronization primitives, and device function pointers. The primary objective of this implementation is to provide a system-level programming interface that enables higher-level runtime APIs and libraries to target heterogeneous hardware. Alongside this core functionality, the L0 implementation provides various other features such as device partitioning, instrumentation, debugging, power management, frequency control, and hardware diagnostics.

2.4 Inter-process Communication (IPC)

MPI processes operate within distinct address spaces, necessitating the use of inter-process communication techniques to exchange data with each other. Similarly, in the realm of GPU inter-process communication, the GPU IPC feature optimizes data movement among GPU processes within a node. CUDA IPC [11] enables direct data copying from the GPU address space of one process to another without host intervention. To achieve this, a process must expose a portion of its address space to the remote processes, thereby creating a memory handle for the shared address. Following this, the handle is transferred to the remote process. Upon reception, the remote processes gain access to and can modify the shared remote address space using the provided IPC memory handle.

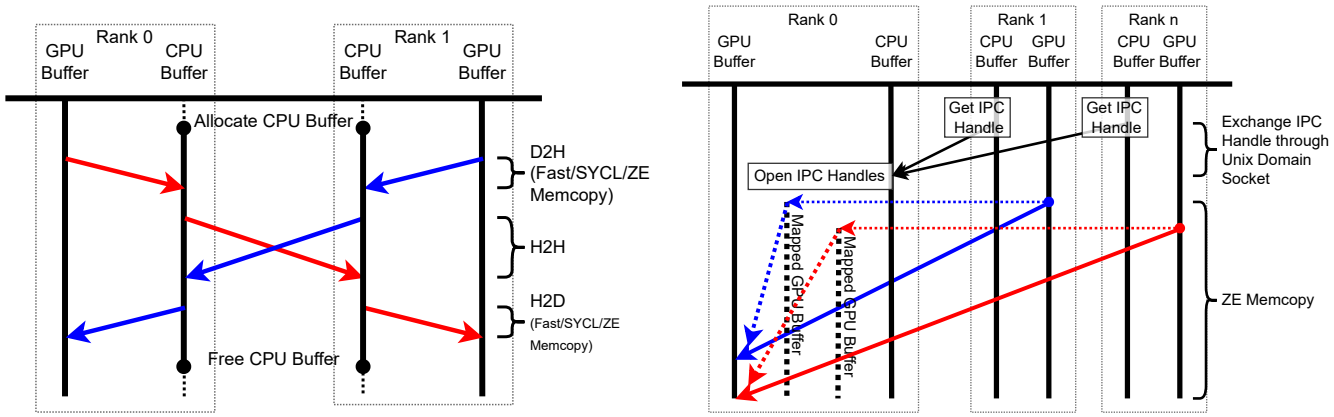
3 DESIGNING THE LARGE MESSAGE COMMUNICATION USING IPC TECHNIQUE

3.1 Overview of the Designs

We delve into the details of the hybrid and IPC-based data movement collective designs in two parts: CPU Staging approaches and IPC designs. These approaches excel in different message size ranges. Typically, IPC designs involve the exchange of IPC handles with peers for initialization, introducing some overhead. These overheads become more significant for small message communication. Therefore, for small messages, we adopt CPU staging approaches to mitigate the IPC initialization overhead. We implemented our designs based on MVAPICH-Plus, building upon the foundation of MPI point-to-point operations as base implementations for the collective operations. In this work, we support common MPI operations such as Alltoall, Allgather, Bcast, Gather, and Scatter. Additionally, we also provide support for the "v" series MPI collectives, including Alltoallv. In this paper, our focus will be on discussing Alltoall, given its dense communication pattern.

3.2 CPU Staging Approaches for Small Messages

CPU staging is the most naive approach to handle GPU buffers when implementing a GPU-aware MPI library. This method involves allocating a host buffer in CPU memory, copying the data from the device to the host, performing regular MPI operations for communication between processes, and storing the data back into



(a) CPU Staging Approaches (b) IPC Designs

Figure 2: Overview of the timeline details for CPU staging approaches and IPC designs.

the device buffer from the host staging buffer. Figure 2(a) provides detailed time results for implementing a CPU staging approach for collective operations. The red and blue arrows indicate the actual data transfer between the device and host buffer and processes.

Given that most MPI libraries optimize MPI collective operations for CPU buffers, the primary overhead that can significantly impact overall performance is the speed of memory copy operations. The previous work [4] has demonstrated that the buffer type (whether allocated with C or SYCL APIs) can influence data copying performance. In our implementation, we leverage two underlying libraries for GPU data allocation and copying: SYCL and Level Zero. Table 1 provides a summary of common APIs used by CUDA, SYCL, and Level Zero. Note that when using Level Zero APIs, the process involves creating and handling a *CommandList* to execute memory copying operations. When copying data, a memory copy command is appended to the *CommandList*, but the operations are not executed until the execute API is called. Synchronization APIs can be used to wait for the command to finish or the command can be tracked with an event. Following this, the *CommandList* also needs to be reset for subsequent usage. The performance implications of this choice will be discussed in Figure 3 and Section 4.2. With the Level Zero library, we can also employ a fast memcpy technique to efficiently copy data between the host and device, particularly beneficial for scenarios involving reused GPU buffers. When encountering a new device buffer for the first time, we create the IPC handle and use the `mmap` APIs to map the device buffer to host space. With a caching design, it can utilize the mapped space after the second time, and the memory copying API can be a simple C `memcpy`. This fast memcpy approach delivers excellent performance for very small messages, ranging from 1 byte to 128 bytes. We also compare the performance in Figure 4 and Section 4.2.

3.3 IPC Designs for Large Messages

IPC enables direct data copying from the GPU address space of one process to another without host intervention. In the Intel GPU ecosystem, it can leverage high-bandwidth data transferring using X^e Links. In general, many data movement collective operations rely on basic point-to-point *Send* and *Recv* calls to transfer data between processes. Consequently, they implement basic IPC support within these point-to-point operations. This implies that there are no other optimizations for the collective operations. While it may be convenient for MPI developers, this approach introduces overhead in terms of launching multiple point-to-point operations and the initialization of IPC handles. Instead of calling a multitude of point-to-point operations in the collective implementation, we consider the entire data movement path as a holistic picture. Figure 2(b) illustrates our designs from the perspective of Rank 0 only. The process begins by creating IPC handles for the required device buffers for all peer ranks. Subsequently, these IPC handles are sent to the target peer processes. In the example, Rank 1 and Rank n create IPC handles for their corresponding device buffer in the host memory and transmit them to Rank 0 through Unix Domain Sockets. Upon opening the IPC handles received from the peer processes, a virtual memory space is mapped to the remote device buffer, enabling the current process to access it. In the example, Rank 0 opens the IPC handles from Rank 1 and Rank N, gaining access to the remote buffers. Having access to peers' device buffers allows us to use the Level Zero APIs to facilitate memory copy between GPUs. With full access to the peers' buffers, we can aggregate multiple memory copy commands into a single command list and execute and synchronize it once to avoid redundant launches and synchronizations.

From an implementation perspective, Level Zero exhibits similar logic and provides comparable APIs to CUDA. For instance, it employs `zeMemGetIpcHandle` and `zeMemOpenIpcHandle` for creating and opening IPC handles. Additionally, an event can be employed to track the completion of a command. In our collective implementation, we simply utilize `zeCommandQueueSynchronize` to wait for all commands to be completed.

3.4 IPC Handler Exchanging through UNIX Domain Socket

During the IPC handles exchanging step, it is necessary to receive the IPC handle from the peer processes and then open it in the

Table 1: The difference mechanism and APIs between CUDA, SYCL and Level Zero.

Category	CUDA	SYCL	Level Zero
Memory Allocation	cudaMalloc	sycl::malloc_device	zeMemAllocDevice
	cudaMallocHost	sycl::malloc_host	zeMemAllocHost
	cudaMallocManaged	sycl::malloc_shared	zeMemAllocShared
	cudaFree	sycl::free	zeMemFree
Synchronization	cudaDeviceSynchronize	sycl::queue::wait	zeCommandQueueSynchronize
IPC	cudaIpcGetMemHandle		zeMemGetIpcHandle
	cudaIpcOpenMemHandle		zeMemOpenIpcHandle
IPC Handle Exchange	Direct send IPC Handle(s) to Peer(s)		Through Unix Domain Socket
Memory Copy	cudaMemcpy	sycl::queue.memcpy	zeCommandListAppendMemoryCopy zeCommandListClose zeCommandQueueExecuteCommandLists zeCommandListReset

current process. While CUDA and ROCm IPC handles can be easily exchanged through various inter-process data transferring approaches, such as shared memory, Level Zero IPC handles can only be exchanged using UNIX Domain Sockets (UDS). As UDS can be established once and reused, we establish the connection at the MPI_Init stage. During runtime, we simply reuse the connection and exchange the IPC handles by making use of `sendmsg` and `recvmsg`. This incurs minimal overhead that can be ignored.

3.5 Data Movement Using IPC

In the Naive IPC design, which uses *Send* and *Recv* pairs to transfer data in a collective operation, developers do not need to modify their algorithms, especially on the receiving side. This is because the receiving processes are aware of where to place the received data into the proper buffer location. However, in our proposed designs, the sender processes take on the responsibility of copying the data to the correct remote buffer regions using the Level Zero APIs so they have to know the exact destinations. In common MPI operations such as Bcast, Scatter, and Allgather, sender processes can calculate the offset of the destinations and copy the data to the appropriate region easily. However, it becomes more challenging for the "v" series collectives, such as Alltoallv, because the destination offsets vary and are invisible to the sender processes. In our implementation, we employ several Gather operations to collect the destination offsets for the sender processes. The evaluation in Figure 11 indicates that there are minimal overheads compared to the Alltoall designs.

4 EVALUATION

4.1 Experimental Setup

Our experiments were conducted on the Intel Developer Cloud, also referred to as Intel DevCloud. The compute node utilized for these experiments features a dual-socket Intel Xeon Platinum 8480+ CPU with 56 physical cores per socket, totalling 112 logical cores. Additionally, the node is equipped with 512 GB of memory and four Intel Data Center GPU Max 1100 GPUs, known as Intel PVC 1100. Each PVC 1100 GPU comprises 56 X^e Cores and 48GB of HBM2e memory. The GPUs are interconnected via high-speed X^e link bridges operating at a speed of 26.5 GB/s.

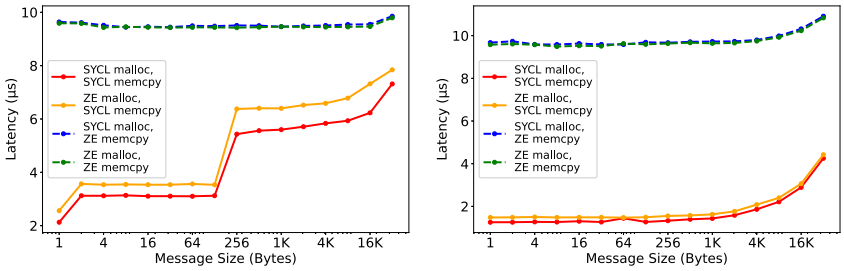
Benchmark-level evaluation: The OSU Micro-Benchmarks (OMB) [2] suite offers MPI-level evaluations for CUDA and ROCm device buffers on NVIDIA and AMD GPUs. Notably, it lacks support for Intel GPUs. To address this limitation, we enhanced OMB version 7.3 to facilitate the allocation of oneAPI/SYCL device buffers specifically for Intel GPUs. Each data point was subjected to 5 iterations, and the mean of the medium values was calculated for analysis.

Application-level evaluation: The heFFTe [1] application provides an exceptionally efficient Fast Fourier Transform (FFT) library with GPU kernel support. While conventional FFT problems rely on point-to-point operations for data exchange, heFFTe enhances efficiency by employing Alltoall or Alltoallv operations concurrently within different process groups. To meet API requirements for Alltoall operations, heFFTe utilizes a data padding approach for effective implementation.

To benchmark our design, we employed Intel MPI 2021.11 and MPICH 4.2.0 as the baseline. Note that utilizing IPC features with Intel MPI requires sudo privileges on Intel DevCloud; therefore, we disabled it with Intel MPI. However, IPC can still be used with MPICH and our proposed designs.

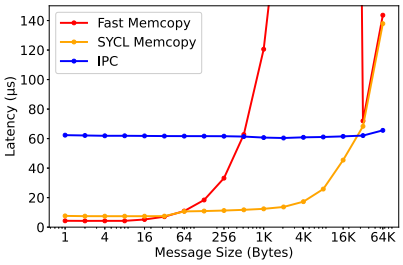
4.2 Protocol Selection

The type of CPU staging buffer chosen can have a substantial impact on data copying performance. The observation that a buffer allocated with oneAPI exhibits superior performance serves as a reminder that performance outcomes may differ when utilizing Level Zero. Therefore, we conducted a performance analysis comparing the utilization of oneAPI/SYCL and Level Zero libraries. Figure 3 illustrates the latency of data movement when employing combinations of SYCL or ZE allocated buffers and SYCL or ZE memory copy APIs. Figure 3(a) displays the performance of device-to-host memory copying, while Figure 3(b) showcases the host-to-device side. The solid lines represent the use of SYCL memcpy. Clearly, SYCL memcpy consistently delivers superior performance compared to ZE memcpy, regardless of the malloc API employed. Additionally, SYCL malloc demonstrates a slightly better performance compared to ZE malloc for both device-to-host and host-to-device cases. Based on the evaluation results, it is recommended to adopt SYCL APIs for allocating the CPU staging buffer and performing data transfers between device and host buffers.



(a) Device-to-Host (b) Host-to-Device

Figure 3: Comparison of data movement performance (D2H, H2D) involving different `malloc` and `memcpy` approaches, encompassing SYCL and Level Zero (ZE) libraries.



(a) Small Messages

Figure 4: Comparison of Alltoall latency utilizing different protocols (fast memcopy, SYCL memcopy, and IPC) on 4 Intel GPUs.

While the CPU staging approach yields consistently low latencies for small messages, the figures exhibit a notable increase beyond medium-sized messages, such as 4K. In Figure 4, the OMB Alltoall latencies are compared between the pure CPU staging approach and IPC designs. Within the CPU staging approach, we further evaluated two different protocols, employing fast memcpy and SYCL memcpy. In the case of very small messages, the fast memcpy approach exhibits the lowest latency, ranging from approximately 4 to 5 µs for message sizes between 1 and 16 bytes. For medium-sized messages (32 B to 16 KB), the SYCL memcpy approach achieves the lowest latencies, ranging from 7 to 45 µs. Despite the constant overhead of around 60 µs in IPC designs, the CPU staging approach incurs higher latencies beyond 32 KB. The evaluation results highlight that no single approach can universally perform well. Fine-tuning the thresholds of each protocol is essential, and these values may vary for different collective operations. However, with this experience, the tuning knowledge gained can be readily extended to optimize performance for other collective operations.

4.3 Profiling of Different Approaches

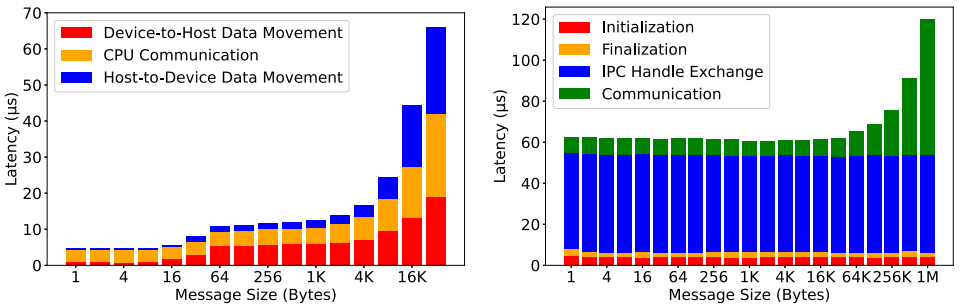
Given that different designs demonstrate excellence in varying ranges, we conducted further profiling of MPI operations to analyze our designs. In Figure 5, the time profiling results for the CPU staging approach and the IPC designs for Alltoall are depicted. Figure 5(a) illustrates the three primary stages of CPU staging, comprising D2H memory copying (in red), CPU-based Alltoall communication (in orange), and H2D memory copying (in blue). For message sizes ranging from 1 byte to 2 KB, the CPU communication times remain consistent and nearly identical, spanning from 3 to 5 µs. The primary factor influencing the total time is the memory copying time. The use of fast memory copy techniques mitigates the memory copying overhead for message sizes from 1 byte to 16 bytes, resulting in a relatively small overhead. Beyond 8 KB, both the memory copying time and the communication time increase, with the memory copying time growing even faster. This leads to the total running time surpassing the running time of using IPC designs at 64 KB.

Figure 5(b) shows the four main stages of our IPC designs, encompassing initialization (in red), IPC handle creation, exchange, and opening (in blue), IPC communication (in green), and finalization (in orange). The initialization and finalization phases are around 4 µs and 2 µs, which are relatively small and can be disregarded. However, a significant overhead is observed in the IPC handle exchange part, with a constant 47 µs used regardless of the message size. This is attributed to the time spent on IPC handling initialization and construction rather than the communication itself. In addition, the IPC communication introduces its own overhead, approximately around 8 µs, noticeable particularly for small messages under 32 KB. For large messages (>32 KB), the communication time increases in direct proportion to the message size.

The profiling results elucidate how time is allocated across different approaches, particularly highlighting the IPC designs. Despite the communication time being relatively small, there is a substantial amount of initialization overhead in the IPC designs, and it explains the necessity of employing the CPU staging approach for small message communication.

4.4 Micro-Benchmark Evaluation: Collective

We conducted a performance evaluation of our data movement collective designs, encompassing Alltoall, Allgather, Bcast, Gather, Scatter, and Alltoallv using OMB on Intel DevCloud with 1 node and 4 GPUs. In comparison to the baseline MPICH, we observed that MPICH only implements its IPC features for Alltoall and Allgather. Figures 6 to 11 illustrate the latencies for each collective, with separate figures for small and large messages. We emphasize the Alltoall numbers in Figure 6 as it represents the most dense communication, and MPICH has optimizations specifically tailored for this collective operation. For very small messages (<16 B), our proposed designs exhibit the lowest latency, approximately around 4 µs, compared to 8 µs using Intel GPU and 15 µs using MPICH. As for larger messages (16 B to 4 KB), the latency of MPICH increases significantly to around 160 to 200 µs, while Intel MPI ranges from 20 µs to 50 µs, and our approach maintains the lowest latency at 11 µs to 16 µs. With the IPC feature disabled for Intel MPI, the latency of Intel MPI increases significantly for large messages. In contrast, MPICH and our proposed designs maintain a very low latency for



(a) CPU Staging Approach (b) IPC Designs

Figure 5: Time profiling results of the CPU staging approach and the IPC designs.

large messages. For example, MPICH exhibits a latency of 3378 µs, while our proposed designs achieve a latency of 3258 µs at 32 MB. In fact, for large messages (>1 MB), our designs demonstrate approximately constant 100 µs lower latencies compared to MPICH. Similar performance trends are observed for Allgather, as shown in Figure 7. Our proposed designs exhibit approximately 100 µs lower latencies compared to MPICH for large messages. Notably, for small messages where MPICH lacks optimized designs, it experiences a latency of around 165 µs, while our designs and Intel MPI achieve significantly lower latencies at only 3 or 4 µs. For Bcast, Gather, and Scatter, as illustrated in Figure 8, Figure 9, and Figure 10, respectively, Intel MPI, MPICH, and our designs demonstrate competitive low latencies for small messages, typically in just several microseconds. However, our designs showcase outstanding performance compared to the others for large messages. For instance, in Bcast, our designs achieve 550 µs at 32 MB, while Intel MPI and MPICH exhibit latencies of 29000 µs and 39000 µs, respectively.

We have also extended our implementation to the "v" series collective operations. Figure 11 presents the performance numbers for Alltoallv, and the results and trends are very similar to our Alltoall performance, with only a slight overhead. Our designs consistently deliver the best performance across all message sizes when compared to the other baselines. Given that MPICH lacks an IPC design for Alltoallv, it exhibits a latency of 348000 µs at 32 MB, whereas our designs achieve a significantly lower latency of only 3250 µs, making it approximately 100 times faster.

Broadly speaking, our designs demonstrate better or competitive performance for small messages with the CPU staging approach using fast memcopy or SYCL memcopy techniques. Additionally, they exhibit significantly better performance for large messages utilizing our IPC designs.

4.5 Application-Level Evaluation

To assess the benefits of our design for real applications, we conducted application-level experiments by running the HPC heFFTe application. Figure 12(a) displays the performance in terms of throughput using Alltoall as the backend, while Figure 12(b) employs Alltoallv as the backend, respectively. In the case of IPC designs for Alltoall, both MPICH and our implementations outperform Intel MPI, achieving throughputs of 56 and 82 GFlops/s in the first two cases, while Intel MPI lags with only 18 GFlops/s. Remarkably, in the last case, our designs achieved a throughput of 128 GFlops/s, surpassing MPICH's 98 GFlops/s by 1.3 times.

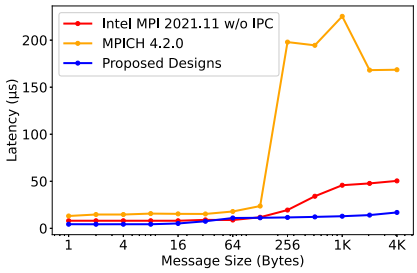
In Alltoallv communication, MPICH lacks IPC optimization, resulting in throughput ranging from 2 to 6 GFlops/s across all three cases. This performance is even worse than Intel MPI's 18 to 26 GFlops/s. In contrast, our proposed designs demonstrate superior throughput, achieving 59, 85, and 130 GFlops/s, marking a significant improvement of 24, 33, and 20 times compared to MPICH. Notably, these values even surpass the performance achieved using Alltoall in Figure 12(a). Ultimately, in addressing the same problem, our proposed design leverages optimized Alltoallv communication to outperform the second solution of MPICH with Alltoall communication by 5%, 4%, and 30%, demonstrating superior performance.

5 RELATED WORK

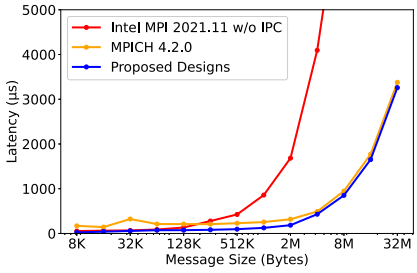
In recent years, GPUs have gained popularity for compute-intensive tasks and are now standard in modern clusters. As a result, the need for efficient communication between GPU-to-GPU became crucial. Wang et al. [16] proposed an optimal CUDA-based design for Infini-Band clusters to achieve efficient GPU-GPU communication. In addition, Wang et al. [15] proposed a novel approach to enhance data transfers between GPUs in RDMA-enabled clusters without the intervention of the CPU. Potluri et al. [12] proposed a novel hybrid design that uses host-based pipelining and GPUDirect RDMA features in CUDA to optimize inter-node GPU-GPU communication. Kawthar et al. [7] compared the point-to-point communication performance of various GPU-aware MPI libraries such as MVAPICH2-GDR, Spectrum MPI, and Open MPI + UCX. Chen et al. [3] optimize the GPU Alltoall communication with IPC designs for dense GPU systems.

While the majority of research has concentrated on CUDA and NVIDIA GPUs, Kuznetsov et al. [10] reported their experience of porting classical Molecular Dynamics (MD) applications from CUDA to ROCm using the HIP framework. Kondratyuk et al. [8] analyzed the MD applications performance on NVIDIA and AMD GPUs. Kawthar et al. [14] proposed novel ROCm-aware MPI designs for MVAPICH2-GDR library for inter and intra-node communication on AMD GPU clusters. Chen et al.[4] proposed GPU-aware MPI designs for Intel GPUs using staging approaches.

In the realm of SYCL and Intel GPUs, several studies have reported the performance of SYCL applications. Reguly et al. [13]

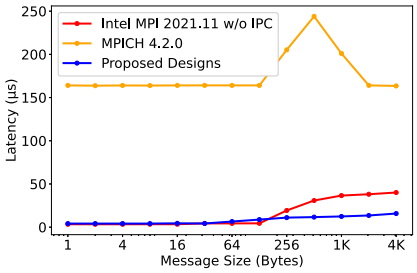


(a) Small Messages

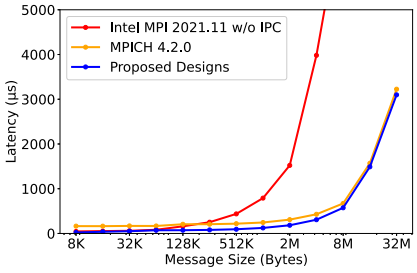


(b) Large Messages

Figure 6: Alltoall

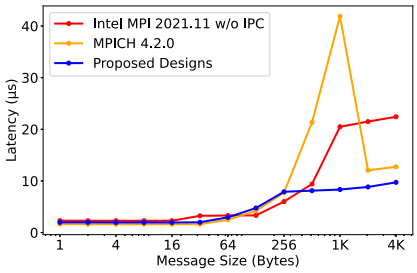


(a) Small Messages

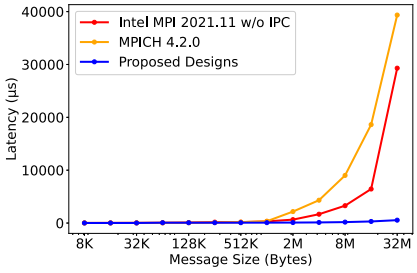


(b) Large Messages

Figure 7: Allgather

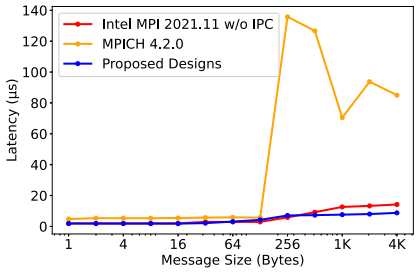


(a) Small Messages

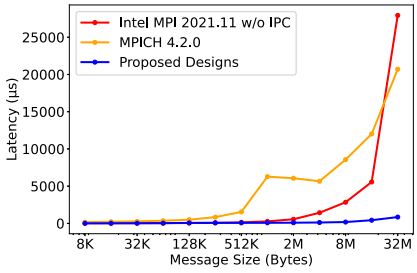


(b) Large Messages

Figure 8: Bcast

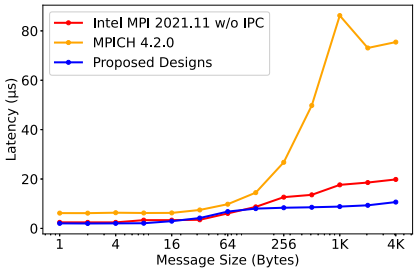


(a) Small Messages

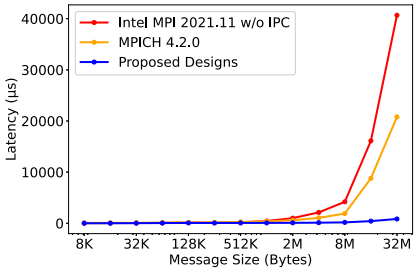


(b) Large Messages

Figure 9: Gather

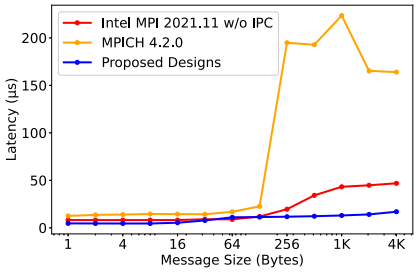


(a) Small Messages

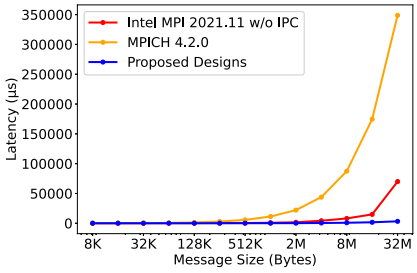


(b) Large Messages

Figure 10: Scatter



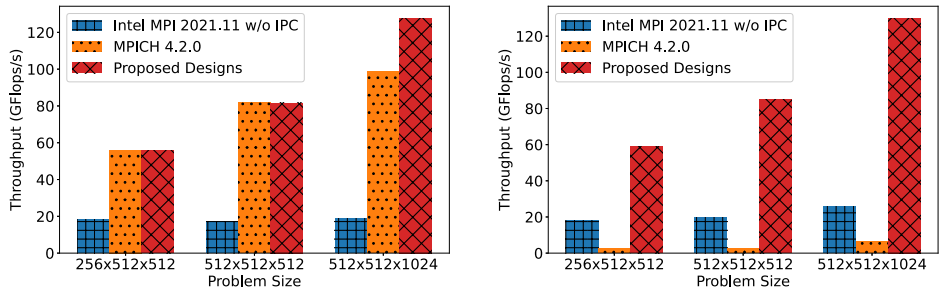
(a) Small Messages



(b) Large Messages

Figure 11: Alltoallv

examined the performance of one particular application on different devices using multiple programming models, including SYCL. Kuncham et al. [9] ported the CUDA applications to SYCL and evaluated the performance of SYCL and CUDA on NVIDIA GPUs. Zhai et al. [17] designed an SYCL-based GPU backend for Microsoft SEAL APIs.



(a) Alltoall (b) Alltoallv

Figure 12: Comparison of application-level (heFFTe) performance on 4 Intel GPUs. (Higher is better)

6 CONCLUSION

As the HPC industry moves towards adopting Intel GPUs and more applications are being ported to SYCL-based implementations, efficient support for collective communications becomes crucial at the MPI level. Over the past decade, libraries like MVAPICH2-GDR and Open MPI have improved support for NVIDIA and AMD GPUs. Such advancements have raised expectations for similar support and optimizations for data transfer on Intel GPUs using cutting-edge MPI libraries. Current MPI libraries like MPICH often rely on point-to-point communication for collectives operations which imposes additional overhead. To address this, we implemented IPC-based data movement designs to optimize collective operations. Our approach combines hybrid staging for smaller messages and IPC-based designs for larger ones, resulting in significantly lower latency compared to Intel MPI and MPICH. During benchmark evaluations, our implementations of Alltoall consistently exhibit a steady 100 µs improvement for large messages, while operations such as Bcast demonstrate a 72-fold increase in performance compared to MPICH at 32 MB. Furthermore, in application-level evaluations, our proposed designs show a significant improvement of up to 30% for heFFTe compared to the second-best solution using MPICH. In the future, we intend to extend these proposed designs to optimize MPI collectives based on reduction operations such as allreduce and reduce. We will assess the performance of these enhancements in both multi-GPU and multi-node environments.

ACKNOWLEDGMENTS

This research is supported in part by NSF grants #1818253, #1854828, #2007991, #2018627, #2311830, #2312927, and XRAC grant #NCR-130002.

REFERENCES

[1] Alan Ayala, Stanimire Tomov, Azzam Haidar, and Jack Dongarra. 2020. heFFTe: Highly Efficient FFT for Exascale. In *Computational Science – ICCS 2020*, Valeria V. Krzhizhanovskaya, Gábor Závodszky, Michael H. Lees, Jack J. Dongarra, Peter M. A. Sloot, Sérgio Brissos, and João Teixeira (Eds.). Springer International Publishing, Cham, 262–275.

[2] D. Bureddy, H. Wang, A. Venkatesh, S. Potluri, and D. K. Panda. 2012. OMB-GPU: A Micro-benchmark Suite for Evaluating MPI Libraries on GPU Clusters. In *Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface (EuroMPI)* (Vienna, Austria). 110–120.

[3] Chen-Chun Chen, Kawthar Shafie Khorassani, Quentin G. Anthony, Aamir Shafi, Hari Subramoni, and Dhabaleswar K. Panda. 2022. Highly Efficient Alltoall and Alltoallv Communication Algorithms for GPU Systems. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 24–33. https://doi.org/10.1109/IPDPSW55747.2022.00014

[4] Chen-Chun Chen, Kawthar Shafie Khorassani, Goutham Kalikrishna Reddy Kuncham, Rahul Vaidya, Mustafa Abduljabbar, Aamir Shafi, Hari Subramoni, and Dhabaleswar K. Panda. 2023. Implementing and Optimizing a GPU-aware MPI Library for Intel GPUs: Early Experiences. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 131–140. https://doi.org/10.1109/CCGrid57682.2023.00022

[5] Intel. 2024. Intel Data Center GPU Max 1100. https://www.intel.com/content/www/us/en/products/sku/232876/intel-data-center-gpu-max-1100/specifications.html.

[6] Intel. 2024. Level-Zero. https://github.com/oneapi-src/level-zero.

[7] Kawthar Shafie Khorassani, Ching-Hsiang Chu, Hari Subramoni, and Dhabaleswar K. Panda. 2018. Performance Evaluation of MPI Libraries on GPU-enabled OpenPOWER Architectures: Early Experiences. In *International Workshop on OpenPOWER for HPC (IWOPH 19) at the 2019 ISC High Performance Conference*.

[8] Nikolay Kondratyuk, Vsevolod Nikolskiy, Daniil Pavlov, and Vladimir Stegailov. 2021. GPU-accelerated molecular dynamics: State-of-art software performance and porting from Nvidia CUDA to AMD HIP. *The International Journal of High Performance Computing Applications* 35, 4 (2021), 312–324.

[9] Goutham Kalikrishna Reddy Kuncham, Rahul Vaidya, and Mahesh Barve. 2021. Performance Study of GPU applications using SYCL and CUDA on Tesla V100 GPU. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[10] Evgeny Kuznetsov and Vladimir Stegailov. 2019. Porting CUDA-Based Molecular Dynamics Algorithms to AMD ROCm Platform Using HIP Framework: Performance Analysis. In *Supercomputing*, Vladimir Voevodin and Sergey Sobolev (Eds.). Springer International Publishing, Cham, 121–130.

[11] NVIDIA. 2024. CUDA Interprocess Communication. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[12] Sreeram Potluri, Khaled Hamidouche, Akshay Venkatesh, Devendar Bureddy, and Dhabaleswar K Panda. 2013. Efficient Inter-node MPI Communication Using GPUDirect RDMA for InfiniBand Clusters With NVIDIA GPUs. In *Parallel Processing (ICPP), 2013 42nd International Conference on*. IEEE, 80–89.

[13] István Z. Reguly. 2019. Performance Portability of Multi-Material Kernels. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 26–35. https://doi.org/10.1109/P3HPC49587.2019.00008

[14] Kawthar Shafie Khorassani, Jahanzeb Hashmi, Ching-Hsiang Chu, Chen-Chun Chen, Hari Subramoni, and Dhabaleswar K. Panda. 2021. Designing a ROCm-Aware MPI Library for AMD GPUs: Early Experiences. In *High Performance Computing: 36th International Conference, ISC High Performance 2021, Virtual Event, June 24 – July 2, 2021, Proceedings*. Springer-Verlag, 118–136.

[15] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. K. Panda. 2014. GPU-Aware MPI on RDMA-Enabled Clusters: Design, Implementation and Evaluation. *IEEE Transactions on Parallel and Distributed Systems* 25, 10 (Oct 2014), 2595–2605. https://doi.org/10.1109/TPDS.2013.222

[16] Hao Wang, Sreeram Potluri, Miao Luo, Ashish Kumar Singh, Sayantan Sur, and Dhabaleswar K. Panda. 2011. MVAPICH2-GPU: Optimized GPU to GPU Communication for InfiniBand Clusters. *Comput. Sci.* (2011), 257–266.

[17] Yujia Zhai, Mohannad Ibrahim, Yiqin Qiu, Fabian Boemer, Zizhong Chen, Alexey Titov, and Alexander Lyashevsky. 2022. Accelerating encrypted computing on intel gpus. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 705–716.