Design and Implementation of MPI-Native GPU-Initiated MPI Partitioned Communication

Yıltan Hassan Temuçin*, Whit Schonbein†, Scott Levy†, Amirhossein Sojoodi*, Ryan E. Grant*, Ahmad Afsahi*
*ECE Department, Queen's University, Kingston, ON, Canada
†Sandia National Laboratories, Albuquerque, New Mexico, USA
*{yiltan.temucin, amir.sojoodi, ryan.grant, ahmad.afsahi}@queensu.ca
†{wwschon, sllevy}@sandia.gov

Abstract—**Graphics Processing Units have become the dominant type of accelerators for high-performance computing and artificial intelligence. To support these systems, new communication libraries have emerged, such as NCCL, RCCL, and NVSHMEM, providing stream-based semantics and GPU-Initiated Communication. Some of the best performing communication libraries are unfortunately vendor-specific, and may use load-store semantics that have been traditionally underused in the application community. Moreover, the Message Passing Interface (MPI) has yet to define explicit GPU support mechanisms, making it difficult to deploy the message-passing communication model efficiently on GPU-based systems. However, MPI-4.0 introduced MPI Partitioned Point-to-Point communication, which facilitates hybrid-programming models. For example, Partitioned Communication is designed to allow GPUs to trigger data movement through a persistent intra- or inter-node channel. In this work, we extend MPI Partitioned to provide Intra-Kernel GPU-Initiated Communication and Partitioned Collectives, augmenting MPI with techniques used in vendor specific libraries. We evaluate our designs on a NVIDIA GH200 Grace Hopper Superchip testbed, to understand the benefits of GPU-Initiated communication on NVLink and InfiniBand networks. We assess the benefits at the application layer using a Jacobi solver and Partitioned Allreduce with Deep Learning Kernels.**

I. Introduction

Graphics Processing Units (GPUs) have become a dominant form of accelerator for high-performance computing (HPC) systems. From the June 2024 Top500 list, 9 of the top 10 supercomputers in the world use GPU-based platforms from vendors such as AMD, NVIDIA, and Intel [1]. As a result, applications in many domains such as Molecular Dynamics [2], Drug Discovery [3], and Deep Learning (DL) [4] have been adapted to use GPUs.

The Message Passing Interface (MPI) [5] is the *de-facto* standard for programming HPC machines. Multiple implementations of MPI exist, including MPICH [6], MVAPICH2 [7] and Open MPI [8]. MPI supports Point-to-Point, Partitioned Point-to-Point, global collectives, neighborhood collectives, and remote memory access (RMA) communication operations. Mirroring the advances in system design, MPI implementations have adapted to be GPU-Aware [9], [10]. However, this has yet to be standardized.

MPI Partitioned Point-to-Point Communication is a new addition to the MPI-4.0 Standard added in June 2021 [5] to better support multi-threaded and heterogeneous systems. In this new model, the send and receive buffers of a Point-to-Point communication are partitioned into distinct chunks which can be addressable by individual actors that are marked ready as they become available. These buffers are persistent and can be repeatedly used within an application's life cycle.

Much of the existing literature on MPI Partitioned has been on multi-threaded CPU workloads [10]–[16]. However, with GPUs becoming the dominant form of accelerators for large HPC systems, it is important that MPI Partitioned and MPI as a whole to adapt to current trends. The standardization of GPU bindings for MPI Partitioned is an active topic of discussion within the MPI Forum Hybrid Working Group, but no consensus has yet been reached [17]. For example, MPI Partitioned could allow for GPU thread, warp, or blocks to mark data as ready, but which one is most performant is an open question.

The goal of allowing for GPU-Initatied communication is not limited to MPI Partitioned, as there is interest in Stream synchronous communication within MPI [18]. Moreover, GPU-Initiated has garnered interest by other programming models and communication API such as NVSHMEM. In this paper, we address these issues by exploring MPI Partitioned optimizations on GPUs while still considering designs that could be applied elsewhere. Specifically, we make the following contributions:

1) We provide the first MPI-Native implementation of MPI Partitioned on GPUs and discuss the challenges with designing a portable implementation;
2) We present the first MPI Partitioned Collective schedule design and how this can be utilized by GPUs;
3) We investigate whether data should be signaled as ready to send by a GPU at the level of thread, warp, or block, and discuss the benefits of partition aggregation on GPUs;
4) And we evaluate the overheads of introducing several additional API calls designed to facilitate GPU-initiated communication under MPI Partitioned.

II. Background

A. Compute Unified Device Architecture (CUDA)

CUDA is a general-purpose parallel programming model that allows users to take advantage of NVIDIA GPU parallel compute engines. The CUDA environment allows users to program in C++ but CUDA can be interfaced with other languages such as C or FORTRAN. CUDA helps solve some of the challenges of

† Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC (NTESS), a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration (DOE/NNSA) under contract DE-NA0003525. This written work is authored by an employee of NTESS. The employee, not NTESS, owns the right, title and interest in and to the written work and is responsible for its contents. Any subjective views or opinions that might be expressed in the written work do not necessarily represent the views of the U.S. Government. The publisher acknowledges that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this written work or allow others to do so, for U.S. Government purposes. The DOE will provide public access to results of federally sponsored research in accordance with the DOE Public Access Plan.

transparently scaling applications in parallel environments. Other GPU vendors such as AMD and Intel have their own equivalent programming models and runtimes. However, we will focus on NVIDIA in this paper but the general ideas are applicable to other platforms.

One of the main components of CUDA programming are kernels. These are similar to traditional functions in C/C++ but they can be executed in parallel using many CUDA threads. Each CUDA thread executes a kernel using its own thread ID. CUDA applications concurrently transfer data and execute kernels via the concept of *streams*. A stream can be thought as a First-In First-Out (FIFO) queue of operations that will be executed in the order they are placed in the queue. All streams or subgroups of streams on a single device can be synchronized.

B. MPI Partitioned

1) MPI Partitioned Point-to-Point: MPI Partitioned Point-to-Point Communication extends traditional MPI Point-to-Point semantics by providing better cohesion with hybrid programming models [5]. An application which uses MPI Partitioned first initializes communication between endpoints using `MPI_Psend_init` and `MPI_Precv_init`. This sets up a communication channel between two processes based on communicator, rank, tag, and the order in which they are posted. The user also specifies how many partitions a buffer is split into. Once the application is ready to communicate it calls `MPI_Start` to notify the library.

MPI requires that a single thread execute the preceding calls. Once `MPI_Start` executes, an application can enter a parallel region which could be in the form of an OpenMP block, POSIX thread, or a GPU kernel. The sender marks data as ready in the parallel region by using `MPI_Pready`. For example, each thread can mark a partition as ready, or multiple threads contributing to the same partition can synchronize with one thread that marks the partition ready. Marking data as ready does not necessarily mean the data is sent at that moment; the timing of when the data is actually sent is determined by the MPI runtime. The receiver can, but is not required to, call `MPI_Parrived` in a parallel region to check if a partition has arrived.

Finally, in a single-threaded region, an application developer can complete a partitioned transfer by calling `MPI_Test` or `MPI_Wait` on the receiving process to check or wait for the arrived data. Because MPI Partitioned is persistent, an application developer could start a new transmission using the same buffer simply by calling `MPI_Start` on the existing request and then calling `MPI_Pready` on each partition as before. The initialization calls are only called once during the lifetime of the send and receive buffers.

2) GPU Support for MPI Partitioned: Currently, MPI + CUDA programs require application developers to wait for a kernel to complete before issuing a communication routine such as `MPI_Send`. This is illustrated in Listing 1, where MPI communication is initiated only after the kernel is executed and the stream is synchronized. As we will see in Section III, `cudaStreamSynchronize` is expensive and leaves the host waiting on a CUDA kernel and unable to communicate. Moreover, when `MPI_Send` is called the GPU is idle, unless a user uses advanced overlapping techniques.

Listing 1: Host Pseudo Code for the Traditional MPI + CUDA Model

```
kernel_A<<<stream>>>(sbuf);
cudaStreamSynchronize(stream);
MPI_Send(sbuf);
```

There has been significant discussion on accelerator support in the MPI Forum Hybrid Working Group. Based in part on this discussion, NVIDIA has developed an MPI Acccelerator Extensions (MPI-ACX) prototype [19] that adds device bindings to GPU Partitioned. Similar to MPI-ACX, we propose `MPIX_Pready`, a GPU version of the existing `MPI_Pready` function. This allows MPI to be called directly within a GPU kernel and stream synchronization is not required to guarantee communication has been completed.

However, even with a device-specific `MPIX_Pready`, there exist challenges. In particular, according to the MPI standard, initialization and start calls are non-blocking. Consequently, there is no guarantee a receiver is ready to receive data. One solution is to block on `Pready` until the receiver is initialized [10], but this will stall the kernel, requires the GPU to handle progression, and potentially increases the chance of deadlock [20]. An additional challenge is that the sender could call `MPI_Start` a second time to reuse a MPI Partitioned channel, but if the receiver is not ready this would result in the receiver buffer being overwritten.

Currently, to address this issue there exists a proposal in the MPI Forum Hybrid Working Group for an MPI extension: `MPIX_Pbuf_prepare` [21]. The purpose of this proposed API call is to provide a guarantee to the sender that the remote buffer is ready to receive. The role of `MPIX_Pbuf_prepare` is shown in Figure 1, where it is used to synchronize the two processes. This prevents both of the issues outlined in the previous paragraph.

An additional requirement for GPU-Initiated MPI Partitioned is for `MPIX_Pready` to be callable from within a GPU kernel [17]. One method for this is to define `MPIX_Device` as shown in Listing 2. This allows MPI to have a generic execution space specifier that the preprocessor would match to the vendor's implementation. For example, with AMD and NVIDIA GPUs, `MPIX_Device` would be replaced with `__device__`. This new `MPIX_Pready` call could mark data as ready or transfer data directly within a GPU kernel.

For `MPIX_Pready` to mark data as ready, it is required that the `MPI_Request` object be accessible by a GPU. One solution is that an MPI library would allocate the required

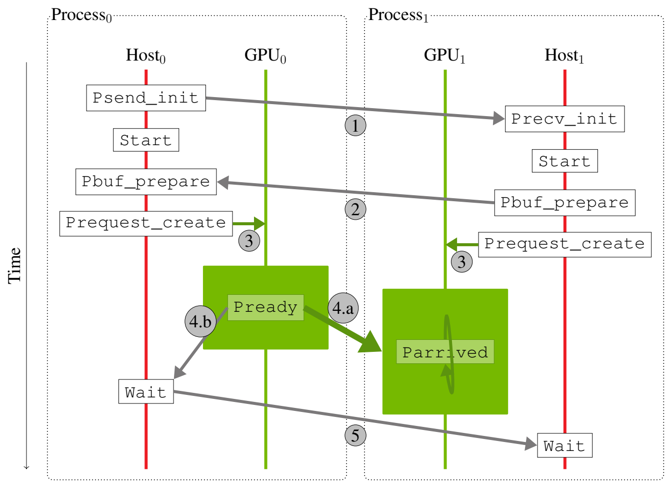

Fig. 1: A High Level Sequence Diagram Presenting GPU-Initiated MPI Partitioned

Listing 2: Device Pseudo Code for the Proposed GPU-Initiated MPI Partitioned Model [19]

```
MPIX_Device int MPIX_Pready(int partition,
                            MPIX_Prequest preq);

__global__ int kernel_B(MPIX_Prequest preq,
                        double *sbuf)
{
  int idx = threadIdx.x + blockDim.x * threadIdx.y;
  /* Do Work */
  MPIX_Pready(idx, preq);
}

__host__ int host_function(MPI_Request req,
                           double *sbuf)
{
  MPI_Start(req)
  MPIX_Pbuf_Prepare(req);

  if (first_iteration)
  {
    MPIX_Prequest_create(preq, req);
  }

  kernel_B<<<stream>>>(preq, sbuf);
  /* Do work on host */
  MPI_Wait(req);
}
```

MPI_Request object and its substructures with unified memory. However, this could pose challenges if a vendor does not provide unified memory or if it is expensive to use on a specific platform. Moreover, GPUs are generally poor at pointer chasing. Ideally, a request would contain only the necessary information for a GPU to conduct its duties. This could be achieved by defining MPIX_Prequest which would be a device specific request object for MPI Partitioned [22]. The API call MPIX_Prequest_create would take an MPI_Request object as an input and output a MPIX_Prequest object. MPIX_Prequest_create would have a corresponding MPIX_Prequest_free to free the memory associated with the MPIX_Prequest object.

3) MPI Partitioned Collectives: Collective communication is the natural extension to MPI Partitioned Point-to-Point Communication as it helps simplify the movement of data between groups of processes [23]. MPI Partitioned Collectives follow the same general control flow as Point-to-Point but has different initialization functions, e.g. MPIX_Pbcast_init, MPIX_Pallreduce_init, etc. These correspond to the equivalent MPI_Bcast and MPI_Allreduce communication patterns. During initialization time, as we have the message size, communicator size, and partition count, we can initialize the resource required to execute a specific collective algorithm. The behavior of MPIX_Pbuf_prepare also changes slightly as we now synchronize the processes associated with the collective rather than just two ranks.

C. Unified Communication X (UCX)

UCX is a communication framework that abstracts many communication primitives to effectively utilize a variety of hardware [24]. The UCP API of UCX implements high-level protocols that are used by other communication libraries such as MPI. UCP supports Remote Memory Access (RMA), active messages, and tag-matching operations, among others. In this paper, we use the UCP RMA API for communication.

To use the UCP API, we must create a UCP **Worker** which represents a communication context that encapsulates communication resource and a progression engine. A Worker object abstracts details regarding the hardware including the network interface, network port, etc. A Worker also encapsulates one or many **Endpoints**, which are used to address a remote Worker (i.e., the target of an initiator). UCP communication routines, such as ucp_put_nbx, use the endpoint address to put data from source to the correct target. Details on how UCX resources are mapped to MPI Partitioned will be further discussed in Section IV-A.

Currently, UCX supports buffers in GPU Global memory, however, it lacks any support for GPU-Initiated communication. There is currently no method to initiate communication from a GPU kernel or to setup a channel and trigger a data transfer. The lack of support is not limited to UCX; libfabric defines a FI_XPU_TRIGGER flag but it is not implemented. A recent survey paper discusses these issues in greater depth [18]. These limitations result in difficulty in providing a GPU-Initiated MPI library. We address this in Section IV-A by proposing designs for intra- and inter-node data transfers.

III. Motivation

It is important for users to understand the costs involved in CUDA-based applications. Figure 2 shows the cost of cudaStreamSynchronize, as well as the cost of launching a simple vector addition CUDA kernel and synchronization (*see* section V for system details). The cost of cudaStreamSynchronize is consistently $7.8 \pm 0.1\mu s$ regardless of kernel size. For smaller kernels (grid sizes up to 256) the synchronization cost is anywhere between 71.6-78.9% of the total time to execute a kernel. This is a significant overhead to endure when sending data after a kernel has completed. Synchronization is less impactful for larger kernels, for example, a kernel with 128K grids, only 0.8% of its total execution time is spent synchronizing. However, this can also be thought of as the CPU being idle for 99.2% of the time the kernel is executing. These lost CPU cycles or computation/communication overlap potential are shown in the gray hashed area in Figure 2. It can cost anywhere between $2.0\mu s$ and $933.4\mu s$ with the kernel sizes we evaluated. These lost resources could be better utilized in CPU compute, progressing communication, or even I/O to prepare data for the GPU. Therefore, it is highly desirable to avoid calling cudaStreamSynchronize during an application's execution.

Communication libraries providing GPU support are growing evermore important with the growth in the popularity of GPU systems. This has resulted in numerous GPU communication

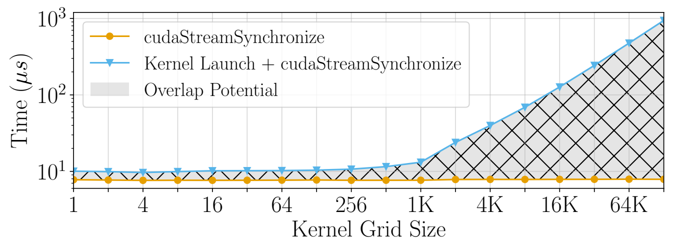

Fig. 2: The cost of cudaStreamSynchronize and the cost of a kernel launch and synchronization for different grid sizes with block size of 1024. The kernel is computing a vector addition $C = A + B$

libraries providing stream-based semantics and GPU-Initiated communication such as RCCL, NCCL, NVSHMEM, etc. [25]–[27]. However, it is important to have a vendor-neutral high performance communication library to ease widespread compatibility of important applications and to support the large number of existing applications currently in production use. In MPI-4.0, MPI Partitioned Point-to-Point communication was introduced to allow support with hybrid programming models. As discussed in Section II-B2, GPU-Initiated MPI Partitioned could be a viable programming model for GPUs because it allows for better overlap between host and device code and avoids the costs associated with calling `cudaStreamSynchronize`. This model allows an application to reduce synchronization overheads for small kernels and provide overlap for large kernels.

IV. Design

Our design consists of a UCX-based MPI Partitioned Point-to-Point library which provides GPU-Initiated communication as discussed in Section IV-A. Partitioned Collectives are described in Section IV-B. Our Partitioned Collectives are built upon our Point-to-Point design.

A. UCX-Based MPI Partitioned Point-to-Point

The Modular Component Architecture (MCA) is used by Open MPI to provide performance and compatibility to a wide variety of networks and memory types [28]. Currently, both RMA and Point-to-Point communication have a UCX component, however, Partitioned Communication lacks a UCX component to optimize this interface. In this paper, we propose a new Partitioned UCX component for Open MPI with extensions for GPU-Initiated communication.

This will reflect the high-level design in Figure 1 where the `MPI_{Psend, Precv}_init` are used to begin initializing our communication resources. Then `MPI_Start` is used to notify the MPI library that we are beginning our communication epoch. `MPIX_Pbuf_Prepare` is used to guarantee our communication resources are initialized. On our first communication epoch `MPIX_Prequest_create` is called to move communication resource to device memory. Then our GPU kernel is launched where `MPIX_Pready` is called to mark our data as ready. Finally, we wait for our communication to complete with `MPI_Wait`. These steps will be explained in further detail below.

1) MPI_{Psend, Precv}_init: On the first call into the MPI Partitioned API, these initialization routines create a UCP context. Each process also creates its own UCP worker and obtains a worker address. The sender pre-populates the desired parameters (`ucp_request_param_t`) for the `ucp_put_nbx` operation as we know the data size, the number of partitions, and our destination. The sender also packs the relevant information such as the tag, rank, and communicator which is used for matching as well as the number of partitions, data size, worker address, etc. into a 'setup_t' object that is sent to the receiver in a non-blocking fashion. The receiver posts a corresponding receive operation. This is shown with ① in Figure 1.

2) MPI_Start, MPIX_Pbuf_prepare: `MPI_Start` simply marks the requests as pending and sets the internal flags to their default state. Thus far, there is no guarantee that progress has occurred as per the MPI standard. The initial call to `MPIX_Pbuf_prepare` is required to guarantee that the receiver is ready. In the Progression Engine, the receiver checks for the 'setup_t' object, once it is received it unpacks the data. Then we register the receive buffer and the internal flags used for the partition status using `ucp_mem_map` and `ucp_rkey_pack`. During the data transfer phase, the sender will write to the partition status flags to notify the receiver that communication for that partition has completed. Then it creates a 'setup_t' object in response with the same parameters as described in the initialization routines as well as the remote key and remote address. This provides the necessary information for the sender to be able to use RMA operations. Simultaneously, the sender waits for the setup object response. Using the response, it creates the relevant endpoints if they do not already exist and unpacks the memory keys. After these steps are completed, the sender can put data into the receiver. The subsequent calls to `MPIX_Pbuf_prepare` are much simpler as the receiver sends a 'ready-to-receive' signal and the sender waits for that signal. No additional setup information transferred or initialization is conducted. These steps are labeled as ② in Figure 1.

3) MPIX_Prequest_{create, free}: These API calls are required for specifically GPU-Initiated MPI Partitioned. We allocate an `MPIX_Prequest` object in GPU global memory that contains the minimal amount of information required by the device for communication. This information includes the type of copy mechanism (intra-node Kernel Copy, intra- and inter-node Progression Engine copy), as well as a threshold parameter specifying the number of CUDA threads that will be aggregated into a single data transfer. The request also contains a list of counters which are incremented until the threshold value is reached. The threshold and counters enable the various thread/warp/block partition aggregation schemes explored below. For progression, a set of flags is created in pinned host memory which are accessible by the device. These parameters are first created in a host buffer then copied to the GPU once populated as shown by ③ in Figure 1. `MPIX_Prequest_free` frees the memory location in GPU global memory and frees the pinned host memory, if applicable.

4) MPI_Pready, MPI_Parrived, MPIX_Pready, MPIX_Parrived: These API calls have host and device bindings. Here we first discuss the former and then describe how the device bindings are built upon them. The host bindings are used by the Progression Engine internally for the partitioned collectives as shown in Line 25 of Algorithm 2.

The host's call to `MPI_Pready` executes `ucp_put_nbx` to send the data associated with that partition. This operation uses the `ucp_request_param_t` previously provided by `MPI_Psend_init`. Attached to the callback of the put operation is another `ucp_put_nbx` call which marks a partition as received on the receiver side. This is required so that `MPI_Parrived` can provide fine-grained information on partition arrival. This additional control signal is required because UCX does not currently provide a put operation that generates a receive side completion (cf., `IBV_WR_RDMA_WRITE_WITH_IMM` [10]).

For the device bindings, we have two copy mechanisms: the Progression Engine and Kernel Copy approaches. For the **Progression Engine** approach, a CUDA thread updates a flag in host memory to notify the Progression Engine that an `MPIX_Pready` call is pending. Upon detecting this notification, the MPI Progression Engine issues an `MPI_Pready` on the host, as described in the previous paragraph. This mechanism is similar to that proposed in MPI-ACX [19].

The number of threads within a CUDA kernel can be large (e.g., a single block can have up to 1024 threads), and having a large number of threads write into host memory can be quite costly. Therefore, we implement aggregation at the thread, warp, and block-level with specific bindings for the Progression Engine approach to understand if there are any benefits of aggregating partitions on GPUs. To evaluate thread-level bindings, we created `MPIX_Pready_thread` where each thread updates flags in host memory. This provides a baseline as well as allowing us to understand how the approach used in MPI-ACX [19] applies in this context. For our evaluation of warp-level bindings, we implemented `MPIX_Pready_warp` which uses `__syncwarp()` to ensure that when all warp-level memory operations have been completed, the 0^{th} thread writes the completion notification into host memory. We use a similar approach at the block-level with `MPIX_Pready_block`, which uses `__syncthreads()` to coordinate within a block. In addition to block-level bindings, we have counters in GPU global memory that we atomically increment until a threshold is hit, before writing into host memory. This allows the aggregation of multiple blocks. As stated earlier, these counters are created in `MPIX_Prequest_create`.

For the **Kernel Copy** approach, data is transferred directly from source to destination via NVLink within the kernel without involving the host by using an assignment statement. This is shown with 4.a in Figure 1. To execute this transfer, we require an address mapped to the physical address of the remote GPU. Currently, UCX exposes this feature only to CPUs via the `ucp_rkey_ptr` API call. To expose this feature for GPUs we modify UCX's IPC transport layer (specifically, `uct_cuda_ipc_rkey_ptr`), upon which `ucp_rkey_ptr` relies. We use `cuIpcOpenMemHandle` to open a memory handle from the remote process to expose the mapped address with the appropriate memory offsets. The same could be implemented for AMD GPUs using equivalent ROCm API calls. The target memory address obtained through `ucp_rkey_ptr` is placed in the `MPIX_Prequest` object during `MPIX_Prequest_create`.

We must also send a control signal to the host 4.b so that it can issue a completion to the receiver 5. After the kernel has completed its transfer we increment the counters in GPU global memory until all threads have transferred data. Then we mark partitions as ready and fall back to the host `MPIX_Pready` for our completion signal.

For `MPI_Parrived`'s host binding we simply poll on the receive-side completion flags. The device version polls a flag in GPU global memory as the cost of accessing global memory is much lower than host memory. However, as our receive-side completion flags are always populated in host memory, we issue a memory copy to the device in `MPI_Wait` as partitions arrive.

5) MPI_Wait: Finally, when `MPI_Wait` is called, the sender progresses any outstanding puts to ensure the callbacks are sent, and the receiver counts the arrived flags until it matches the number of partitions. Currently we only have a single thread which progresses partitions.

B. MPI Partitioned Collective

Partitioned Collectives are implemented using the Point-to-Point library described in the previous section. In this section, a **user partition** is a partition that a user will see when using a collective. A **transport partition** is a partition our collective uses with regards to our Point-to-Point layer. The two can differ due to the aggregation of multiple user partitions into a single transport partition, for example.

1) MPIX_P<collective>_init: Similar to the Point-to-Point API, the current MPI Partitioned Collective proposals have an initialization function for each collective (e.g., `MPI_Bcast`, `MPI_Allreduce`, etc.). In this paper, we generalize these collective initialization calls and refer to them as `MPIX_P<collective>_init`. Generalization of Partitioned Collectives are incredibly important to consider as the current proposals have at least 21 collectives that must be implemented by MPI libraries [23]. As this is quite burdensome for MPI developers, we take inspiration from MPI Neighborhood Collectives and create a schedule for arbitrary communication patterns [29]. Although our schedule is designed to be generic, we will focus on a partitioned allreduce operation since we investigate DL kernels in Section VI-D2.

During initialization, we allocate a request object, construct a schedule ($\mathbb{S}$) attached to the request, and add the request to a queue used to track active requests to progress. The schedule comprises a series of steps $\mathbb{S} = \{S_0, \cdots, S_k\}$ that are executed. While a single schedule is created, each partition independently executes that schedule and stores its current state.

Each step is a tuple $S_i = (I, R, \oplus, O, A)$ including a set of incoming neighbors $I = \{I_0, \cdots, I_n\}$, the `MPI_Pready` offset R, an operation $\oplus$ that must be executed during that step, outgoing neighbors $O = \{O_0, \cdots, O_n\}$, and the `MPI_Parrived` offset A. The $\oplus$ corresponds to the `MPI_Op` associated with a collective or a NOP. For example, an `MPIX_Pbcast` using a binary-tree algorithm will consist of only NOPs, but an `MPI_Allreduce` using a Ring-based reduce-scatter-allgather algorithm will consist of an `MPI_Op` for the first $P - 1$ steps and then a NOP for the remaining $P - 1$ steps.

Algorithm 1 shows the schedule creation for a Ring-based reduce-scatter-allgather algorithm. For a given rank r, I is the rank preceding r in the Ring, and O is the rank following r. During the schedule construction, r calls `MPI_Psend_init`, `MPI_Precv_init` on its outgoing and incoming neighbors, respectively. Then in lines four and five, R and A offsets are

Algorithm 1: `MPIX_Pallreduce_init` schedule creation for a Ring-Based RSA algorithm

```
1  for $i \leftarrow 0$ to $2(P-1)$ do
2      $I \leftarrow (\text{rank} - 1) \mod P$
3      $O \leftarrow (\text{rank} + 1) \mod P$
4      $R \leftarrow (\text{rank} + 2P - i) \mod P$
5      $A \leftarrow (\text{rank} + 2P - i - 1) \mod P$
6      if $i < (P-1)$ then $\oplus \leftarrow MPI\_Op$ ;
7      else $\oplus \leftarrow NOP$ ;
8      $S_i \leftarrow (I, R, \oplus, O, A)$
9      $\mathbb{S} \leftarrow S_i$
10 end
```

calculated. Creating a different offset for each step allows us to pipeline the Ring algorithm using partitions. In Lines 6 and 7, $\oplus$ is set to true or false based on whether we are in the reduce scatter or allgather portion of the collective.

2) MPI_Pready, MPI_Parrived: We first calculate the transport partition index using the equation: transport partition $=$ (user partition $*$ user partition size) $+ R$. The R value is associated with the step in the schedule as seen in Algorithm 1. Using the calculated transport partition `MPI_Pready` is called as described in Section IV-A4. For `MPI_Parrived`, we simply read a flag in memory to see if the allreduce has been completed.

3) MPI_Wait: Much of the Partitioned Collective is executed in the Progression Engine as there are no progress guarantees for calls other than `MPIX_Pbuf_prepare`. Algorithm 2 shows how partitions are progressed in `MPI_Wait`.. The algorithm iterates over the number of partitions in the collective, as each partition has its own state. This allows for each partition to progress to the next step of the collective independently. We first check that we have not exceeded the last step in our collective in Line 4 to minimize our progression overhead. Then we check if the number of partitions that have arrived is the equal to the number of incoming neighbors in Line 5. If this condition is satisfied it would signify that all relevant data has arrived in the current step of the algorithm for this partition's state. If the data is incomplete, we iterate over our incoming neighbors in Lines 6-12. In this loop we individually check if the partition has arrived for that neighbor, and reduce that data if applicable. The algorithm is presented at a high level, and the implementation details are omitted. In our particular implementation, we ensure that the reduce operation is only executed once for each incoming neighbor. Then in Line 14, we check if the number of partitions that have arrived is equal to the number of incoming neighbors and if the number of partitions marked as ready is equal to the number of outgoing neighbors. If evaluated to be true, the current step in the algorithm is complete. Therefore, we would move to the next step (S_{i+1}) in our Partitioned Collective schedule, and reset our counters to zero. In Line 21, we check if any `MPI_Pready` calls have been made. We verify that the state is not S_0 since the first `MPI_Pready` should be called by the application programmer. We also check that we are k^{th} step (reaching the maximum number of steps in our algorithm) to ensure we do not transfer any additional data unnecessarily.

Algorithm 2: `MPI_Wait` Progression of a Partitioned Collective Schedule

```
1  for $part \leftarrow 0$ to num_partitions do
2      state = states[part]
3      $S \leftarrow$ state.step
4      if $S > S_k$ then continue ;
5      if state.parrived_complete $\neq |I|$ then
6          for $I_x \in I$ do
7              MPI_Parrived(flag)
8              if flag = true then
9                  state.parrived_complete++
10                 if $\oplus \neq$ NOP then reduceData() ;
11             end
12         end
13     end
14     if state.parrived_complete $= |I|$ and
15         state.pready_complete $= |O|$
16      then
17         $S \leftarrow S_{(i+1)}$
18         state.parrived_complete $\leftarrow 0$
19         state.pready_complete $\leftarrow 0$
20     end
21     if $S \neq S_0$ and $S! = S_k$ and
22         state.pready_complete $= 0$
23      then
24         for $O_x \in O$ do
25             MPI_Pready(...)
26             state.pready_complete++
27         end
28     end
29 end
```

V. Experimental Platform

Our evaluation is based on a two-node NVIDIA GH200 Grace Hopper Superchip testbed. Each NVIDIA Grace CPU has 72 ARM Neoverse V2 CPU cores with 120GB of LPDDR5X memory [30]. This is combined with an NVIDIA Hopper GPU that has 96GB of HBM3 memory. These two elements are connected via the NVIDIA NVLink-C2C, which provides a 900GB/s total bandwidth chip-to-chip interconnect. Each node has four NVIDIA GH200 Grace Hopper Superchips. The NVIDIA Hopper GPUs are connected with 18 NVLink 4 links per device, resulting in an aggregate bandwidth of 900GB/s. Between each GPU pair, there are 6 NVLink connections resulting in a total uni-directional bandwidth of 150GB/s to each neighbor. Each node is composed of four Mellanox ConnectX-7 network cards (400Gbit). The software environment is NVHPC version 23.11. The GNU/Linux distribution is Ubuntu 22.04.2 LTS, with GCC version 12.3.0, UCX master branch (commit bc85b70e6, ca. March 19th, 2024), and Open MPI version v5.0.1rc1.

The GH200 platform differs from many other GPU systems in production today. However, the conclusions drawn from Section VI are still applicable to most other systems. For example, most NVIDIA GPU-based systems have some form of NVLink capability so our comparison for different copy methods would easily apply to different contexts. The same would be true for an AMD system as they have their own intra-node network, Infinity Fabric.

VI. Experimental Results

In our tests, each CUDA thread works on 8 bytes of data. For example, for a kernel with 1024 CUDA threads, where each contributes 8B to an allreduce operation, the total data size is 8KiB. Goodput is used as an evaluation metric as we want to understand the amount of useful work the GPU can do per unit of time. Goodput is defined as the total amount of data being processed divided the total execution time (computation time + communication time). Goodput is a better metric than bandwidth for this situation as it includes the cost of computation and communication, and their overlap, rather than pure network bandwidth which would be limited by hardware. Unless stated otherwise, we use a vector addition $C = A + B$ kernel as our workload for our CUDA Kernels.

We evaluate our design at the Point-to-Point, collective and application-kernel layer. For the send/receive and traditional `MPI_Allreduce` data points we measure the time to execute computation, synchronization, and communication as shown in Listing 1, then we calculate our Goodput. For our partitioned test we measure time to execute the equivalent of `Kernel_B` and `MPI_Wait` in Listing 2, then use this time to calculate Goodput.

A. MPI Partitioned Point-to-Point

1) Device-Side Partition Aggregation: As noted in Section IV, it is an open question whether there are benefits to aggregating thread-level partitions into warp or block-level transport partitions. In Figure 3, we evaluate the different aggregation strategies of `MPIX_Pready` calls (thread-level (no aggregation), warp-level, and block-level) for intra-node GPU-to-GPU communication, from a single thread to the maximum block size for a GH200 (1024 threads). For a single thread, the cost is the same (within error) for all three methods. This is also true for warp-level and block-level aggregation up to 32 threads. In this block size range, we have yet to fully occupy a warp. Above 32 threads, is where we see the discrepancy grow between the warp-level and block-level aggregation of our partitions. For a fully occupied block, a block-level `MPIX_Pready` call costs 271.5x less than at the thread-level and 9.4x less than the warp-level call. This is due to the thread-level `MPIX_Pready` requiring 1024 writes to memory and the warp-level requiring 32 writes, compared to a single write for the block-level.

From this test, it is abundantly clear that there is a large performance penalty for the finer-grained `MPIX_Pready_thread` and `MPIX_Pready_warp` calls. That said, we believe that each GPU thread should call `MPIX_Pready` to simplify the programming model and that MPI should aggregate to the block-level internally. We also investigated how to aggregate multiple blocks, and those results showed that a single transport partition was what provided the highest Goodput. Although this was sufficient for this initial test, we will revisit this throughout our evaluation.

2) Comparison with Different Communication Models: In Section IV, we discussed two copy mechanisms for intra-node copies, using a copy kernel and issuing a `ucp_put_nbx` via the MPI Progression Engine. In Figure 4, we evaluate the two copy mechanisms and compare them to the traditional model, for intra-node communication. The maximum uni-directional NVLink bandwidth is provided as a reference for the upper bound of our Goodput value.

Using MPI Partitioned with the Progression Engine approach outperforms traditional for all kernel sizes up to a grid size of

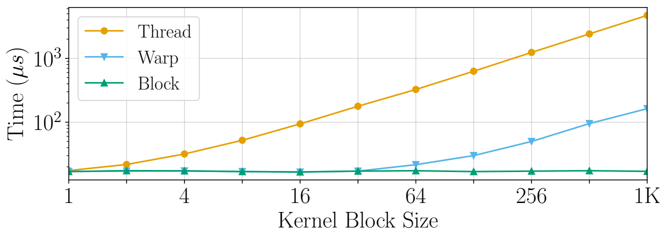

Fig. 3: The Cost of Mapping Partitions to Threads, Warps and Blocks for an Intra-Node Partitioned Point-to-Point Data Transfers

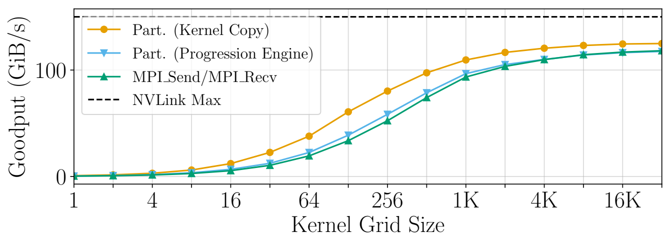

Fig. 4: Results for two GH200 on a single node. Comparison of `MPIX_Pready` copying data within a CUDA Kernel and `MPIX_Pready` writing to a flag in memory where the MPI Progression Engine issues the copy, and using `MPI_Send/Recv`

2K. For larger grid sizes, the speedup is around 1.0x, thus there is no benefit to this approach but there is also no performance penalty. For smaller grid sizes we observe a maximum of 1.28x improvement in Goodput. MPI Partitioned using the Kernel Copy design outperforms both the Progression Engine design as well as the traditional communication model for all kernel sizes. For large kernels, such as 32K grids, we observe a 1.06x speedup compared to the traditional communication model. The impact is much larger for smaller kernels as we observe up to 2.34x improvement in our Goodput.

In Figure 5, we compare GPU-Initiated MPI Partitioned to traditional MPI Send/Recv for inter-node communication. The benefits of GPU-Initiated MPI Partitioned are more significant for inter-node communication. Similar to the intra-node case, the largest performance improvement is seen for smaller kernels. For one grid we observe a 2.80x improvement in Goodput. For the largest grid we evaluated, we obtained a 1.17x higher Goodput. Our performance improvements are better for the inter-node case than the intra-node case as inter-node communication cost is much higher, so the overlapping is more impactful. We found for large kernels that aggregating into two transport partitions provided the best performance.

For both intra- and inter-node scenarios we observed better performance for smaller kernels. This is expected, as in Figure 2 we saw that the synchronization cost of the kernel would be up to 78.9% of the total time to execute a kernel. GPU-Initiated MPI Partitioned avoids this synchronization cost. Similarly this is why the performance improvement for larger kernels is significantly lower as we are not bound by synchronization.

One important consideration is that in the MPI standard, `MPI_Pready` is described as ‘a send-side call that indicates that

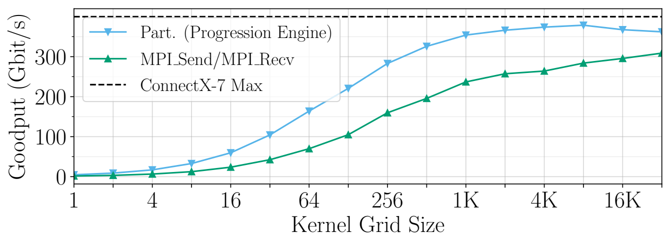

Fig. 5: Results for two GH200 on two nodes. Comparison of `MPIX_Pready` writing to a flag in memory where the MPI Progression Engine issues the copy, and using `MPI_Send/Recv`

a given partition is ready to be transferred' [5]. Our implementation for the inter-node and the intra-node Progression Engine approach adheres to this specification. For the intra-node case where we use a Kernel Copy, we are in a gray area as we conduct the data transfer within `MPIX_Pready`. However, the standard does not explicitly state that this behavior cannot occur. Therefore, if MPI were to adopt `MPIX_Device MPIX_Pready` this should be clarified. This becomes more important with Partitioned Collectives, as discussed in Section VI-B.

B. MPI Partitioned Collectives

In Figure 6 and Figure 7, we evaluate our Partitioned Collective approach as applied to allreduce, and compare it to traditional approaches as well as NCCL. The Ring algorithm is used in all cases, as this algorithm is important in Machine Learning contexts, and our goal is to observe the differences between libraries/interfaces rather than algorithm design. Since the Ring algorithm is used to maximize bandwidth for large messages, we evaluate large kernel grid sizes. For multi-node experiments ranks [0-3] and [4-7] are on the same nodes so that each processes' neighbor is located optimally for all communication libraries.

For both the single-node and multi-node results, we see a significant improvement in time required to execute a kernel and communicate, when comparing `MPI_Allreduce` to the partitioned allreduce. The time required for a partitioned allreduce is multiple orders of magnitudes lower, a result that stems from moving the communication initiation and the computation aspect of our collective to the device.

When comparing the partitioned allreduce to NCCL, NCCL does provide better performance for a single and two nodes. For a kernel with 1K grid size, there is around $226.1\mu s$ between the two libraries when executing the kernel and communicating. This stems from our partitioned allreduce only marking data as ready using `MPIX_Pready`. Specifically, in Line 10 in Algorithm 2, there is an operation to reduce our data. If the buffers for this collective were in host memory, this would not cause any issues. However, as our buffers are in GPU memory we are required to launch an additional kernel for our reduction operation. This reduction is required to be completed before we move on to the next step of the algorithm to have numerically correct data. This results in the requirement of calling `cudaStreamSynchronize` within the collective itself. However, this is still better than the the traditional `MPI_Allreduce` as we remove `cudaStreamSynchronize` application code to provide a better programming model for the user. This would also apply to the partitioned variants of Reduce, ReduceScatter, Scan, and ExScan,/ but this would not be an issue for collectives such as Bcast or Gather which do not have a computation component.

As discussed, the current proposed `MPIX_Device MPIX_Pready` device bindings have some limitations such as only being required to mark data as ready. We suggest that this should be relaxed to allow for computation and communication within the call as that would allow the execution of an entire allreduce operation within a kernel. This is important for the current GH200 and will become even more performance-critical for the GB200 GPUs as over 500 GPUs can be connected via NVLink. Moreover, this is not limited to systems with NVIDIA NVLink, as there are many intra-node interconnects this could apply to such as AMD's Infinity Fabric^{TM} [31], or Cerio's Multi-row scale PCIe networks [32]. An alternative would be to introduce something like collective specific device calls to separate the initialization and execution of a collective, e.g. `MPIX_Device MPI_Pallreduce`. Either of these options should be strongly considered to reduce the performance differential between MPI and NCCL to ensure that MPI stays relevant for decades to come.

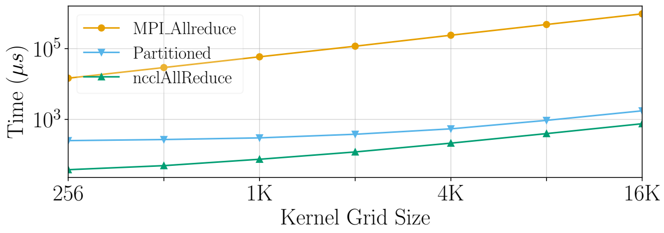

Fig. 6: Allreduce results for four GH200s

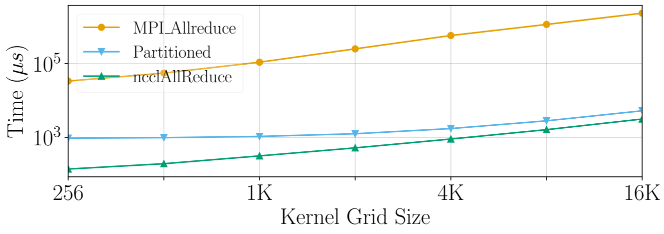

Fig. 7: Allreduce results for eight GH200s

C. Overheads

Communication costs are incredibly important to the viability of GPU-Initiated MPI Partitioned. Table I summarizes overheads associated with those parts of MPI Partitioned that we have not yet covered. We place timer around the API calls listed in the table, and ran the control flow for MPI Partitioned Point-to-Point and Collectives for 100 iterations. Average values of 10 samples with standard deviations are reported in Table I. The API calls fall into three categories: non-blocking initialization, blocking initialization, and synchronization.

Non-blocking initialization includes `MPI_Psend_init`, `MPI_Precv_init`, and `MPIX_Pallreduce_init`. The overheads of these calls are mostly hidden as any required progression is deferred until the first time `MPIX_Pbuf_prepare` is called. `MPI_PSend/Recv_init` has a cost of $17.2\mu s$ and `MPIX_Pallreduce_init` has a cost of $62.3\mu s$. The collective initialization has a higher cost than the Point-to-Point initialization because collective initialization requires multiple Point-to-Point initializations as well as creating the communication schedule.

`MPIX_Prequest_create` is a blocking initialization call that moves the relevant data structures to the device. This

TABLE I: Overheads for Different MPI Calls

MPI Call	Overhead
`MPI_PSend/Recv_init`	$17.2 \pm 10.2\mu s$
`MPIX_Pallreduce_init`	$62.3 \pm 6.2\mu s$
`MPIX_Prequest_create`	$110.7 \pm 37.8\mu s$
`MPIX_Pbuf_prepare`	$193.4\mu s$ first, $3.4 \pm 1.4\mu s$ avg.

is required to be blocking to ensure that when the first `MPIX_Pready` is called its corresponding `MPIX_Prequest` is valid. This call has an overhead of $110.7\mu s$ which is mostly memory registration of flags and a memory copy from host to device.

The prior calls are only called once during an application's life. The call `MPIX_Pbuf_prepare` differs in that it is used to synchronize processes to guarantee remote buffer readiness and is called multiple times during an application's life cycle. The first call will incur a significantly higher overhead than subsequent calls, therefore, two values are given in Table I. The initial call has an overhead of $193.4\mu s$, which includes the overheads of initializing the MCA module and any prior requests. The cost of subsequent calls is $3.4\mu s$, averaged over 100 iterations. This is important to consider as after the initial call only synchronization is performed.

D. Application-Kernel Results

In this section we evaluate two application-kernels, a Jacobi solver and our DL Kernel. The Jacobi solver, evaluates our Point-to-Point design within the context of an application-kernel, and the DL Kernel is designed to benchmark allreduce performance. The data presented in this section differs from prior evaluations as measurements now include the initialization overheads as well as communication, rather than assessing them independently.

1) Jacobi Solver: For this evaluation, we modified the MPI + CUDA example from NVIDIA [33] to use MPI Partitioned Communication. In this implementation, the problem is decomposed across multiple GPUs, and processes on different GPUs engage in a Point-to-Point halo exchange communication pattern while calculating a solution. The problem size must be a multiple of the number of GPUs that are used. For example, on four GPUs it must be a multiple 2x2 and on eight it is a multiple of 4x2. For these experiments, the multiplier is varied from 1 to 32 in powers of 2.

Results for one and two nodes are shown In Figures 8 and 9, respectively. The best performance improvement (in terms of GFLOP/s) on a single node is modest at 1.06x but for our two-node test, we obtain a maximum speedup of 1.30x. MPI Partitioned is also most impactful for smaller kernel sizes and the performance eventually plateaus. Both of these observations are consistent with what we saw in Section VI-A2, where we see better performance for multi-node results as well as for smaller kernels.

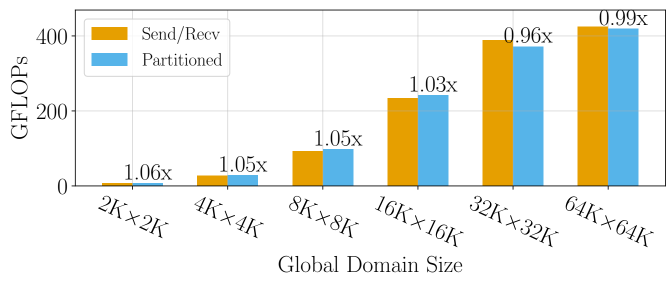

Fig. 8: Jacobi solver GFLOPs results for four GH200

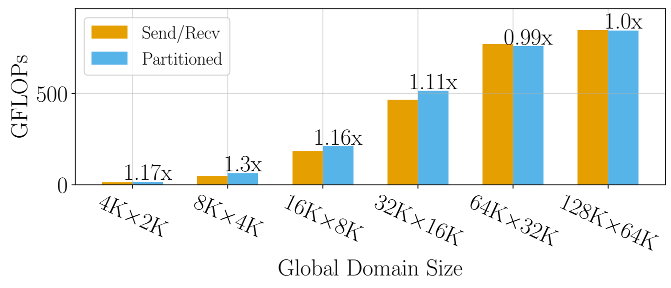

Fig. 9: Jacobi solver GFLOPs results for eight GH200

2) Data Parallelism Deep Learning Kernel: In this section, we evaluate a common kernel and communication pattern used in DL. Gradient descent is an optimization algorithm frequently used in deep learning to minimize a cost function. Roughly speaking, in data parallel training, each GPU receives a copy of the model, and trains on a different subset of the total training data. Periodically, the parameters of the copies are synchronized by exchanging gradients using an allreduce operation.

For these tests, we evaluate a CUDA-based Binary Cross-Entropy kernel from [34] in conjunction with a traditional `MPI_Allreduce` operation, a partitioned allreduce, and a `ncclAllreduce`. For the partitioned allreduce, the cost of `MPI_Start` and `MPIX_Pbuf_prepare` are included in our measurement as this would be present in a training loop. The results are shown in Figure 10 and Figure 11. There is a significant improvement over `MPI_Allreduce` when compared to the Partitioned Collective. However, NCCL still outperforms due to the same reasons as stated in Section VI-B, as the application-kernel is heavily dependent on the collective operation.

VII. Related Work

A. MPI Partitioned

MPI Point-to-Point communication was initially introduced in [11], [12], [35] before being adopted into the MPI 4.0 standard. Temuçin et al. [16] developed a set of MPI Partitioned benchmarks for MPI developers that includes halo exchange patterns. Gillis et al. [36] and Schonbein et al. [37] each provide models for investigating the potential benefits of using MPI Partitioned, allowing for variation in buffer size, number of partitions, etc. The model given in [37] was used by Temuçin et al. [10] to dynamically optimize MPI Partitioned via aggregation.

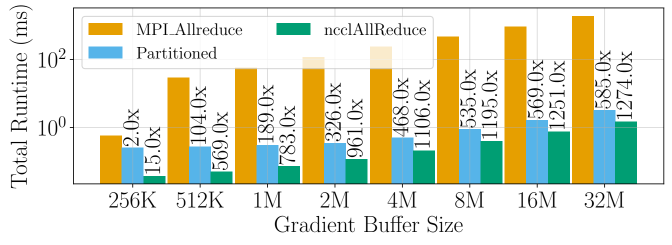

Fig. 10: Deep learning kernel results for four GH200

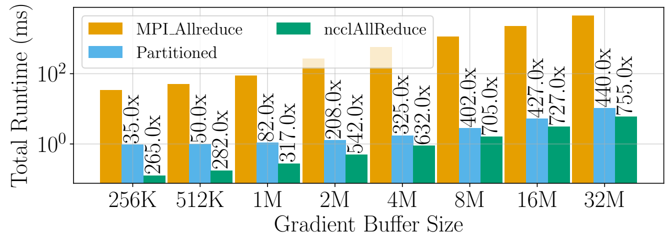

Fig. 11: Deep learning kernel results for eight GH200

In addition, [10] presented the first optimization of MPI Partitioned for specific hardware using InfiniBand Verbs. Dosanjh et al. [14] compare a an implementation of MPI Partitioned using MPI persistent send/receive to one using RMA. They found that an RMA implementation provides some additional performance benefits compared to the persistent implementation. MPI Advance [38] includes a Partitioned Communication library built on persistent communications as part of its collection of optimizing libraries on top of MPI. To our knowledge there is only a single peer reviewed paper on MPI Partitioned Collectives [23]. In that work they proposed the semantics on which MPI Partitioned Collectives should follow.

B. Accelerator-Aware Partitioned Communication

The previously cited works focus on MPI Partitioned using CPU buffers. To our knowledge, the single published work on MPI Partitioned on accelerators (FPGAs) is Christgau et al. [39]. In that work, the authors developed an MPI Partitioned library for FPGAs, observing that there are many limitations on obtaining good performance with current FPGA hardware when compared to MPI Send/Recv. The MPI Accelerator Extensions Prototype by NVIDIA [19] provides a proposed interface for GPU-initiated Partitioned communication, but does not provide any optimizations. The lack of work on accelerators underscores the need for research on MPI Partitioned in this area as most of the world's top supercomputers use GPUs.

C. GPU-Initiated Communication

The concept of GPU-Initiated communication has been around for a long time. Stuart et al. [40] proposed an MPI-like library where communication could be initiated from the GPU using a CPU helper thread to orchestrate the data transfer. Miyoshi et al. [41] proposed making MPI calls directly within GPU kernels. Their method did not require an additional helper thread but rather that the kernel would be paused, communication would occur using the host, then the GPU kernel would resume execution. This implementation requires GPU kernels to be completely synchronous with respect to the host which has obvious disadvantages. They found that their proposal improved GPU programmability with MPI codes but their performance did not scale. Oden et al. [42] implemented InfiniBand verbs on GPUs. This work differed from earlier work insofar as the GPU can control communication by directly accessing the NIC. However, the host is still involved for initialization of the NIC since many system calls are made during that phase. Network resources such as the doorbell register or queue pairs (QPs)s are mapped to the GPU address space to provide direct access to the device. Despite this novel approach, using a host-assisted method performed significantly better for small messages, and performance gains for large messages were fairly small. It was noted that these performance issues were largely due to GPUs being poor with control, the need to minimize PCIe transfers to the NIC, and that NIC hardware needs to be improved. Agostini et al. [43], compared initiating communication on kernel boundaries and within a kernel. They found that controlling communication from inside a kernel provided the lowest ping-pong latency. Initiating communication on the stream outperformed a synchronous model but not as good as the intra-kernel method. For a 2D halo exchange, kernel initiated communication worked better for smaller messages but stream initiated communication outperforms for larger messages. NVSHMEM [44] also provides InfiniBand GPU Direct Async transport. Practically speaking, allowing GPUs to directly control network hardware is still not sufficiently mature. Although it could be implemented on InfiniBand hardware using Direct Verbs and DevX, there is not high-level support for MPI as there is with CPU initiated communication with libraries such as OFI and UCX. This is something that needs to be addressed with the multiple network and GPU vendors.

D. GPU-Initiated MPI

Bridges et al. [18] provide an in-depth summary of past and present proposals for better GPU support for MPI. Venkatesh et al. [45], extended the `MPI_Send/Recv` interface to include a stream parameter where a CUDA stream could be placed. Zhou et al. [46] propose similar MPI extensions for making calls stream-aware. In addition to previously `MPI_Send/Recv` calls working on their streams, they propose adding an `MPIX_Stream` object so that this interface can be accelerator agnostic. They also propose `MPI_Wait` calls that wait for on those streams to synchronize. Alongside those proposals, they also suggest having a stream communicator so that streams can be addressed between different processes. HPE Cassini NICs have the capability to store communication operations in hardware that can then be triggered at a later time. Namashivayam et al. [47], [48] leverage these capabilities to allow GPU streams to trigger MPI Send or MPI Put operations. Our work differs from the other MPI Proposals as we are focused on MPI Partitioned.

VIII. Conclusions

As the prevalence of GPU systems grows in the TOP500, efficient GPU support for MPI becomes critical to maintain performance for HPC and AI applications. Device bindings for MPI Partitioned potentially allow MPI to keep up with specialized libraries such as RCCL or NCCL.

In this paper, we presented the first detailed work on GPU-Initiated MPI Point-to-Point Partitioned Communication. We used UCX to provide GPU-to-GPU intra-node Kernel Copy communication without host control and compare its performance to issuing copies using the MPI Progression Engine. Aggregation of user partitions are explored using counters. We extend our Point-to-Point library implementation to present the first results on GPU Partitioned Collectives. Our Partitioned Collective uses a generic scheduling algorithm designed to be algorithm-independent. The designs are evaluated using micro-benchmarks for Point-to-Point and collective, then finally for a Jacobi solver and a Data Parallel Deep Learning Proxy application. Our design is compared against the state-of-the-art NCCL communication library and brings MPI performance much closer to the load-store vendor communication solutions in terms of performance.

IX. Acknowledgments

We acknowledge the support of Mitacs and the Natural Sciences and Engineering Research Council of Canada (NSERC), [ALLRP 5785392022]. We thank Nvidia for access to and support of the GH200 nodes used throughout this work.

REFERENCES

[1] TOP500 June 2024. https://www.top500.org/. https://www.top500.org/lists/top500/2024/06/, (accessed March. 26, 2024).

[2] J. Glaser, T. D. Nguyen, J. A. Anderson, P. Lui, F. Spiga, J. A. Millan, D. C. Morse, and S. C. Glotzer, "Strong scaling of general-purpose molecular dynamics simulations on GPUs," *Computer Physics Communications*, vol. 192, pp. 97–107, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010465515000867

[3] M. Pandey, M. Fernandez, F. Gentile, O. Isayev, A. Tropsha, A. C. Stern, and A. Cherkasov, "The transformational role of GPU computing and deep learning in drug discovery," *Nature Machine Intelligence*, vol. 4, no. 3, pp. 211–221, 2022.

[4] J. Yin, S. Gahlot, N. Laanait, K. Maheshwari, J. Morrison, S. Dash, and M. Shankar, "Strategies to deploy and scale deep learning on the summit supercomputer," in *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*, 2019, pp. 84–94.

[5] (2024) Message Passing Interface. [Online]. Available: http://www.mpi-forum.org

[6] (Jul.) MPICH is a high performance and widely portable implementation of the Message Passing Interface (MPI) standard. [Online]. Available: http://web.archive.org/web/20230722200816/https://www.mpich.org/

[7] "MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, RoCE, and Slingshot," https://mvapich.cse.ohio-state.edu/, Accessed Jul. 11, 2023.

[8] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine *et al.*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11*. Springer, 2004, pp. 97–104.

[9] C.-H. Chu, P. Kousha, A. A. Awan, K. S. Khorassani, H. Subramoni, and D. K. D. K. Panda, "NV-Group: Link-Efficient Reduction for Distributed Deep Learning on Modern Dense GPU Systems," in *Proceedings of the 34th ACM International Conference on Supercomputing*, ser. ICS '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3392717.3392771

[10] Y. H. Temuçin, S. Levy, W. Schonbein, R. E. Grant, and A. Afsahi, "A Dynamic Network-Native MPI Partitioned Aggregation Over InfiniBand Verbs," in *2023 IEEE International Conference on Cluster Computing (CLUSTER)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2023, pp. 259–270. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CLUSTER52292.2023.00029

[11] R. E. Grant, M. G. F. Dosanjh, M. J. Levenhagen, R. Brightwell, and A. Skjellum, "Finepoints: Partitioned Multithreaded MPI Communication," in *High Performance Computing*, M. Weiland, G. Juckeland, C. Trinitis, and P. Sadayappan, Eds. Cham: Springer International Publishing, 2019, pp. 330–350.

[12] M. Dosanjh and R. Grant, "Receive-Side Partitioned Communication," 9 2019. [Online]. Available: https://www.osti.gov/biblio/1763213

[13] P. V. Bangalore, A. Worley, D. Schafer, R. E. Grant, A. Skjellum, and S. Ghafoor, "A Portable Implementation of Partitioned Point-to-Point Communication Primitives," in *Poster: Proceedings of the 27th European MPI Users' Group Meeting*, ser. EuroMPI/USA '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1–3.

[14] M. G. Dosanjh, A. Worley, D. Schafer, P. Soundararajan, S. Ghafoor, A. Skjellum, P. V. Bangalore, and R. E. Grant, "Implementation and evaluation of MPI 4.0 partitioned communication libraries," *Parallel Computing*, vol. 108, p. 102827, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819121000752

[15] A. Worley, P. Prema Soundararajan, D. Schafer, P. Bangalore, R. Grant, M. Dosanjh, A. Skjellum, and S. Ghafoor, "Design of a Portable Implementation of Partitioned Point-to-Point Communication Primitives," in *50th International Conference on Parallel Processing Workshop*, ser. ICPP Workshops '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458744.3474046

[16] Y. Hassan Temuçin, R. E. Grant, and A. Afsahi, "Micro-Benchmarking MPI Partitioned Point-to-Point Communication," in *Proceedings of the 51st International Conference on Parallel Processing*, ser. ICPP '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3545008.3545088

[17] J. Dinan, "Accelerator Bindings for Partitioned Communication," https://github.com/. https://github.com/mpiwg-hybrid/hybrid-issues/issues/4, Accessed Oct. 18, 2022.

[18] P. G. Bridges, A. Skjellum, E. D. Suggs, D. Schafer, and P. V. Bangalore, "Understanding gpu triggering apis for mpi+x communication," 2024. [Online]. Available: https://arxiv.org/abs/2406.05594

[19] J. Dinan, "MPI Accelerator Extensions Prototype," https://github.com/. https://github.com/NVIDIA/mpi-acx, Accessed Jul. 7, 2023.

[20] B. Klenk, L. Oden, and H. Froening, "Analyzing Put/Get APIs for Thread-Collaborative Processors," in *2014 43rd International Conference on Parallel Processing Workshops (ICCP Workshops)*, 2014, pp. 411–418.

[21] R. E. Grant. Synchronization on Partitioned Communication for Accelerator Optimization. https://github.com/. https://github.com/mpi-forum/mpi-standard/pull/264, (accessed Jul. 7, 2023), link requires repoistory access.

[22] A. Venkatesh, S. Potluri, J. Dinan, and H. Mirsadeghi. Stream Synchronous Communication in UCX. https://ucfconsortium.org/. https://ucfconsortium.org/wp-content/uploads/2023/02/Day-3_2022-09-21-Stream-Synchronous-UCX-Jim-Dinan.pdf, (accessed July 4, 2024).

[23] D. J. Holmes, A. Skjellum, J. Jaeger, R. E. Grant, P. V. Bangalore, M. G. Dosanjh, A. Bienz, and D. Schafer, "Partitioned Collective Communication," in *2021 Workshop on Exascale MPI (ExaMPI)*, 2021, pp. 9–17.

[24] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, Y. Shahar, S. Potluri, D. Rossetti, D. Becker, D. Poole, C. Lamb, S. Kumar, C. Stunkel, G. Bosilca, and A. Bouteiller, "UCX: An Open Source Framework for HPC Network APIs and Beyond," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 2015, pp. 40–43.

[25] ROCm Communication Collectives Library. https://github.com/. https://github.com/ROCm/rccl, (accessed March. 26, 2024).

[26] NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/. https://developer.nvidia.com/nccl, (accessed March. 26, 2024).

[27] NVSHMEM. https://developer.nvidia.com/. https://developer.nvidia.com/nvshmem, (accessed March. 26, 2024).

[28] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine *et al.*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11*. Springer, 2004, pp. 97–104.

[29] S. H. Mirsadeghi, J. L. Traff, P. Balaji, and A. Afsahi, "Exploiting common neighborhoods to optimize mpi neighborhood collectives," in *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, 2017, pp. 348–357.

[30] NVIDIA Grace Hopper Superchip Architecture Whitepaper. https://resources.nvidia.com/. https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper, (accessed March. 26, 2024).

[31] AMD Infinity Architecture. https://www.amd.com/en/technologies/. https://www.amd.com/en/technologies/infinity-architecture, (accessed March. 29, 2024).

[32] Cerio Platform. https://www.cerio.io/. https://www.cerio.io/cerio-platform/, (accessed March. 29, 2024).

[33] J. Kraus. Benchmarking CUDA-Aware MPI. https://developer.nvidia.com/. https://developer.nvidia.com/blog/benchmarking-cuda-aware-mpi/, (accessed March. 29, 2024).

[34] CUDA Neural Network Implementation. https://luniak.io/. https://luniak.io/cuda-neural-network-implementation-part-1/, (accessed March. 29, 2024).

[35] R. Grant, A. Skjellum, and P. V. Bangalore, "Lightweight threading with MPI using Persistent Communications Semantics." in *Workshop on Exascale MPI (ExaMPI). Held in conjunction with the 2015 International Conferencefor High Performance Computing, Networking, Storage and Analysis (SC15)*, 2015, pp. 1–3. [Online]. Available: https://www.osti.gov/biblio/1328651

[36] T. Gillis, K. Raffenetti, H. Zhou, Y. Guo, and R. Thakur, "Quantifying the Performance Benefits of Partitioned Communication in MPI," in *Proceedings of the 52nd International Conference on Parallel Processing*, ser. ICPP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 285–294. [Online]. Available: https://doi.org/10.1145/3605573.3605599

[37] W. Schonbein, S. Levy, M. G. F. Dosanjh, W. P. Marts, E. Reid, and R. E. Grant, "Modeling and Benchmarking the Potential Benefit of Early-Bird Transmission in Fine-Grained Communication," in *Proceedings of the 52nd International Conference on Parallel Processing*, ser. ICPP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 306–316. [Online]. Available: https://doi.org/10.1145/3605573.3605618

[38] A. Bienz, D. Schafer, and A. Skjellum, "Mpi advance: Open-source message passing optimizations," *arXiv preprint arXiv:2309.07337*, 2023.

[39] S. Christgau, M. Knaust, and T. Steinke, “A First Step towards Support for MPI Partitioned Communication on SYCL-programmed FPGAs,” in *2022 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, 2022, pp. 9–17.

[40] J. A. Stuart and J. D. Owens, “Message passing on data-parallel architectures,” in *2009 IEEE International Symposium on Parallel Distributed Processing*, 2009, pp. 1–12.

[41] T. Miyoshi, H. Irie, K. Shima, H. Honda, M. Kondo, and T. Yoshinaga, “FLAT: A GPU Programming Framework to Provide Embedded MPI,” in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, ser. GPGPU-5. New York, NY, USA: Association for Computing Machinery, 2012, p. 20–29. [Online]. Available: https://doi.org/10.1145/2159430.2159433

[42] L. Oden, H. Fröning, and F.-J. Pfreundt, “Infiniband-Verbs on GPU: A Case Study of Controlling an Infiniband Network Device from the GPU,” in *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, 2014, pp. 976–983.

[43] E. Agostini, D. Rossetti, and S. Potluri, “Offloading Communication Control Logic in GPU Accelerated Applications,” in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2017, pp. 248–257.

[44] P. Markthub, J. Dinan, S. Potluri, and S. Howell. Improving Network Performance of HPC Systems Using NVIDIA Magnum IO NVSHMEM and GPUDirect Async. https://developer.nvidia.com/. https://developer.nvidia.com/blog/improving-network-performance-of-hpc-systems-using-nvidia-magnum-io-nvshmem-and-gpudirect-async/, (accessed April 2, 2024).

[45] A. Venkatesh, K. Hamidouche, S. Potluri, D. Rosetti, C.-H. Chu, and D. K. Panda, “MPI-GDS: High Performance MPI Designs with GPUDirect-aSync for CPU-GPU Control Flow Decoupling,” in *2017 46th International Conference on Parallel Processing (ICPP)*, 2017, pp. 151–160.

[46] H. Zhou, K. Raffenetti, Y. Guo, and R. Thakur, “MPIX Stream: An Explicit Solution to Hybrid MPI+X Programming,” in *EuroMPI/USA’22: 29th European MPI Users’ Group Meeting*, ser. EuroMPI/USA’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1–10. [Online]. Available: https://doi.org/10.1145/3555819.3555820

[47] N. Namashivayam, K. Kandalla, T. White, N. Radcliffe, L. Kaplan, and M. Pagel, “Exploring GPU Stream-Aware Message Passing using Triggered Operations,” 2022. [Online]. Available: https://arxiv.org/abs/2208.04817

[48] N. Namashivayam, K. Kandalla, J. B. W. I. au2, L. Kaplan, and M. Pagel, “Exploring fully offloaded gpu stream-aware message passing,” 2023. [Online]. Available: https://arxiv.org/abs/2306.15773