Utilizing Network Hardware Parallelism for MPI Partitioned Collective Communication

Yıltan Hassan Temuçin*, Amirreza Barati Sedeh*, Whit Schonbein†, Ryan E. Grant*, Ahmad Afsahi*
*ECE Department, Queen's University, Kingston, ON, Canada
†Sandia National Laboratories, Albuquerque, New Mexico, USA
*{yiltan.temucin, amirreza.baratisedeh, ryan.grant, ahmad.afsahi}@queensu.ca
†wwschon@sandia.gov

***Abstract*—Parallel distributed applications running on large-scale high-performance computing systems depend on effective point-to-point and collective communication to meet performance goals. Beginning with version 4.0, the Message Passing Interface (MPI) introduced the partitioned communication API, providing tools for addressing communication bottlenecks raised by hybrid communication models. This API allows individual actors (CPU threads, GPU threads, etc.) to initiate communication on portions of complete buffers, enabling additional communication/computation overlap. Intuitively, the utility of partitioned communication could benefit from network-level support: If there are multiple paths between endpoints, an MPI-aware network could disperse partitions across these paths, avoiding the data serialization entailed by a dependency on a single path. The Cerio Rockport Ethernet Fabric has the ability to expose this capability to communication middleware. In this work we develop this capability to allow for user-level path selection for MPI partitioned communication and explore how this capability impacts point-to-point performance, collective design, and Allreduce efficiency in a Large Language Model task**

***Index Terms*—Message Passing, Partitioned Communication, UCX, Collective Communication, Cerio Rockport Ethernet Fabric**

I. Introduction

Large-scale high-performance computing (HPC) systems depend on efficience point-to-point (P2P) and collective communication to achieve peak performance. For example, multi-threaded applications may complete computational work quickly through parallel computation, but lose this advantage as soon as thread synchronization is required for inter-process data transfer. Similarly, contemporary data-parallel machine learning algorithms rely heavily on the Allreduce collective, requiring periodic data transfer between all participating nodes. Efficient communication is therefore essential to application performance.

† 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.

The Message Passing Interface (MPI) is the *de-facto* standard for parallel programming in HPC [1]. Beginning with version 4.0, the MPI standard offers the partitioned communication API, which provides tools for addressing potential communication bottlenecks. Broadly speaking, the API allows a single memory buffer to be divided into partitions, where each partition is sent independently of the others. This means individual threads within a single MPI process could each initiate data transfer without having to wait for synchronization, and collectives can get ahead of synchronization by starting data movement when the data is ready.

Intuitively, the utility of partitioned communication could benefit from network-level support. Specifically, an MPI-aware network could disperse partitions across multiple, non-overlapping paths between endpoints, avoiding the potential data serialization entailed by a dependency on a single path.

While contemporary networks generally provide multiple paths for congestion control, how data is distributed across those paths is typically not under the control of higher-level communication libraries. A notable exception is the 'switchless' optical network offered by Cerio [2]. Under this technology, nodes are interconnected using passive top-of-rack (TOR) optical connections, and routing decisions are controlled entirely by the node-local network interface card (NIC). Each endpoint created on this network can have a distinct path in the network. This feature can be exposed via software to higher-level communication libraries (e.g., UCX or libfabric), providing a mechanism by which MPI can utilize the parallel endpoints.

In this paper we develop low-level optimizations for the Cerio Rockport Ethernet Fabric to enable MPI-aware multi-link communication for partitioned communication, and explore how this capability impacts P2P performance, collective design, and Allreduce efficiency in a large language model (LLM) task. Specifically, we make the following contributions:

- We provide the first study leveraging the user-level path-selection capabilities of the Cerio Rockport Ethernet Fabric for supporting MPI Partitioned communication;
- We present a UCX-based MPI Partitioned implementation that utilizes multiple paths to improve network bandwidth;
- We explore proposed MPI Partitioned Collectives and how multiple UCX workers can be used in collective design;

- And we investigate how MPI Partitioned communication can be applied to large language models such as GPT2.

The rest of the paper is organized as follows: Section II provides the necessary information on MPI Partitioned communication, the Cerio Rockport Ethernet Fabric, and UCX. Then in Section III, we present our UCX-based MPI Partitioned library as well as our MPI-based LLM. The designs are evaluated in Section V-A and used to understand the benefits of using multiple paths on a Cerio Network. Finally, we put our work into context in Section VI and bring it to a conclusion in Section VII.

II. Background

A. Cerio Rockport Ethernet Fabric

The Cerio Rockport Ethernet Fabric differs from most contemporary networks by eschewing central switches in favor of a distributed direct-interconnect design. Under this fabric, NICs located on each node connect to a Cerio SHFL (pronounced "shuffle"), a passive optical interconnect that provides pre-wired topology providing a 300Gbps fabric. This topology allows for multiple paths between any two nodes. The passive nature of the SHFL results in lower power consumption than alternatives that use a centralized switch. The 300Gbps connection is split between 12 fiber optic connections that each provide a 25Gbps bandwidth. The NIC connects to the host via a 100Gbps link. The fabric bandwidth (300Gbps) is higher than the host bandwidth (100Gbps) so that the NIC has the capacity to route messages to other nodes without transferring to host memory. This results in the NIC being able to saturate 4 optical links (4x25Gbs) when sending data. Using a proprietary method, data is injected into the network then it is converted into FLITs and routed across the multiple network paths.

A Cerio NIC allows for a single message to be split across multiple paths and combined at the receiver using their *MultiSpray* feature. However, the individual paths in a Cerio NIC can also be separately addressed in software by creating an InfiniBand queue pair (QP) for each path a user would like to send messages on. Since each link utilizes a different path, a specific path can be selected by choosing the appropriate QP. In this paper, we rely on the NICs default path selection mechanism. However, Cerio provides a tool, `flintpi`, to interact directly with hardware to select a path.

B. Unified Communication X (UCX)

UCX is a communication library that provides a high-performance network stack for many MPI libraries [3]. For this work, we focus on the Unified Communication Protocols (UCP) API and the InfiniBand specifics of UCX. A UCP *worker* represents an instance of communication resources that encapsulates a progress engine as well as a completion queue used to notify higher levels of the communication stack when communication requests have completed. Associated with each worker are one or more *endpoints* comprising connections with other endpoints associated with other workers. Each endpoint includes registered memory regions allowing the NIC to directly access host memory, as well as a pair of queues for processing communication requests targeting incoming or outgoing data. As previously noted, each queue pair (QP) is associated with a specific Cerio NIC path, and hence with a single path in the Cerio network. Details on how UCX resources are mapped to MPI Partitioned Communication are discussed in Section III-A.

C. MPI Partitioned Communication

1) MPI Partitioned Point-to-Point: Traditional MPI P2P communication follows a standard message-passing model: A receiver posts a buffer into which data will be placed, and a sender sends data from its local memory to that remote buffer. Communication is monolithic in the sense that all of the data to be sent must be ready to send before communication can be initiated. This can lead to delays in data movement under hybrid programming models. For example, when multiple threads are contributing to the data to be sent, a lagging thread holds up communication despite most of the data being ready [4]. MPI Partitioned communication addresses this issue by allowing individual contributors to a send buffer (e.g., threads) to independently initiate communication [1].

Figure 1 shows the sequence of steps involved in MPI Partitioned data movement. An application first initializes a persistent uni-directional communication channel between a sender and receiver using `MPI_Psend_init` and `MPI_Precv_init`. At this step, the application specifies how many partitions the buffers are split into. When the application is ready to communicate, it calls `MPI_Start` to notify the library communication will begin.

There is no guarantee the receiver is actually ready to receive data. This is especially problematic for an RMA-based implementation, as the sender needs to know the remote address where data is to be put. To address this limitation, we implement a `MPIX_Pbuf_prepare` procedure designed to guarantee a receiver is ready to receive, as is currently under

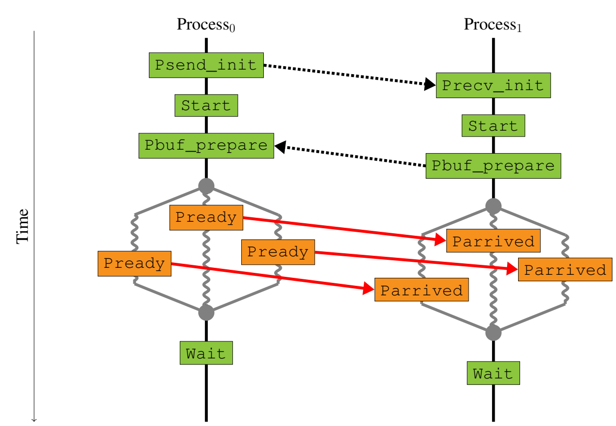

Fig. 1: Sequence Diagram of MPI Partitioned Point-to-Point Communication Model. Adapted from [5] to include the proposed call `MPIX_Pbuf_prepare`

consideration by the MPI Hybrid Working Group for inclusion in an upcoming version of the MPI standard [6].

Since the channel is now ready for data transfer, the application can enter a parallel region where each actor (e.g., a thread) marks its data as ready using `MPI_Pready`, which notifies the communication library that this partition can be transferred. Since each actor independently signals their data is ready, a well-designed MPI library can initiate data transfer for early arriving threads.

Finally, a receiver can test for the arrival of a specific partition using `MPI_Parrived`, and the entirety of the buffer through standard MPI methods such as `MPI_Wait`.

2) MPI Partitioned Collectives: Partitioned collective communication is a natural extension to P2P partitioned communication as it helps simplify the movement of data between groups of processes [7]. MPI Partitioned Collectives follow the same general structure as Partitioned P2P but require different initialization API calls (e.g., `MPI_Pbcast_init` and `MPI_Pallreduce_init`), and `MPIX_Pbuf_prepare` now requires synchronization across all of the processes participating in the collective; for more information, see [7].

III. Design

Our design consists of the P2P optimizations discussed in Section III-A and the partitioned collectives themselves in Section III-B. Our collectives are built upon our P2P design.

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

The OpenMPI implementation of the MPI standard supports using UCX as a transport layer. Currently, both RMA and P2P have a UCX component, but partitioned communication does not. For this work, we design a new UCX component optimized specifically for the Cerio Rockport Ethernet Fabric.

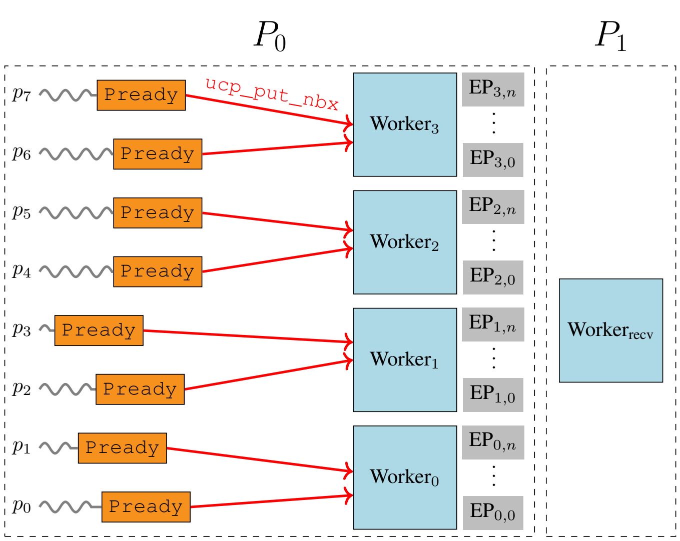

Fig. 2: High-Level Diagram of Implementing MPI Partitioned Point-to-Point Communication Over UCX

1) MPI_{Psend, Precv}_init: Initialization of partitioned communication creates a UCP context, a set of workers, and endpoints. A general representation of this process is shown in Figure 2, where the sending process (P_0) instantiates multiple workers, each responsible for handling work requests from multiple threads, and each progressing multiple associated endpoints ($EP_{i,j}$). Since a single worker can have multiple endpoints, each endpoint contains a QP, and each QP utilizes a different NIC path, one possible design uses a single worker and four endpoints (one per NIC path). However, to reduce contention between threads for locks within a worker, we opted to create one worker per path, where each worker has a single endpoint. The receiving process (P_1) requires only a single worker and endpoint.

During initialization, the sender also pre-populates the parameters for the put operation and packs the information into a `setup_t` object which is sent to the receiver in a non-blocking fashion. The receiver posts a corresponding receive operation to accept the incoming `setup_t` object.

2) MPI_Start, MPIX_Pbuf_Prepare: `MPI_Start` marks the MPI requests associated with the present communication as pending and sets internal flags to their default state. `MPIX_Pbuf_Prepare` is required to ensure the receiver is ready to receive. The first time this API is called, the receiver checks for the setup object sent by the sender, and once it is received, registers the receive buffer (using `ucp_mem_map`) and sets internal flags used to track the status of partitions. The receiver then creates a setup object in response, containing information necessary for the sender to use RMA operations, such as memory keys. Simultaneously, the sender waits for the setup object response, and using the response, creates the relevant endpoints and unpacks the memory keys. After these steps are completed, the sender can put data to the receiver. Subsequent calls to `MPIX_Pbuf_Prepare` do not incur the overheads of this initial call, because the sender and receiver already have all of the information required to perform RMA operations; consequently, subsequent calls are nothing more than the sender waiting for a 'ready-to-receive' signal from the receiver.

3) MPI_Pready, MPI_Parrived: Once a thread is ready and calls `MPI_Pready`, it executes `ucp_put_nbx` to move the partition to the receiver's buffer. UCX allows users to chain multiple functions together, so we attach a callback to these operations: a second `ucp_put_nbx` call which marks a partition as received on the receiver. This is required as UCX does not provide receive-side completions and issuing a flush could block additional partitions being sent on that QP. This requires some send-side progression but it is deferred to `MPI_Wait`. The `MPI_Parrived` call simply polls a flag in memory to obtain the status. This does not require any additional receive-side progression, or using any additional network hardware resources, as the flags are updated by the sender.

4) MPI_Wait: This call progresses put operations that have not completed. The sender tracks the number of completions to determine when the send buffer is safe to reuse and the request can be marked as complete. On the receive-side, flags (one per partition) are checked until the expected number of partitions are received.

B. MPI Partitioned Collectives

Partitioned Collectives are implemented using the P2P library previously described.

1) MPIX_P<collective>_init: Similar to the P2P API, current MPI Partitioned Collective proposals have an initialization function for each collective (e.g. `MPI_Bcast`, `MPI_Allreduce`, etc.). We generalize these collective initialization calls and refer to them as `MPIX_P<collective>_init`. The generalization of Partitioned Collectives is incredibly important to consider as the current proposal has at least 21 collectives to be implemented by MPI libraries [7]. As this is quite burdensome for MPI developers, we take inspiration from MPI Neighborhood Collectives and create a schedule for arbitrary communication patterns [8], based on the design given in [9]. We first initialize the collective and within the request object, we populate the communication schedule that is unique to the algorithm that we are executing. During `MPI_Pready` we issue a put as per the P2P design. In `MPI_Wait`, we progress the outstanding puts as well as progressing the collective schedule.

IV. Experimental Platform

For our evaluation we use an eight node Intel Xeon Gold 6338 CPU 2.00GHz system, where each CPU has 32 cores (two hardware threads per core) and 128GiB of memory. Nodes are connected using the Cerio Rockport Ethernet Fabric (NC1225) in a 6D torus topology, as described previously. For our software environment we use the GNU/Linux distribution Rocky Linux 9.3, OpenMPI v5.0.1rc1, and UCX v1.14.1 compiled with GCC version 11.4.1.

V. Experimental Results

We begin by focusing on MPI Partitioned P2P communication, go on to consider Partitioned Collectives, and finally evaluate our collective design with an LLM application.

A. Partitioned Point-to-Point

In this section we first evaluate how using different numbers of paths can impact partitioned communication performance before comparing the performance of MPI Partitioned to traditional P2P.

1) Multiple UCX Workers in MPI Partitioned: Each UCX worker contains an endpoint mapped to a QP that is associated with one of the four 25Gbs Cerio NIC paths. To confirm that the Cerio hardware can be directly controlled from UCX without modification to lower software layers, and that the number of paths impacts Partitioned performance, we utilized a bandwidth test with 32 partitions distributed across one to eight workers for message sizes varying from 128B to 32MiB (Figure 3). Consistent with expectations, using four workers provides peak bandwidth for large messages, with a maximum

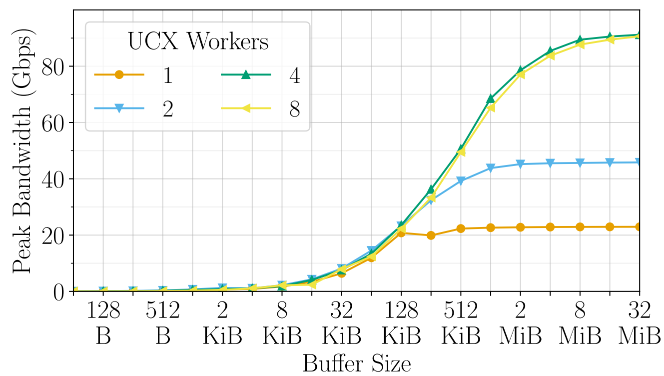

Fig. 3: Bandwidth For MPI Partitioned Point-to-Point With Different Number of Workers

3.97x improvement over a single worker. Going beyond four workers provides no additional benefit.

2) Comparison to Existing MPI Libraries: In Figure 4 we compare our UCX design to OpenMPI's (v5.0.1rc1) existing implementation of MPI Partitioned (with and without Cerio's hardware-based multispray) as well as to traditional `MPI_Send/Recv`. Hardware multispray is denoted as **MS** in all figures, and OpenMPI is abbreviated as **OMPI**. We compare against hardware multispray as it allows us to compare our design (which is effectively a software-based multispray) with the hardware approach. It also provides a fairer reference baseline as we can compare two designs that both use multiple paths rather than only compare to current software designs which only use a single path.

From our evaluation, both the default OpenMPI partitioned implementation and traditional `MPI_Send/Recv` obtain a peak bandwidth of 22.93Gbps. This is expected, because `MPI_Send/Recv` only uses a single worker and endpoint when UCX is used for its communication backend, resulting in using only a single 25Gbs link in the network. Likewise, the default Open MPI partitioned implementation is internally built upon OpenMPI's Point-to-Point Management Layer, resulting in similar behavior. The discrepancy between the two occurring at between 32KiB and 128KiB is due to MPI Partitioned subdividing the buffer, enabling the use of an eager protocol in comparison to traditional, which uses rendezvous.

After 2MiB, the hardware multispray approach outperforms the default Send/Recv and the default OpenMPI implementation. For a 32MiB message we observe a maximum 3.91x performance improvement in partitioned communication when hardware multispray is enabled. This improvement over baseline is expected because hardware multispray utilizes multiple paths. We believe that the performance degradation in the 32KiB-1MiB range stems from MPI Partitioned subdividing the message into 32 partitions which are then further subdivided by the hardware multispray among the eight paths. The maximum transmission unit (MTU) on this network is 4096KiB therefore we do not fill a single MTU until 1MiB ($32 \times 8 \times 4096$KiB).

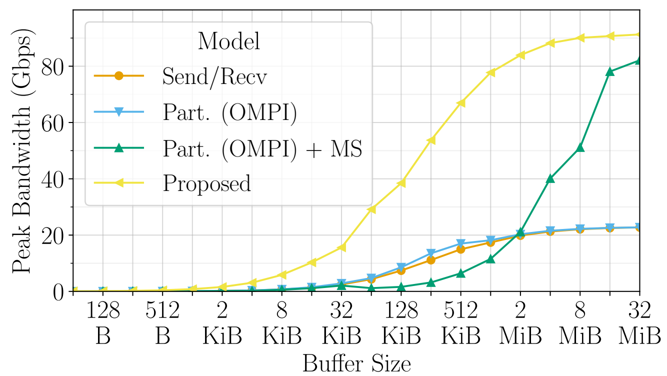

Fig. 4: Bandwidth For The UCX Optimized MPI Partitioned Point-to-Point compared to the existing MPI Partitioned Implementation, as well as MPI_Send/Recv

Our UCX-based MPI Partitioned design outperforms both the default OpenMPI partitioned implementation and `MPI_Send/Recv` due to using multiple paths, reaching a peak bandwidth of 91.27Gbps. Our design also provides an additional 11.2% performance improvement compared to the hardware multispray at 32MiB. These results illustrate the benefits of using a software solution for MPI Partitioned on hardware that provides access to communication parallelism via multiple communication contexts (e.g., UCX endpoints).

3) MPI_Parrived Overhead: `MPI_Parrived` allows a receiver to determine whether a particular partition (or set of partitions) has arrived. The overhead of this API call is critical for collectives, as they will be built upon this P2P library, and `MPI_Parrived` is used to poll partitions to determine when the collective should progress to the next step in the communication schedule.

To measure the overhead of `MPI_Parrived`, we launch n OpenMP threads, each assigned to a distinct partition, and poll `MPI_Parrived` for 1000 iterations. Figure 5 presents results from 100 samples of this microbenchmark for the default OpenMPI (v5.0.1rc1) implementation and our proposed UCX implementation.

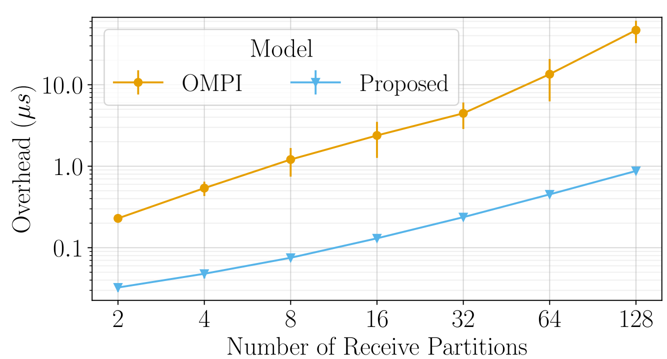

Fig. 5: Overhead of polling receive-side partitions using `MPI_Parrived`

For all partition counts the UCX implementation outperforms OpenMPI. At two partitions we observe 7.05x decrease in total polling time and the best improvement (53.41x) occurs at 128 partitions. The OpenMPI implementation of `MPI_Parrived` requires each thread acquire a lock before entering the progression engine. During each entry to the progress engine, OpenMPI calls `ompi_request_test` n times for each partition. As our micro-benchmark ran for 1000 iterations, this causes the MPI library to enter the progression engine anywhere between 1000 and 1000$*n$ times. This behavior is reflected in Figure 5 with the larger error bars shown for the OpenMPI implementation. Our maximum standard error for the OpenMPI implementation is $\sigma_M = 14.34\mu s$, where as for UCX it is $\sigma_M = 0.04\mu s$. This large variability is not shown in the UCX implementation as `MPI_Parrived` is only required to poll a flag in memory. This more simple and lower-cost design is allowed by our usage of an RMA-based implementation rather than a two-side send/recv implementation by OpenMPI which requires receive-side progression.

B. Partitioned Collectives

As per the P2P evaluation, in this section we evaluate our partitioned allreduce design using multiple paths and compare it to the existing OpenMPI `MPI_Allreduce`. As our target is LLMs, we focus on large message sizes in our collective evaluation, and use a ring allreduce algorithm for both the `MPI_Allreduce` and the partitioned allreduce.

1) Multiple UCX Workers in MPI Partitioned: Figure 6 shows the impact of varying the number of workers (i.e., paths) on Goodput for message sizes ranging from 512KiB to 512MiB. On eight nodes, using two workers provides a maximum speedup of 1.54x at 128MiB in comparison to a single path, and four paths provide a peak 2.33x performance improvement at 128MiB. Using multiple paths is an important aspect in providing performance for collective communication on this platform. However, using eight workers results in a performance degradation. As we are using the ring allreduce algoirthm, we receive from the previous rank, and send to the

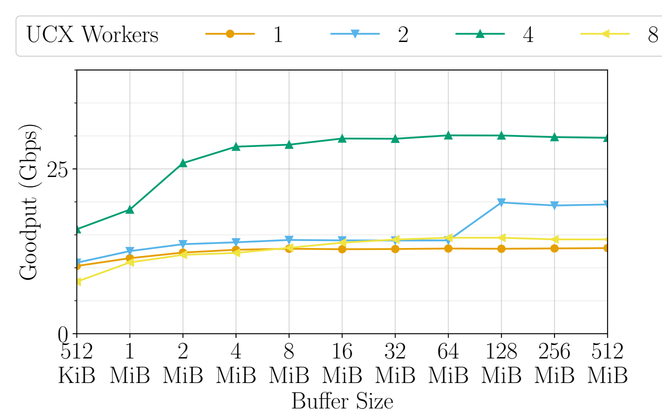

Fig. 6: Partitioned Allreduce Bandwidth for Different Number of UCX Workers on Eight Nodes

next. With eight workers, our design attempts to receive data from eight distinct paths and send data through eight distinct paths. This results in the software trying to use 16 paths despite only having 12 physical paths, severely impacting performance.

2) Comparing Different Communication Models: An important consideration when using our Multi-Path Partitioned Collectives is if there is any benefit to using it compared to the traditional `MPI_Allreduce`. Figure 7 compares the performance of `MPI_Allreduce` with and without hardware multispray (**AR** and **AR + MS**) to its partitioned variant with and without using `MPI_Parrived` (labelled **PAR** and **PAR + Parrived**). The partitioned communication experiments use four partitions. Unfortunately, we cannot include results for allreduce implemented using OpenMPI's partitioned communication, as the collectives hang. This occurs despite using the same collective code (only the underlying implementation of partitioned communication is changed), and occurs on other systems, indicating the issue does not stem from the Cerio hardware. That said, we expect such an implementation to perform similarly to the traditional `MPI_Allreduce` since by default OpenMPI only uses a single path.

The default `MPI_Allreduce` provides the lowest Goodput for most of the message sizes we evaluated. Enabling hardware multispray provides a 2.47x speedup (at 512MiB) compared to the default `MPI_Allreduce`, indicating the benefit of using multiple paths. In contrast, our partitioned collective design provides a speedup of 3.05x at 64MiB, indicating a more effective use of multiple paths than hardware multispray. Using `MPI_Parrived` shows minor benefits for messages smaller than 8MiB. Calling `MPI_Parrived` allows the MPI runtime to progress messages for each independent path in the network as well as reduce it's own portion of data. If we continue to scale, we would expect the delta between using partitioned collectives with and without `MPI_Parrived` to decrease. However, using `MPI_Parrived` nonetheless allows users to obtain fine-grained information on their buffers with no performance penalties.

In general, these results indicate using multiple paths for communication on a Cerio network is highly beneficial for

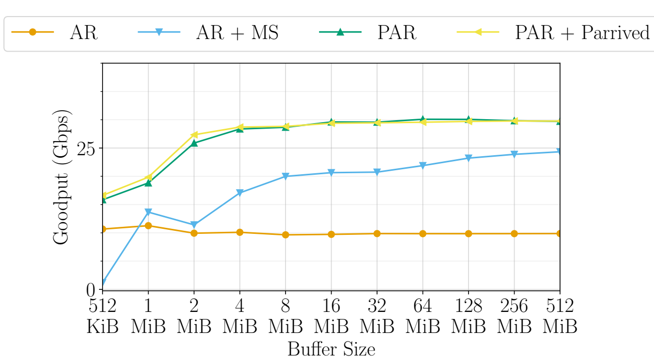

Fig. 7: Different Communication Models for an Allreduce Operations Compared on Eight Nodes

allreduce collectives. The benefits are amplified with our software-based path selection design, but hardware multispray could be useful to applications that do not use partitioned communication.

C. Data Parallel Large Language Model Results

In data parallel training, each process trains an instance of an ML model on different training data before parameters are exchanged amongst processes, each copy of the model is updated, and the process repeats. For this experiment, we use `llm.c` framework [10], a project that provides a GPT2 model from Open AI [11], specifically a 124M parameter model. The number of model parameters correlate with the message size of our Allreduce operations. We use the tinyshakespear [12] dataset, which contains text from a variety of Shakespeare's plays. We modify the framework to handle data-parallel distributed training using MPI. To distribute data across processes, we modify the dataloader of `llm.c`, so that each mini-batch of data is offset by the world rank, ensuring each process trains on a distinct batch of data.

During the sharing of parameters across models, we use AdamW as our optimizer [13]. AdamW is an extension to the Adam (Adaptive Moment Estimation) optimizer where the learning rate (step size) and weight decay (L_2 regularization) are optimized separately. The optimizer is applied as part of an allreduce collective, with the results distributed back to each model instance.

For the `MPI_Allreduce` approach we used the parameter `MPI_IN_PLACE` so that only a single buffer is used for model parameters. This is the same approach used by frameworks such as Horovod [14]. For the partitioned allreduce approach we create two buffers for model parameters, and we call `MPI_Pallreduce_init` twice with the send and receive buffers permuted. This allows for double buffering of model parameters. The memory cost of these two designs is the same because OpenMPI's implementation of `MPI_Allreduce` internally allocates a temporary buffer at the start of the call and frees it upon exit. However, we expect the initial iteration of the partitioned allreduce to have a significant initialization cost.

We will evaluate the overhead in Section V-C1. Additionally, we evaluate the overheads of using partitioned collectives, weak scaling throughput, and our training loss convergence.

1) Overheads: The idea behind using partitioned communication is that there is an initial cost to communicating that is amortized after many iterations of using the same persistent buffer. The LLM we used also has an initial cost which stems from lazy allocation of its buffers. In Figure 8, we measure the cost of our first iteration in training for `MPI_Allreduce` with and without multispray as well as our partitioned collective. As expected, the partitioned collective has a significant cost to initialization whereas the `MPI_Allreduce` versions do not. Interestingly, using hardware multispray has an overhead of at most 9% more than without.

2) Throughput: To put these overheads into context we consider throughput measurements. In this test, we present

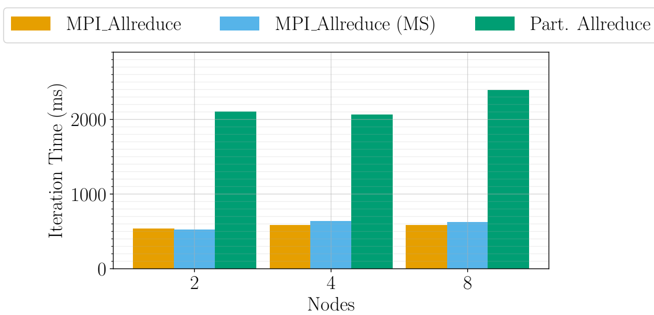

Fig. 8: Overhead of initial iteration when using different communication models for GPT2

the average cost of a single iteration of training over 100 trials. Figure 9 shows iteration time for the different allreduce methods.

Since both Cerio's hardware and our software multispray approaches outperform the baseline `MPI_Allreduce` implementation, it is clear that using multiple paths is integral to obtaining good performance on this network. Moreover, considering the overheads shown in Figure 8 (which show significant overheads for our UCX partitioned allreduce), the current results confirm that the initialization costs are sufficiently amortized to put our approach in line with the hardware multispray technique, which has significantly lower overhead costs.

To understand the true trends we would need to have access to a larger cluster with the Cerio Rockport Ethernet Fabric to understand the tradeoffs between our design and hardware multispray. That said, it is clear from these experiments that collectives using multiple paths are required to obtain better performance on this type of network, and enabling user-level path selection can help.

3) Training Loss: For completeness, we also investigated the impact of our optimized Partitioned communication implementation on model training. Training is often not included in HPC-focused papers due to the resource requirements to obtain the data. Consequently, the data in Figure 10 is only from a single run due to these resource constraints. We trained the model for 8000 iterations with the different allreduce

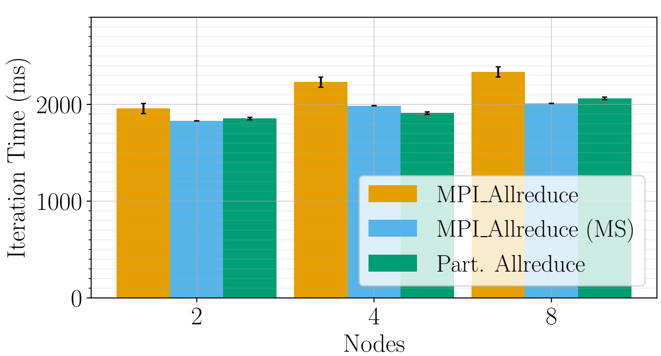

Fig. 9: Training Iteration Time of GPT2 when using A Partitioned Allreduce relative to `MPI_Allreduce`

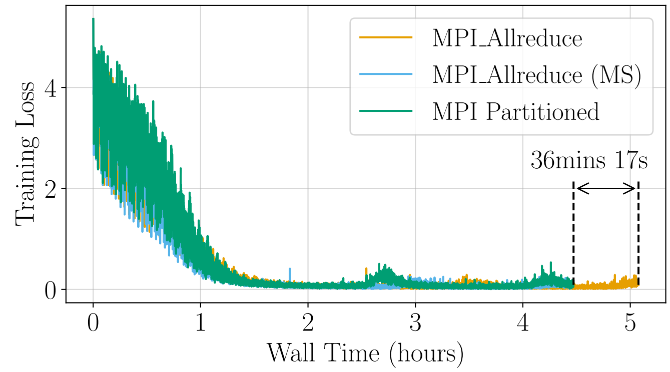

Fig. 10: Training Loss of Allreduce Operations For GPT2 on Eight Nodes

operations that we are studying. For all tests in this section, we used eight nodes, one process per node, and 128 threads per process. Partitioned allreduce results utilize four partitions.

As can be seen from the figure, our optimized implementation decreased training time by 36 minutes and 17 seconds. This is significant, since without multi-pathing training takes five hours. Similarly to the throughput results, the delta between hardware multispray and our software multi-path design is small (0.65%). However, it is certainly safe to claim that using multiple paths itself significantly improves performance on this network compared to the default state of the network.

VI. Related Work

MPI Partitioned Point-to-Point was first introduced by Grant et al. where the authors address the problem of allowing multiple threads to easily commit data which is ready for transfer without the MPI runtime acquiring internal locks [15]. The `MPI_Pready` function to commit data was introduced in [16]. Temuçin et al. provide network hardware optimizations for MPI Partitioned [17]. They explored using guidance from a model to select different combinations of Work Queue Element (WQE) aggregation and using multiple QPs to optimize communication on InfiniBand hardware. While the present work also considers the use of multiple QPs, we present a method to access network hardware parallelism via UCX. Moreover, we consider the implications of using multiple QPs on the Cerio Network to select distinct links, allowing multi-threaded MPI Partitioned applications to transfer data in parallel. Holmes et al. introduced Partitioned Collectives [7] by defining behavior for various collective types, and the relationship that partitions should have between the sender and receiver. However, to our knowledge, this is the first work on implementing and optimizing partitioned collectives.

Optimizing MPI libraries for different networks has been studied in great depth. Zambre et al. explored the implementation trade-off between performance and resource usage on Mellanox InfiniBand hardware for MPI Endpoints [18]. They highlight the limitations of QP locks on ConnectX-4 and how threads can be mapped to different device contexts to alleviate resource contention. This is somewhat resolved on the Cerio Network as our QPs are intended to be used in a fashion that selects independent links in our network. These types of optimizations for network hardware are not unique to InfiniBand-Based hardware. Hjelm et al. investigated methods to reduce thread contention for network resources on Cray Aries and Gemini networks [19]. They created multiple ugni device contexts that were serviced in a round-robin fashion to allow MPI RMA to issue multiple `MPI_Put` and `MPI_Get` operations in a multi-threaded enviroment. Our designs could also be applied to the RMA interface, however, here we focus on MPI Partitioned due to its recent introduction into the MPI standard. Besta et al. investigated low diameter topoloigies to understand how routing protocols can be used to exploit path diversity [20]. Our work differs as we are the end users of the Cerio network and are unable to directly control the routing algorithms. Rather, we rely on their exposure to different paths and how it can be applied to MPI Partitioned.

VII. Conclusion

The Cerio Rockport Ethernet Fabric provides multiple distinct paths between source and destination. Typically, interconnects do not provide user-level access to distinct links or paths. These paths can be controlled by creating a distinct QP for each path, so that we can send messages in parallel, to better utilize network bandwidth on the NC1225. In this work, we present a UCX-Based MPI Partitioned Communication library design that effectively utilizes the different links on the Cerio network by creating multiple UCX endpoints between source and destination. We evaluated our design at the micro-benchmark level, demonstrating the benefits of using multiple links and the trade-offs involved, including a comparison with the hardware multispray feature. Both point-to-point and collective communication is evaluated. It is clear that using multiple links during allreduces collectives provide significant performance improvement to the Cerio hardware. For our LLM application, we are able to close the gap between the hardware multispray and our software multi-pathing design.

A. Future Work

Graphics Processing Units are well known for accelerating AI workloads. Therefore, we plan to use the CUDA version of `llm.c` to scale our application to Cerio's composable disaggregated infrastructure (CDI) products as we see value in multi-path networking on GPU systems. GPU architectures contain multiple streaming processors (SM) so it would be valuable to understand how SMs could be mapped to paths in the network to provide GPUs with their own separate communication contexts as kernel are scheduled.

References

[1] (2024) Message Passing Interface. [Online]. Available: http://www.mpi-forum.org

[2] Cerio Platform. Https://www.cerio.io/cerio-platform/.

[3] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss *et al.*, "UCX: an open source framework for HPC network APIs and beyond," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 2015, pp. 40–43.

[4] W. P. Marts, M. G. F. Dosanjh, W. Schonbein, S. Levy, and P. G. Bridges, "Measuring Thread Timing to Assess the Feasibility of Early-bird Message Delivery," in *Proceedings of the 52nd International Conference on Parallel Processing Workshops*, ser. ICPP Workshops '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 119–126. [Online]. Available: https://doi.org/10.1145/3605731.3605884

[5] 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

[6] R. E. Grant. (2024) Synchronization on Partitioned Communication for Accelerator Optimization. [Online]. Available: https://github.com/mpi-forum/mpi-standard/pull/264

[7] 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 *Workshop on Exascale MPI (ExaMPI). Held in conjunction with the 2012 International Conferencefor High Performance Computing, Networking, Storage and Analysis (SC21)*, 2021, pp. 9–17.

[8] 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.

[9] Y. Hassan Temuçin, W. Schonbein, S. Levy, R. E. Grant, A. Sojoodi, and A. Afsahi, "Design and Implementation of MPI-Native GPU-Initiated MPI Partitioned Communication," in *Workshop on Exascale MPI (ExaMPI). Held in conjunction with the 2024 International Conferencefor High Performance Computing, Networking, Storage and Analysis (SC24)*.

[10] A. Karpathy. (2024) LLM training in simple, raw C/CUDA. [Online]. Available: https://github.com/karpathy/llm.c

[11] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[12] A. Karpathy, "char-rnn," https://github.com/karpathy/char-rnn, 2015.

[13] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," 2019.

[14] Y. H. Temuçin, A. Sojoodi, P. Alizadeh, and A. Afsahi, "Efficient multi-path nvlink/pcie-aware ucx based collective communication for deep learning," in *2021 IEEE Symposium on High-Performance Interconnects (HOTI)*, 2021, pp. 25–34.

[15] 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

[16] 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.

[17] 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)*. IEEE, 2023, pp. 259–270.

[18] R. Zambre, A. Chandramowlishwaran, and P. Balaji, "Scalable Communication Endpoints for MPI+Threads Applications," in *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, 2018, pp. 803–812.

[19] N. Hjelm, M. G. F. Dosanjh, R. E. Grant, T. Groves, P. Bridges, and D. Arnold, "Improving mpi multi-threaded rma communication performance," in *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3225058.3225114

[20] M. Besta, J. Domke, M. Schneider, M. Konieczny, S. D. Girolamo, T. Schneider, A. Singla, and T. Hoefler, "High-performance routing with multipathing and path diversity in ethernet and hpc networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 4, pp. 943–959, 2021.