SAND2023-08000C

A Dynamic Network-Native MPI Partitioned Aggregation Over InfiniBand Verbs

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

***Abstract*—Modern HPC systems require efficient hybrid programming model to utilize their hardware resources effectively. The Message Passing Interface (MPI) has accommodated next-generation hardware by providing new APIs such as the MPI Partitioned interface. This API provides a user with fine-grain communication without the overhead of traditional MPI point-to-point communication in multi-threaded workloads.**

To the best of our knowledge, we present the first work on detailed low-level design for an MPI Partitioned implementation. We guide readers through a method to map the MPI Partitioned interface to the InfiniBand Verbs API. Alongside implementation details, we also study the aggregation of user partitions and how we can efficiently send them over the network. We study a brute force approach and using the Partitioned LogGP (PLogGP) model to predict ideal aggregation. We observe that using the PLogGP model provides comparable performance without exhausting computing resources to search the entire solution space. The PLogGP design was further optimized by considering how the partition arrival pattern can be used to dynamically modify our aggregation scheme. We profiled our micro-benchmarks to provide analysis on how and why this additional optimization is beneficial to our results and how we can fine-tune this mechanism. Finally, we evaluated our PLogGP and Timer-based PLogGP designs with a commonly used communication pattern in HPC (communication sweep) to observe the impact when communicating with multiple processes in an application-like scenario at 1024 cores.

***Index Terms*—Message Passing, Partitioned Communication, InfiniBand, Aggregation, Multi-Threaded**

I. Introduction

The Message Passing Interface (MPI) [1] is one of the most popular programming models for high-performance computing (HPC). MPI allows a user to transparently program an application without explicitly considering the hardware platform. Ideally, platform specific hardware optimizations are left to MPI implementer. For example, an implementer can optimize MPI for specific network hardware [2]. Alongside hardware considerations, it is important to understand how a user may call into an MPI library. Multi-threading MPI applications enables programmers to maximize resource utilization and application performance on HPC systems built around modern, many-core processors. As a result, a recent survey on MPI usage within the US Exascale Computing Project (ECP) [3], showed that 86% of application and system software developers plan to use MPI with multiple threads, and 82% of users plan to make MPI calls within multi-threaded regions of their code.

The promise of multi-threading in MPI applications has led to the development of several related features in MPI. Since MPI-2.0, the standard has had the threading modes; `MPI_THREAD_FUNNELLED`, `MPI_THREAD_SERIALIZED`, `MPI_THREAD_MULTIPLE`. However, theses threading modes have limitations that require many internal locks to share hardware/software resources across different threads when using MPI point-to-point. Lock-contention can arise from sharing MPI objects [4], message matching [5] and accessing network hardware [6]. To address these issues, researchers have proposed additional interfaces that MPI could have to alleviate these problems, such as Endpoints [7] and Finepoints [8]. Modifications were made to the Finepoints proposal and it was formalized as MPI Partitioned Point-to-Point Communication in the MPI-4.0 standard, in June of 2021.

MPI Partitioned provides semantics similar to point-to-point communication but it can leverage software-level optimizations and hardware-level triggered operations to enhance the performance of multi-threaded applications. Its API is designed in a manner which minimizes the lock contention issues that were present in traditional MPI point-to-point communication. In this new programming model, the send and receive buffers of a point-to-point communication are partitioned into distinct chunks which can be addressable by individual actors. MPI Partitioned communication will be further explained in Section II-A.

At this current moment, MPI Partitioned has yet to be fully optimized on specific hardware platforms. The majority of work on MPI Partitioned has been on the interface itself and high-level software optimizations [8], [9], [10], [11], [12]. In this paper we will explore low-level optimizations and we will make the following contributions:

- We provide the first study on MPI Partitioned Point-to-Point communication that uses low-level networking libraries in detail. We present our mapping of the MPI Partitioned interface to InfiniBand Verbs as an example of how an MPI implementer may want to design their library.
- We study aggregation of user partitions to improving network utilization by using a brute force approach as well as using the extended LogGP model for MPI Partitioned to optimize library performance for medium-sized messages.
- We investigate how partition arrival could impact user partition

aggregation and propose a novel aggregation scheme which provides additional speedup for a sweep communication pattern.

The rest of the paper is organized as follows: Section II provides the necessary information on MPI Partitioned Point-to-Point Communication, InfiniBand Verbs, and the Partitioned LogGP model. In Section III, we discuss the motivation for this work. Then in Section IV, we explain the details of our design and evaluate it in Section V. We discuss related research in Section V-E and then conclude our work in Section VI.

II. Background

In this section, we introduce the MPI partitioned point-to-point communication model and highlight the relevant parts of the MPI interface. We also provide some background information on Remote Direct Memory Access (RDMA) programming with InfiniBand network hardware using InfiniBand verbs.

A. MPI Partitioned Point-to-Point Communication

MPI Partitioned Point-to-Point Communication extends traditional point-to-point semantics in a way that allows for easy use with hybrid programming models [1]. With partitioned communication, the send and receive buffers are segmented and actors (threads, in the context of this paper) mark data ready for transfer. Actors can take the form of OpenMP threads, POSIX threads, GPU thread block, and so on. A high level diagram is presented in Figure 1 to help visualize this programming model.

An application which uses MPI Partitioned first initializes communication using `MPI_Psend_init` and `MPI_Precv_init`. With these function calls the MPI run-time registers the persistent buffers, the partition size, and the partition count before any data transfers occurs. The `Psend/Precv` calls are also matched between processes using the tag, rank, and communicator in the order they are posted. Unlike MPI point-to-point communication, MPI Partitioned does not support wildcards. This is beneficial for highly-threaded codes as it avoids matching list overheads when wildcards are allowed. Once the application is ready to communicate, `MPI_Start` is called to start communication between the predefined buffers. Everything prior to this point must be called within a serial portion of an application or by a single thread and is required to be non-blocking [13, Chapter 4.2].

In the parallel region of application code, the sender thread computes its calculation, and once the data is ready to transfer, the application calls `MPI_Pready` to inform the MPI run-time that the partition can now be transferred to the receiver. In Figure 1, it is shown that the data is directly transferred between threads as we call `MPI_Pready`. However, calling `MPI_Pready` only indicates that the partition is ready to be transferred; it does not require the MPI implementation to transfer it immediately. The frequency and mechanism of transfer is ultimately dictated by the MPI implementation (as we discuss in Section IV) but this is how MPI users can reason about their program. Once the sender exits a parallel region it must call `MPI_Test` or `MPI_Wait` to complete a partitioned communication transfer.

On the receive side, the process can use `MPI_Test` to check if all partitions have arrived, or it can use `MPI_Parrived` which provides finer grained information to test whether individual partitions have arrived. `MPI_Parrived` can also be called by individual threads in a parallel region. Again, to complete a partitioned communication transfer we must use `MPI_Test` or `MPI_Wait` on the receiving process.

As MPI Partitioned is persistent, to restart the communication and reuse the buffer, the application developer can call `MPI_Start` to begin the data transfers again. If MPI Partitioned is implemented efficiently it should not suffer from some of the issues we see from using MPI send/receives in multi-threaded environments. As communication has different stages, the message matching occurs only once, prior to communication and it is less problematic than searching the message queue with multiple threads. A good implementation should also reduce any lock contention that is observed with MPI point-to-point communication [15].

B. InfiniBand Verbs

InfiniBand is a network specification maintained by the InfiniBand Trade Association [16]. InfiniBand is used to connect high-performance compute nodes together to provide high throughput and low latency. This is achieved by InfiniBand allowing applications to transfer data between two nodes without depending on the remote node's operating system for the data movement.

To use an InfiniBand network card, a user first creates a user space device context and allocates a *protection domain* (PD). A PD encapsulates resources such as *memory regions* (MRs) and *queue pairs* (QPs) to prevent arbitrary access. MRs allow for a network interface controller (NIC) to remotely access data. A QP consists of a *send queue* (SQ) and a *receive queue* (RQ). QPs which communicate with one other must be connected to together during initialization. *Completion queues* (CQs) are not within the PD but they are used to notify users when their communication request has completed.

A user can create a *work request* (WR) which contains an opcode, the remote key, the remote address, the local data layout, and some immediate data (if applicable). The data layout can be specified with the use of a scatter/gather array (`sg_list`) where each item is a scatter/gather element (SGE). A SGE contains the start address of the local memory buffer within our MR, the length of the buffer and the local key. For a single contiguous buffer we would have list of size one containing a single SGE.

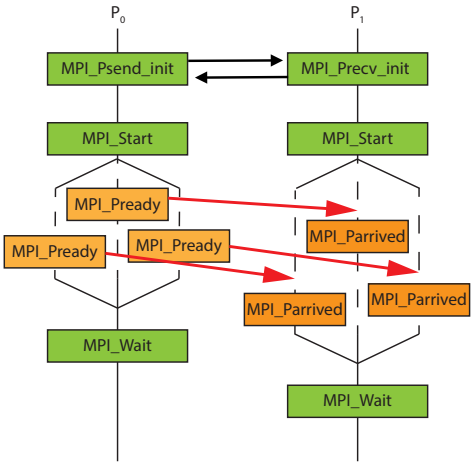


Fig. 1: Overview of the MPI Partitioned point-to-point communication model, adapted from [14]

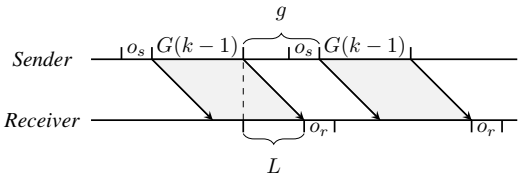


Fig. 2: The LogGP model applied to the transmission of two back-to-back k-byte messages.

Using `ibv_post_send`, a user can post a WR to a QP to generate a *work queue entry* (WQE). Then a user can poll our CQ using `ibv_poll_cq`. Once the device has completed the corresponding WR it issues a *work completion* (WC) on the CQ. If an opcode using an `*_WITH_IMM` suffix is used, then WC contains the associated immediate data.

C. The Partitioned LogGP (PLogGP) Model

The LogGP model of parallel communication models the time to transfer a message from sender to receiver as a linear function of network latency (L), sender and receiver processor overheads (o_s and o_r, respectively), the minimum time between successive messages (g), the time to send one byte (G), and the number of bytes sent (k) [17]. This model can be extended to apply to partitioned communication by taking into account the additional overheads introduced by dividing a buffer into multiple messages [18]. For example, Figure 2 shows the model extended to two partitions sent back-to-back; the total time to transmit the data is:

$$o_s + 2G(k-1) + \max(g, o_s, o_r) + L + o_r$$

Without going into details, the model can be further extended to accommodate an arbitrary number of partitions, and take into account opportunities afforded by partitioned communication for initiating data movement early relative to traditional sends. Provided parameter values for a given network, it is thus possible to use the model to select a number of partitions that maximizes communication performance relative to traditional communication, as discussed below. Throughout this paper we will refer to the Partitioned LogGP model as the **PLogGP** model for brevity.

III. Motivation

Due to the relatively new addition of MPI Partitioned to the MPI standard, there has yet to be a thorough investigation into optimizing MPI library support for partitioned communication. In the MPI-4.0 standard [13], the authors provide the following advice to MPI library implementers:

> *"It is expected that an MPI implementation will attempt to balance latency and aggregation for data transfers for the requested partition counts on the sender-side and receiver-side to allow optimization for different hardware. A high quality implementation may perform significant optimizations to enhance performance in this way; they may, for example, resize the data transfers of the partitions to combine partitions ... in a single data transfer"*

This advice outlines some potential design paths to take in improving the performance of MPI Partitioned. In this work, we examine how we can directly use network hardware resources and how message aggregation can be used in the context of MPI Partitioned to maximize communication performance.

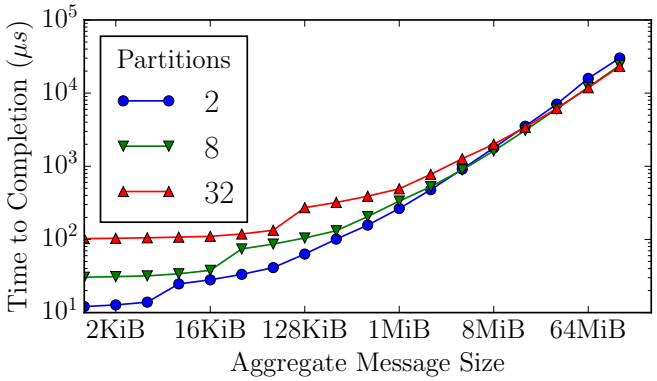


Fig. 3: Modelled time to completion of MPI Partitioned point-to-point communication using the PLogGP with different partition counts, we use an delay time of $4ms$.

To verify if this advice was applicable to the Niagara Supercomputer (see Section V-A for system details), we used Netgauge [19] to collect LogGP parameters [17]. We used Netgauge's MPI module to obtain our measurements, the configuration of the Open MPI + UCX library used can be seen in Section V-A. The InfiniBand module would have been more appropriate to our work as we would obtain the LogGP parameters directly from our network hardware. However, it is listed as an 'experimental InfiniBand implementation' and we were unable to get it to work on our platform.

We fed the measured LogGP values into the PLogGP model to see the impact of different partition counts on this system [18]. The result of this modelling can be seen in Figure 3. Prior work on MPI partitioned [8], [14], has used 100ms of computation with 4% noise applied to a single thread to mimic a realistic use case. Therefore, we have used 4ms of delay to our laggard thread in this model to reflect prior work, as well as the benchmarking results we present in Section V. For small and medium messages, we observe that larger partition counts (i.e. 32) take longer to transfer data than fewer partition counts. However, for large messages we can see that the model favours larger partition counts. This is expected, as a common MPI optimization is to use message combining to improve the performance of smaller-medium messages and message splitting for large messages.

As with any model, there are limitations in the information it produces to provide simplicity to its user. Therefore, we can expect that these general trends to be present in the MPI library that we will design in this paper but the exact thresholds may not match. In this paper we will explore those thresholds to provide advice to other MPI Partitioned implementers.

IV. Design

In our design we explored how the InfiniBand Verbs API can be used by the MPI Partitioned interface without passing through a middleware library such as Unified Communication X (UCX) [20]. We first present the common parts of our design in Section IV-A. Then we explain our three different aggregation approaches: Tuning Table Aggregator, PLogGP Aggregator, Timer-based PLogGP Aggregator.

A. MPI Partitioned over InfiniBand

When programming with MPI Partitioned, `MPI_Psend_init` and `MPI_Precv_init` are used to initialize communication between MPI process pairs. Therefore, within those function calls we; create our device contexts, if one does not exist, allocate our PD, register our MRs (i.e. register our send and receive buffers), create and begin exchanging our QPs, and create our CQ. In this process, we also bring our NICs to the Ready to Send (RTS) state and Ready to Receive (RTR) state for the sending and receiving processes respectively.

In this paper, we define **User Partitions** to refer to the partitions that a user will interact with, and **Transport Partitions** as partitions that the MPI library will send over the network. A user will not have access to the transport partitions other than any environment variables we create for fine-tuning of our library. In Figure 4, we present a high level overview of how we map user partitions to our network resources. As we are working with InfiniBand network cards, we will create a WR for each transport partition and post them onto our QPs. When we refer to *aggregation* in this paper, we mean that multiple user partitions are sent in a single WR, we are not staging the data in another buffer. The hardware that we use for our evaluation is limited to 16 concurrent RDMA WR per QP. Although we could design a mechanism to limit the number of transactions posted on the queue at any one time, we opted to use multiple QPs as we can use up to 262,144 on the ConnectX-5 network cards. The focus of this paper is on medium messages, therefore we have not explored using features such as inlining or BlueFlame which improves the latency of small messages. However, our evaluation (*see* Section V) includes comparisons to Open MPI + UCX, which does leverage these features.

The QP exchange and bringing the NIC to the correct state are done asynchronously to adhere to the non-blocking semantics of `MPI_Psend_init` and `MPI_Precv_init`. However, we do not know that the NIC are in the correct state when we call `MPI_Pready`. One approach, proposed by the MPI forum is to create an new function `MPI_Pbuf_prepare` which would provide remote buffer readiness guarantees from MPI [21]. To get around this issue, we currently poll the progress engine in `MPI_Start` until the remote buffer is ready. This only occurs for the first round of communication as the remote buffer will remain ready for future rounds. In `MPI_Start` we also post our receive WRs as they are required by `IBV_WR_RDMA_WRITE_WITH_IMM`.

`MPI_Pready` executes an atomic add-and-fetch on an array of flags we have for our transport partitions. This marks the arrival of a user partition. If all user partitions mapped to our transport partitions have arrived, then it will call `ibv_post_send` on the associated WR. This work request uses

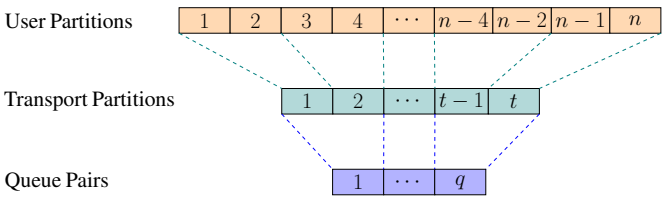


Fig. 4: Mapping of user partitions to queue pairs

the opcode `IBV_WR_RDMA_WRITE_WITH_IMM`, so that we can use the immediate value to encode which user partitions are contained within the transport partition that is being posted. The immediate value must be of type `__be32`. So to encode the required information we store the starting user partition and the number of contiguous partitions as two variables of type `uint16_t`. We shift the bits appropriately so we can store them as the type that is required by the WR.

`MPI_Test` and `MPI_Wait` hook into Open MPI's progress engine, when the request that is being serviced is associated to partitioned communication we call `ibv_poll_cq`. If we obtain a successful WC, we update the receiving user partition flags. `MPI_Parrived` initially checks receiving user partition flag to determine if the partition has arrived. If the queried user partition has not set the flag to acknowledge an arrival, it calls the progress engine. Our progress engine design is single-threaded, we only allow a single thread to progress at a time. `MPI_Parrived` tries to acquire a lock. If it is successful, it will progress all MPI messages and release the lock upon completion. Otherwise it just returns from the function.

Thus far we have discussed the general design of our library, in the next three subsections we present three optimization strategies we have explored to improve our design further.

B. Tuning Table Aggregator

The solution space for searching for an optimal configuration is quite large. It is dependent on the number of process pairs, the number of possible user partitions, the number of possible transport partitions, the number of QPs used, maximum transmission unit (MTU), and the message size. This gives us a very large, six-dimensional search space if we only consider the variables we have mentioned. We searched a subset of this solution space for two processes with a 4KiB MTU, to create a tuning table that provides an optimal configuration of these variables. We ran our tests for 100 iterations and it took just under 23 hours on two nodes. While this exhaustive approach is clearly not scalable to larger process counts, it provides an important baseline to evaluate the accuracy of the model-based approach against.

During initialization, we create a hash table where the key is a tuple *(number of user partitions, message size)*, and the value is a tuple *(number of transport partitions, number of QPs)* for that key. Each time `MPI_Psend_init` and `MPI_Precv_init` are called to initialize a process pair, we query the hash table to determine the optimal number of transport partitions and QPs to use. We then initialize our QPs as described in Section IV-A. As we know the aggregation scheme during initialization we preemptively create the send WRs that will be called by `MPI_Pready`.

C. PLogGP Aggregator

In this design, we use the PLogGP model to create a platform-specific tuning table. We use Netgauge to obtain the required LogGP parameters and build a hash table where the key is the message size and the value is the set of LogGP parameters for the model. Running Netgauge has a significantly lower cost than exhaustively searching the solution space (*cf.* Section IV-B), especially at scale.

The PLogGP model considers different arrival patterns [18], in this paper we focus on the many-before-one scenario, where

TABLE I: Optimal number of transport partitions predicted by the PLogGP model for different message sizes on Niagara

Aggregate Message Size	Transport Partitions
<256KiB	1
512KiB-1MiB	2
2MiB-4MiB	4
8MiB-16MiB	8
32MiB-64MiB	16
>128MiB	32

all but one of the threads finish simultaneously and one is delayed. This is a possible occurrence if the operating system moves a thread, during a programs execution. We implement the many-before-one scenario inside of the Open MPI library. Each time `MPI_Psend_init` and `MPI_Precv_init` are called, we use the message size, the requested number of user partitions, and a delay value as input into PLogGP model. We iterate the model over power of two transport partition counts to arrive at an optimal transport partition count. This model gives us an optimal transport partition count regardless of user partitions requested. If the model suggests a transport partition count that is larger than what the user requested, then we fall back to the user's request. In this design we do not disaggregate (subdivide) user partitions as there is little benefit since the application does not expose any finer-grained communication pattern that can be exploited. Disaggregation would result in issuing more transactions than necessary to network hardware. On the receive side, it would not provide much benefit as we would incur additional completion overheads [9]. In Table I we present a summary of the model for different message sizes on Niagara (*see* Section V-A for details on this system). One restriction we have placed on using this model, to reduce complexity, is that we only consider power of two transport and user partition counts. Our transport partition count is bound between one and the number of user partitions, they are contiguous, and aligned on $\frac{\#n \text{ user parts}}{\#n \text{ transport parts}}$ boundaries. We use the calculated number of transport partitions to initialize our QPs and create our WRs.

D. Timer-based PLogGP Aggregator

In this section, we extend the design outlined in Section IV-C by making our design aware of the partition arrival pattern since it has been shown that considering arrival patterns can greatly improve the performance of MPI libraries [22]. The PLogGP model receives an input value for noise but this value is used once during `MPI_Psend_init`/`MPI_Precv_init` and is not adjusted at run time. An online auto-tuning approach could be used to tune the PLogGP model input delay parameter to adjust the model output. However, this method would be far outside the scope of this paper. In this work, we opt to use a rough user partition aggregation grouping generated from the PLogGP model and we dynamically adjust the grouping based on how the user partitions arrive.

In Figure 5, we can see four user partitions $\{p_0, p_1, p_2, p_3\}$ calling `MPI_Pready` at slightly different times. This figure would represent the user partitions for a single transport partition that the PLogGP model would define. We propose that, the first thread to arrive would sleep for a small amount of time and periodically check a flag to see if it needs to wake. The thread will sleep for a maximum amount of time defined by a δ value. Upon wake, if all `MPI_Pready`'s have been called (e.g. $\delta = \delta_a$ in Figure 5), the first thread to arrive does nothing, as the last thread

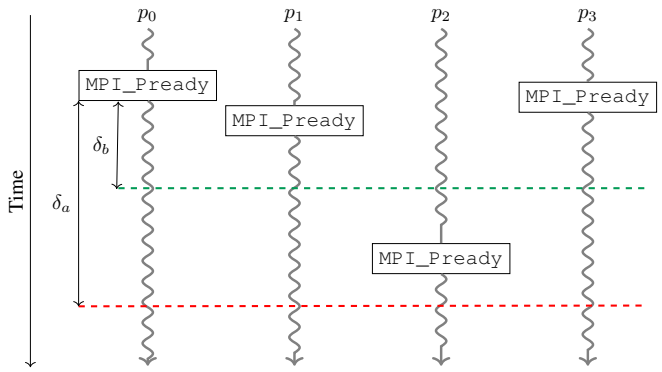


Fig. 5: Diagram presenting a possible user partition arrival pattern. We also illustrate how our aggregator could selectively choose to which user partitions we aggregate based on our δ value

to arrive would have aggregated and sent all the user partitions as defined in Section IV-C. Providing a δ value that is too large results in this design providing no additional benefit to the original PLogGP aggregator while also introducing an additional delay into an another thread. However, if our $\delta = \delta_b$, then our user partitions above the green line would be aggregated into their largest contiguous user partitions and sent. So in this example, p_0 would execute two RDMA write operations containing user partitions $\{0, 1\}$ and $\{3\}$. Once p_2 arrives, after δ_b, it would send the remaining user partition $\{2\}$. The ideal scenario is aggregating all user partitions which have arrived into a single WR, and then a second for the laggard thread. However, we still issue fewer RDMA writes than no aggregation at all. We did consider using scatter/gather support for noncontinuous data but it is only supported on the sending side for RDMA write operations. This would require data to be staged on the receiving side as well as providing the receiver with information about the data layout.

V. Performance Results and Analysis

A. Experimental Setup

Experiments were conducted on the Niagara cluster at the SciNet HPC Consortium [23]. Each Niagara node has two sockets with 20 Intel Skylake cores at 2.4GHz, for a total of 40 cores and 188GB of RAM per node. The 2024 nodes are connected using an EDR InfiniBand network in a Dragonfly+ topology. Niagara uses the GNU/Linux distribution CentOS 7.6. In this section we refer to the persistent implementation as using the `part_persist` Modular Component Architecture (MCA) module of MPI Partitioned in Open MPI (5.0.x) using UCX v1.12.1. For our three proposed designs, we use use the same Open MPI version but we create our own MCA module that directly uses the InfiniBand Verbs API.

We modified the public benchmarks listed in [14], to use Open MPI rather than the MPIPCL [24], which was used in that work. These benchmarks assign one user partition to each thread and only evaluate power of two user partition counts. For the point-to-point benchmarks, we ran each job for 10 warm-up iterations, then we obtained our measurements over 100 iterations. For the sweep benchmarks, each job used three warm-up iterations and we obtained our measurements over 10 iterations. Fewer trials were used for the sweep benchmarks as

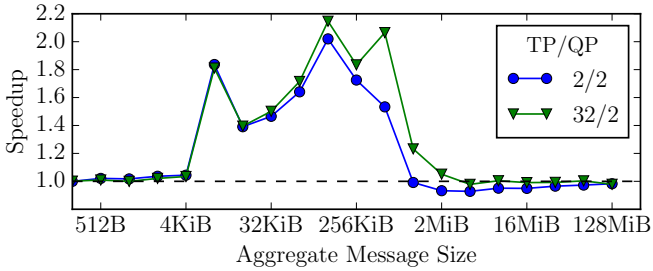


Fig. 6: Overhead benchmark for 32 user partitions comparing the number of transport partitions

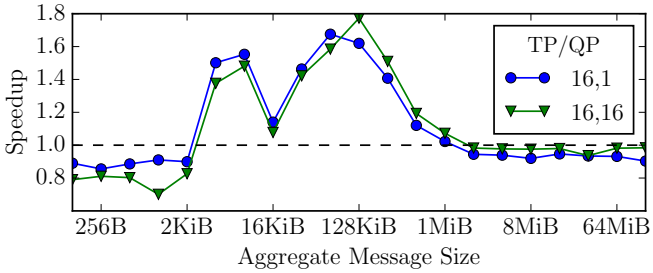


Fig. 7: Overhead benchmark for 16 user/transport partitions comparing the number of QPs

they are more computationally intensive as they run on multiple nodes and we were required to keep within our compute allocation budget. We submitted a minimum of three jobs for each test and on our figures we present the mean of those job submissions. Throughout this paper we present our results with compute amounts of 1ms or 100ms and with noise values of 1% or 4%. These values were chosen to evaluate MPI Partitioned using the same parameters as prior work in this area [8], [14].

For our profiling results, we created our own MPI Partitioned profiler[1] built upon the MPI Profiling interface (PMPI) to collect the data that is presented.

B. Overhead of Aggregation

In this section, we used a modified version of the overhead benchmark from [14], to evaluate the wire efficiency of our MPI Partitioned module and compare it to the implementation of partitioned communication in OpenMPI. We give some initial results that compares different mappings of WR and QPs to user partitions. Then we compare our Tuning Table Aggregator and PLogGP Aggregator designs. We do not investigate our Timer-based PLogGP Aggregator in this subsection as we not using noise, thus no partition imbalance are present in these tests. We will evaluate that design in Section V-C as the perceived bandwidth benchmark is more appropriate.

1) Mapping WRs and QPs to Partitions: In Figure 6 and Figure 7, we present results to begin our exploration into different mappings mechanisms. The speedup is shown relative to Open MPI's persistent communication module. Although, we were able to recreate the modelling results of the PLogGP model, we want to verify we saw the same behaviour if we used it to tune an MPI library.

We ran the overhead benchmark while keeping the number of QPs constant and varying the number of transport partitions in Figure 6. Using fewer transport partitions does yield better performance for small messages. However, there is around 0.16% to 1.77% difference between using two and 32 transport partitions up to 8KiB. For small messages, we cannot be conclusive that there is any impact to varying transport count, while using two QPs. Therefore, the difference between the two partition counts is not as large as we would have expected, from the PLogGP modelling results. The source of the discrepancy between the PLogGP model and our results could stem from a few places. For example, we obtain our LogGP parameters from Netguage using the MPI transport but use the values to estimate a InfiniBand Verbs transport. The PLogGP model does not consider QPs. Finally, we also do not use the small message features of InfiniBand hardware such as inlining and BlueFlame. This does not detract from the work in this paper, as our focus is for medium sized messages since they have been shown to be the most important message range for MPI Partitioned (see [8], [14] and Section V-C). After 16KiB, more transport partitions are favourable. This is expected as Table I shows that, as message size grows the PLogGP model suggests using more partitions. Once we reach around 4MiB we drop to a speedup of 1.0, this is due to the saturation network hardware bandwidth.

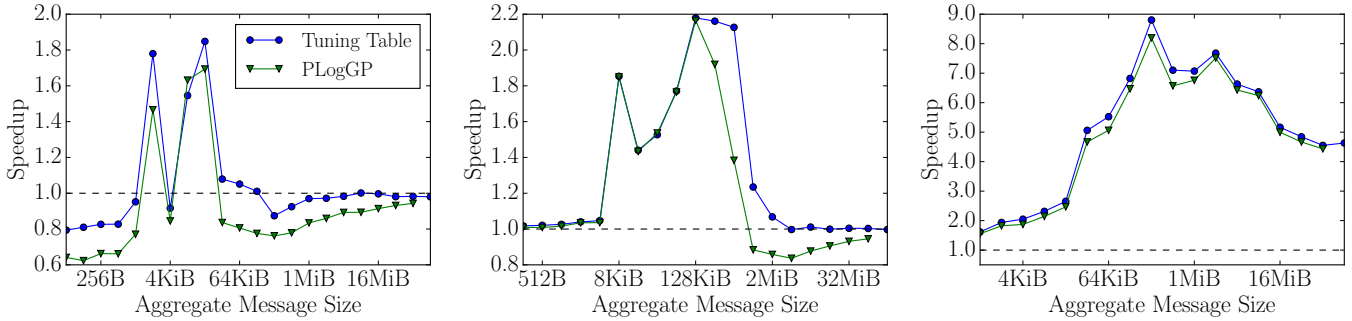
In Figure 7, we also investigate the impact of QPs, since this is not accounted for in the PLogGP model. When no aggregation is used for 16 partitions, a single QP is sufficient until around 64KiB then using 1 QP per partitions performs better. We suspect this is due to large messages preferring more concurrency when possible.

For brevity, we have not included all results but we did notice the exact cut off points varied with number of user and transport partitions that we request. However, the static tuning table we created from that data presented the same trends as the PLogGP model of more transport partitions as message size increases.

2) Evaluation of Different Aggregation Mechanisms: In Figure 8, we compare our tuning table aggregator and our PLogGP aggregator designs for different user partition counts. With four user transport partitions, our aggregators do not perform well for messages below 2KiB and above 32KiB when compared to the persistent implementation in Open MPI. This leaves a very narrow range where there is some performance benefit. In this message range, the performance difference between the tuning table and PLogGP approach is at most 8.73%. The spikes shown in our data are from calculating the speedup over a protocol switch that occurs in Open MPI + UCX. For example, in Figure 8, there is a dip at 4KiB (where individual transport partition size is 1KiB). 1KiB is the threshold where UCX switches from its eager/bcopy to its eager/zcopy protocol. As the overhead benchmark has no thread imbalance, this suggest that using aggregation may not be the most beneficial for an application with only a few balanced threads present.

For both 32 and 128 user partitions, we see a significant speedup over the persistent implementation, which indicates that aggregation is favourable when there are more user partitions.

[1]https://github.com/Yiltan/MPI-Partitioned-Profiler



(a) 4 User Partitions (b) 32 User Partitions (c) 128 User Partitions

Fig. 8: Overhead benchmark for different user partitions counts comparing the tuning table aggregator to the PLogGP aggregator

At 32 user partitions, the difference is minimal and ranges between 0.21-1.22% until 128KiB where we observe our peak speedup of 2.17x over the persistent implementation. After 512KiB, our speedup drops as we begin to reach the bandwidth limits of the network hardware. With the larger messages the discrepancy between the two model can be quite large. At 512KiB the tuning table approach achieves 2.13x speedup and 1.38x for PLogGP model. However as the message sizes grow, the difference between the two models shrink.

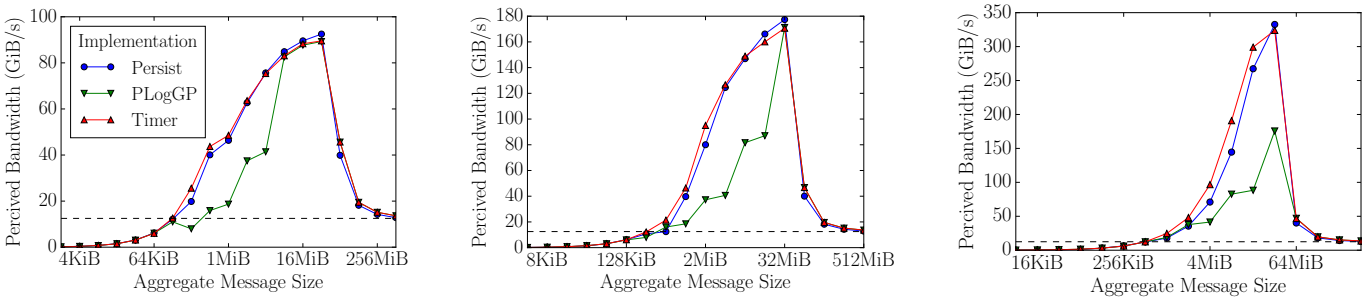
With 128 user partitions, we see up to 8.80x speedup for our implementations, but there is some discrepancy between approaches (2.02-9.10%). However, we always better than our baseline by a considerable amount. This is because we have over-subscribed the number of threads on our system. Message aggregation in this scenario reduces the amount of lock contention to post a WR. So we suspect that aggregation could be beneficial to workloads that have higher user partition counts as well as those that require oversubscription.

For all three user partition counts presented in Figure 8, using the Tuning Table Aggregator and the PLogGP Aggregator generally follow similar trends. This shows PLogGP model predicts reasonable results, which can be relatively close to doing a brute force search on the solution space, while requiring much less compute to calculate ideal user partitions groupings. For the remainder of this paper we will focus on the PLogGP aggregator, to minimize the compute resource that would be required to create a new tuning table for each test and communication pattern we evaluate.

C. Perceived Bandwidth

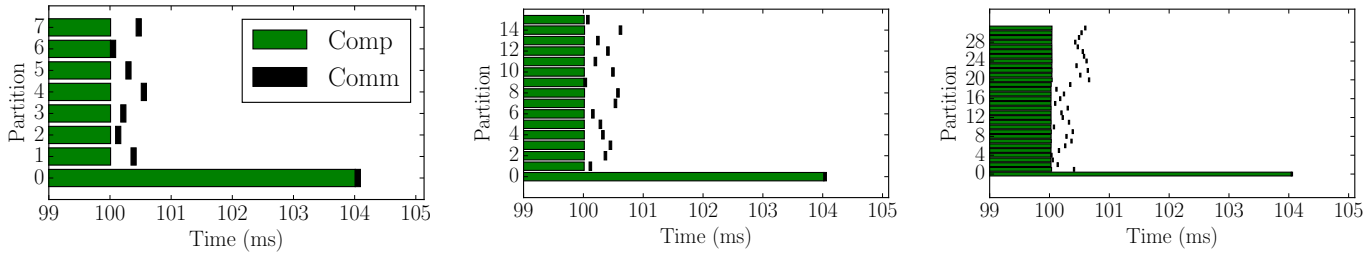
In this subsection, we use the perceived bandwidth benchmark from [14], [8]. This micro-benchmark is useful for measuring thread imbalance, as we inject noise in to our tests, unlike the overhead benchmark. This shows us how well we could tolerate a thread imbalance in our design, as well as, what the effective network bandwidth we would require from the network hardware, if we used a single-threaded MPI point-to-point implementation rather than MPI Partitioned to achieve the same performance.

1) Evaluation of Different Aggregator Designs: In Figure 9, we present a comparison of the persistent implementation, our PLogGP aggregator, and our Timer-based PLogGP aggregator. The value of $\delta = 3000\mu s$ in this figure is arbitrary and is used here for illustration. Choosing an appropriate δ is discussed in Section V-C3. The persistent implementation provides some the highest perceived bandwidth results. This is expected as the persistent implementation provides no aggregation. In this benchmark, we measure the latency to send the last partition and divide it by the total buffer size. A lack of aggregation minimizes the latency for the last user partition which results in a higher perceived bandwidth. Our PLogGP aggregator still provides additional performance improvement as the perceived bandwidth is above the hardware limits for a single-threaded MPI point-to-point communication (shown by the dotted line). However, it still falls short of the persistent implementation in Open MPI. This is especially true for 32 Partitions. We believe this stems from the aggregator combining multiple messages, which effectively



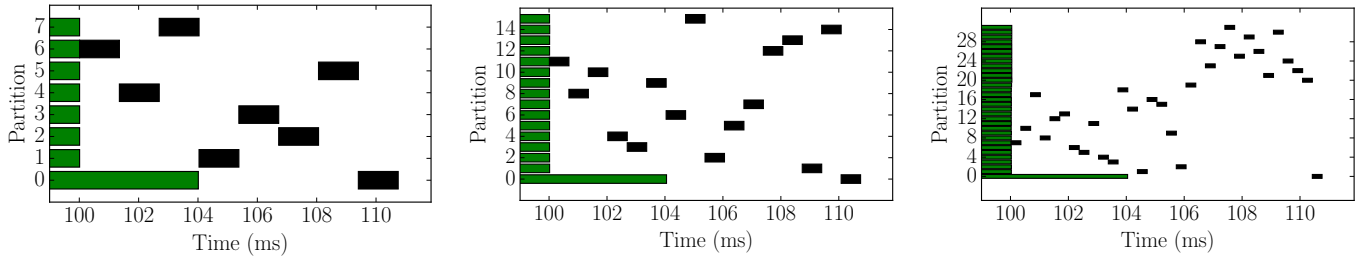
(a) 8 User Partitions (b) 16 User Partitions (c) 32 User Partitions

Fig. 9: Comparison of our different aggregator designs using the perceived bandwidth benchmark. 100ms compute, 4% noise, and the single thread delay model is used. For the Timer-based PLogGP aggregator, we use a value of $\delta = 3000\mu s$



(a) 8 Partitions (b) 16 Partitions (c) 32 Partitions

Fig. 10: Profiling the user partition arrival pattern of the perceived bandwidth benchmark. 8MiB, 100ms compute, 4% noise, and the single thread delay model is used



(a) 8 Partitions (b) 16 Partitions (c) 32 Partitions

Fig. 11: Profiling the user partition arrival pattern of the perceived bandwidth benchmark. 128MiB, 100ms compute, 4% noise, and the single thread delay model is used. This figure has the same legend as Figure 10 but it has been omitted for space

increases the message size of the last transport partition and thus the time taken for that last user partition to be sent. This increase in latency results in a lower perceived bandwidth result.

The Timer-based PLogGP aggregator, improves the shortcoming of the PLogGP aggregator as it performs much closer to the persistent implementation. This aggregator does not delay all user partitions before transmission, it sends as many as it can before the laggard thread, to utilize the idle network. Therefore, the library still views the last user partitions as a single user partition so that it does not hold back others. This balances aggregation and latency, as we are able to have fewer RDMA writes but maintain good latency for the last user partition.

2) Profiling Partition Arrival Patterns: To further understand why the Timer-based PLogGP aggregator and the persistent implementation perform best, we profiled the benchmark to visualize the user partition arrival pattern. We developed the profiler using the PMPI interface to measure the time the program arrives at `MPI_Start`, and at each `MPI_Pready` call. From this information we calculated the computation time as shown in green in Figure 10 and Figure 11. The communication time was estimated from the theoretical network bandwidth of our hardware. For each user partition, we calculated the estimated communication time using the basic bandwidth equation $comm_n = \frac{\textit{user partition size}}{\textit{bandwidth}}$. As each user partition would arrive, we would append the communication time to the plot, shown in black.

As we used the single thread delay model in this benchmark, we clearly see that most user partitions arrive within a reasonable time frame, apart from the laggard thread. For "medium" message sizes, such as in Figure 10, we can complete the data transfer of $n-1$ user partitions before our last thread arrives. This provides us a large perceived bandwidth as the application only perceives the communication time of a single user partition.

The possible range of δ values for our Timer-based PLogGP aggregator is relatively large, just over $3000\mu s$ would be sufficient to communicate all user partitions early and benefit from early bird-communication. However, it is clear that our δ value could be much smaller while providing sufficient performance. Although this value has some flexibility for these point-to-point benchmarks, in a scenario where a process communicates with multiple processes, like an application, this would be more impactful. For example if a single thread calls `MPI_Pready` multiple times for different requests, a large δ would introduce an artificial delay that would impact communication requests other than the one that the MPI library is acting on.

For "larger" messages as shown in Figure 11, we are unable to communicate all our early user partitions before our last partition arrival. For 128MiB we send roughly $\frac{3}{8}$ of our user partitions early as we are limited by the actual hardware bandwidth. Our PLogGP aggregator is required to wait for all user partitions to arrive before it communicates. Therefore, this could result in $\frac{3}{8}$ not being sent early and thus not efficiently using network bandwidth. However, the benefit of early-bird communication in this case is minimal in comparison to medium sized messages. This is also reflected in Figure 9, as we see our perceived bandwidth drop closer to our theoretical max (the dotted line) for large messages. We are now sending multiple user partitions after our deadline and are network limited.

3) Is There an Optimal Delta Value?: After obtaining our profiling result, we asked ourselves if there is an optimal δ value, we could find. We used the profiling data to compute a minimum δ value that would cover most user partitions. For each message size and partition count, we obtained the average arrival time for each partition that was not the laggard thread. Then we obtained our minimum δ by calculating the difference between the first and last

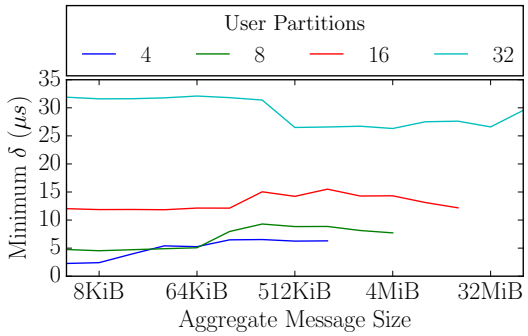


Fig. 12: Estimated minimum δ value for our Timer-based PLogGP aggregator

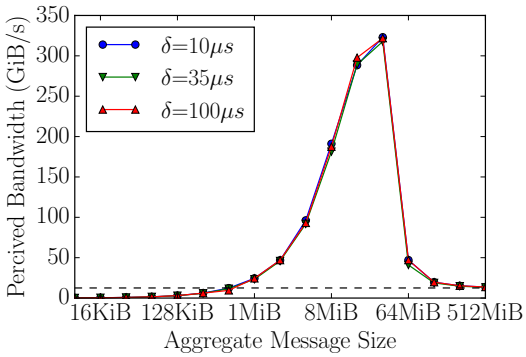


Fig. 13: Perceived bandwidth using a window that falls around our minimum δ value for 32 user partitions

(non-laggard) thread to arrive. The results can be seen in Figure 12. The missing data points in Figure 12 are due to the PLogGP model requesting no aggregation for those message sizes. The trends are somewhat expected, as the user partition count grows, the minimum δ value grows. For large user partition counts we would expect the aggregator to wait for more threads which would result in larger imbalances as the must take turns to increment the atomic counter that we use to keep track of partition arrival.

In Figure 12, we can observe that for 32 Partitions, a minimum δ value of $35\mu s$ should be sufficient to aggregate most user partitions that are not the laggard thread. So in Figure 13, we chose two δ value above and below $35\mu s$ to evaluate how well our estimation would work. The difference between $\delta = 10\mu s$, $\delta = 35\mu s$, and $\delta = 100\mu s$ is at most 6.15%. So while picking an optimal number is important, as user can be off by 3.5x and the performance penalty is not to the same extent. For the value bellow our minimum we have to consider that the aggregator is not instantaneous so the user partitions may arrive while it is processing. With the larger δ value, the impact of waiting slightly longer is small. We saw in Figure 10 that the room for sending data is quite large and our Timer-based PLogGP aggregator can tolerate a δ value that is a little too large than the optimal value. With the larger message size of 128MiB, we saw that those data sizes are network limited in Figure 11, so having a $\delta = 100\mu s$ value that is a little too large has minimal impact when the latency of a 128MiB is around $10ms$ and a very small fraction of time comparatively.

D. Communication Pattern Results

Thus far we have only investigated point-to-point micro-benchmarks, so in this section we evaluate a Sweep3D communication pattern which is commonplace in HPC applications. In Figure 14, we apply our PLogGP and Timer-based PLogGP designs to this communication pattern to gain insight on how these optimizations could benefit applications. In these tests, we subtract the computation time listed in each subfigure caption from the round of communication, so that we present only the speedup of the communication portion of our code.

We evaluate our design with different amount of computation and noise values and we generally see that our aggregation mechanisms predominantly benefit small-medium size messages. At 1MB We see up to 1.60x, 1.63x, and 1.04x for Figure 14a, Figure 14b, and Figure 14c respectively. Those experiments have $10\mu s$, $40\mu s$, and $400\mu s$ of noise. The speedup for those message sizes is related to the amount of compute and noise. For $400\mu s$ of noise our our speedup is reduced as the percentage of noise contributes to a larger delay time for the laggard thread. This results in a larger total communication time, which is also shown in Figure 10 when the computation is larger than the original communication time. In Figure 14a and Figure 14b the noise is still sufficiently small that we can benefit from MPI Partitioned's early bird communication.

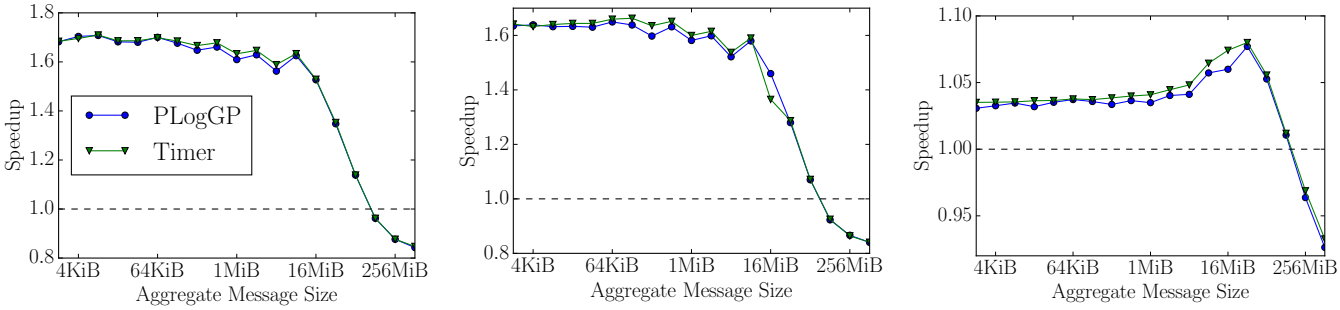
In Figure 14, we also evaluated our PLogGP and Timer-based PLogGP aggregation mechanisms. Timer-based design provides an additional speedup when we consider the arrival pattern of threads for medium messages. For larger messages, the two designs perform comparably as we are now limited by network bandwidth rather than the computation time. The difference between the two designs are minimal for small messages but becomes more noticeable for medium messages. That is largely due to the benefit of early-bird transmission of small messages are minimal as shown in our perceived bandwidth tests in Figure 9

The performance for very large messages with both approaches does not provide any speedup. As we have previously noted, MPI Partitioned is aimed at medium sized messages. If an application were to use those very large messages, traditional MPI point-to-point communication would be more suitable than MPI Partitioned.

E. Related Work

Multi-threaded MPI communication faces many challenges with regards to good performance at scale. MPI Endpoints was initially proposed in the MPI Forum, where it suggested that assigning ranks to threads could be one possible solution to handling hybrid MPI codes [7], [25]. This approach had some issues as it would increase the rank space by the number of threads, and it does not necessarily avoid some of the issues present with multi-threaded MPI from an implementation perspective. In [2], the authors explored the implementation trade-off between performance and resource usage on Mellanox InfiniBand hardware for MPI Endpoints. Our work differs are we are evaluating the trade-offs within the context of MPI Partitioned. MPI Partitioned does not require all threads to access hardware, only to mark that the data is ready. This allows us to explore user partition aggregation, which presents different limitations on Mellanox hardware for MPI implementers.

MPI Partitioned Point-to-Point, which was eventually standardized, was first introduced in [26]. In that work, the authors



(a) 1ms Comp, 1% Noise, $\delta = 5\mu s$ (b) 1ms Comp, 4% Noise, $\delta = 20\mu s$ (c) 10ms Comp, 4% Noise, $\delta = 100\mu s$

Fig. 14: Speedup in communication time (computation time not included), Open MPI's persistent implementation compared to our PLogGP and Timer-based PLogGP designs. We ran the benchmark on 1024 cores (16 threads x 64 nodes)

describe the need for threads to easily commit data that is ready for transfer without the MPI run-time acquiring internal locks. They propose partitioning a buffer during initialization and sending data as needed or once all data is ready. At this time, many of the final features had yet to be proposed but this work laid the foundations for MPI Partitioned. In [8], the authors present a two-sided MPI interface that is much closer to the final MPI Partitioned interface. They formally describe that agreeing on message size and partition count, and message matching should occur with the initialization functions. Here is where we first see the `MPI_Pready` function to commit data. They also describe how this interface can be implemented using triggered operations on existing hardware. However, the implementation presented in this paper was on top of the MPI library using the RMA interface. During their evaluation, they show how preemptively sending data can yield performance improvements for finite elements codes. That work did not include receive-side partitions in its proposal but these were first seen in [9]. In their evaluation, they found the overhead of checking for arrival of data by the receiver was minimal and actually performed better when using more threads. They were also able to perform some additional receive side computation using the `MPI_Parrived` interface.

The work discussed so far has been about how MPI Partitioned Point-to-Point communication came to be. In [10], the authors propose a portable interface built on top of existing MPI libraries. They mention the necessity of a different request object which can store additional information that is specific to MPI Partitioned, such as partition count and which partitions are ready. Their design also used a helper thread to maintain the non-blocking semantics of `MPI_Psend_init`/`MPI_Precv_init`/`MPI_Start`. This differs from our design where we wait on `MPI_Start` to ensure that our RDMA buffers are ready. This was possible as they relied on existing MPI point-to-point communication rather than one-sided communication. Performance result for this portable interface was shown in [12] where it was also compared to an implementation within the Open MPI library. The Open MPI implementation used in this paper is what eventually become the persistent implementation used in this paper. There was minimal difference between the layered library approach and the Open MPI persistent MCA module. This library was further evaluated in [11] where they compare persistence send/receive implementation to an RMA implementation. They found that an RMA implementation provides some additional performance benefits compared to the persistent implementation. However, an RMA implementation still has limitations as it is limited by a software API. A software-based RMA implementation does does not provide access to fine grain RDMA hardware features to further optimize but it does have the benefit of being portable.

The authors of [14] noticed that despite the many papers investigating MPI Partitioned there was no standardized set of micro-benchmarks available to the public. Therefore, they developed their own public set of benchmarks for MPI developers to use. Alongside the micro-benchmarks they evaluated common communication patterns used by MPI applications such as a halo exchange and a sweep3d. These micro-benchmarks and the sweep3d pattern is used throughout this paper to evaluate our designs.

The paper which presents the PLogGP model used in this paper [18], validated the model different hardware platforms. However, they did not present a use case for the model with results. In this paper we address that by providing an MPI implementation which can benefit from the model.

Optimizing MPI libraries using RDMA hardware features has been present for many years [27]. The authors of [27] investigated how an RDMA design can reduce MPI latency. Aggregation for the improvement of RDMA traffic in Open MPI was performed [28] where they find that aggregation is beneficial when ordering of messages is not needed. Therefore, the method in [28] is only useful for MPI RMA types of traffic.

VI. Conclusion

As modern HPC systems continue to grow in terms of CPU-core counts and heterogeneity, the need for efficient hybrid programming models to use these system effectively is increasingly important. Extensions to long standing programming models such as MPI have adapted to accommodate next generation hardware by providing new APIs such as the MPI Partitioned interface.

We present one of the first works on detailed network low-level design optimizations for an MPI Partitioned implementation. We initially provide a method to map MPI Partitioned interface to the InfiniBand Verbs API in a logical manner as an example of how an MPI implementer may want to design their library. We also investigate the aggregation of user partitions and how we can efficiently send them over the network. We combine our user partitions into our transport partitions using a brute force approach and using the PLogGP model. It was shown that using the PLogGP model to predict an aggregation scheme with reasonable

results while using much less compute time than the brute force approach to arrive at a reasonable answer. This design was further optimized by considering the pattern in which user partitions call `MPI_Pready` to dynamically modify our aggregation scheme. We profiled our micro-benchmarks to provide analysis on how and why this additional optimization was beneficial to our results and how we can fine-tune this mechanism. Finally, we evaluated our PLogGP and Timer-based PLogGP designs with a commonly used communication pattern in HPC (communication sweep) to observe the impact when communicating with multiple processes in an application-like scenario.

A. Future Work

This paper focuses on medium messages sizes but there are InfiniBand hardware features such as BlueFlame or inlining which are commonly used for small messages that could be studied to further optimize an MPI design. Continuing with the work presented, in this paper, by evaluating our designs on real and proxy-applications could be incredibly beneficial to the community. At this time, there are currently no production-grade applications which use MPI Partitioned. Therefore, this requires an application porting effort which is outside the scope of this work. In [29], the authors identify possible application that could benefit from a porting effort.

Application of this work to novel accelerators and network cards could be advantageous for next generation systems. We could investigate how we could offload our aggregation design onto a DPU to remove the burden of aggregation and progression off of the host. It is also important to consider GPUs as they are actively discussed in the MPI forum as well as FPGA which are still in their infancy with regards to MPI Partitioned support [30].

ACKNOWLEDGEMENT

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada Grant RGPIN 05389-2016 and Compute Canada. Computations were performed on the Niagara supercomputer at the SciNet HPC Consortium. SciNet is funded by: the Canada Foundation for Innovation; the Government of Ontario; Ontario Research Fund - Research Excellence; and the University of Toronto.

REFERENCES

[1] (2023) Message Passing Interface. [Online]. Available: http://www.mpi-forum.org

[2] 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.

[3] D. E. Bernholdt, S. Boehm, G. Bosilca, M. Gorentla Venkata, R. E. Grant, T. Naughton, H. P. Pritchard, M. Schulz, and G. R. Vallee, "A survey of MPI usage in the US exascale computing project," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 3, pp. 1–16, 2020, e4851 cpe.4851. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4851

[4] D. Goodell, P. Balaji, D. Buntinas, G. Dózsa, W. Gropp, S. Kumar, B. R. de Supinski, and R. Thakur, "Minimizing MPI Resource Contention in Multithreaded Multicore Environments," in *2010 IEEE International Conference on Cluster Computing (Cluster)*, 2010, pp. 1–8.

[5] M. G. Dosanjh, R. E. Grant, W. Schonbein, and P. G. Bridges, "Tail queues: A multi-threaded matching architecture," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 3, pp. 1–13, 2020, e5158 cpe.5158. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5158

[6] R. Zambre, A. Chandramowliswharan, and P. Balaji, *How I Learned to Stop Worrying about User-Visible Endpoints and Love MPI*. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3392717.3392773

[7] J. Dinan, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur, "Enabling MPI Interoperability through Flexible Communication Endpoints," in *Proceedings of the 20th European MPI Users' Group Meeting*, ser. EuroMPI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 13–18. [Online]. Available: https://doi.org/10.1145/2488551.2488553

[8] 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.

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

[10] 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.

[11] 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

[12] 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

[13] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard Version 4.0," 2021. [Online]. Available: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[14] Y. H. 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

[15] A. Amer, H. Lu, Y. Wei, P. Balaji, and S. Matsuoka, "MPI+Threads: Runtime Contention and Remedies," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 239–248. [Online]. Available: https://doi.org/10.1145/2688500.2688522

[16] "InfiniBand Trade Association." [Online]. Available: https://www.infinibandta.org/

[17] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "LogGP: Incorporating Long Messages into the LogP Model for Parallel Computation," *Journal of Parallel and Distributed Computing*, vol. 44, no. 1, pp. 71–79, 1997. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731597913460

[18] W. Schonbein, S. Levy, M. Dosanjh, W. 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.

[19] T. Hoefler, A. Lichei, and W. Rehm, "Low-Overhead LogGP Parameter Assessment for Modern Interconnection Networks," in *Proceedings of the 21st IEEE International Parallel & Distributed Processing Symposium, PMEO'07 Workshop*. IEEE Computer Society, Mar. 2007.

[20] 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.

[21] R. E. Grant, "Synchronization on Partitioned Communication for Accelerator Optimization." [Online]. Available: https://github.com/mpi-forum/mpi-issues/issues/302

[22] P. Alizadeh, A. Sojoodi, Y. Hassan Temucin, and A. Afsahi, "Efficient Process Arrival Pattern Aware Collective Communication for Deep Learning," in *Proceedings of the 29th European MPI Users' Group Meeting*, ser. EuroMPI/USA'22. New York, NY, USA: Association for Computing Machinery, 2022, p. 68–78. [Online]. Available: https://doi.org/10.1145/3555819.3555857

[23] M. Ponce, R. van Zon, S. Northrup, D. Gruner, J. Chen, F. Ertinaz, A. Fedoseev, L. Groer, F. Mao, B. C. Mundim, M. Nolta, J. Pinto,

M. Saldarriaga, V. Slavnic, E. Spence, C.-H. Yu, and W. R. Peltier, "Deploying a Top-100 Supercomputer for Large Parallel Workloads: The Niagara Supercomputer," in *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)*, ser. PEARC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3332186.3332195

[24] "MPIPCL." [Online]. Available: https://github.com/mpi-advance/MPIPCL

[25] J. Dinan, R. E. Grant, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur, "Enabling Communication Concurrency through Flexible MPI Endpoints," *Int. J. High Perform. Comput. Appl.*, vol. 28, no. 4, p. 390–405, Nov. 2014. [Online]. Available: https://doi.org/10.1177/1094342014548772

[26] 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

[27] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda, "High Performance RDMA-Based MPI Implementation over InfiniBand," in *Proceedings of the 17th Annual International Conference on Supercomputing*, ser. ICS '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 295–304. [Online]. Available: https://doi.org/10.1145/782814.782855

[28] N. Hjelm, M. G. 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*, 2018, pp. 1–11.

[29] 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," 2023.

[30] 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.