Micro-Benchmarking MPI Partitioned Point-to-Point Communication

Yıltan Hassan Temuçin
Department of Electrical and Computer Engineering
Kingston, ON, Canada
yiltan.temucin@queensu.ca

Ryan E. Grant
Department of Electrical and Computer Engineering
Kingston, ON, Canada
ryan.grant@queensu.ca

Ahmad Afsahi
Department of Electrical and Computer Engineering
Kingston, ON, Canada
ahmad.afsahi@queensu.ca

ABSTRACT

Modern High-Performance Computing (HPC) architectures have developed the need for scalable hybrid programming models. The latest Message Passing Interface (MPI) 4.0 standard has introduced a new communication model: MPI Partitioned Point-to-Point communication. This new model allows for the contribution of data from multiple threads with lower overheads than with traditional MPI point-to-point communication. In this paper, we design the first publicly available micro-benchmark suite for MPI Partitioned to measure various metrics that can give insight into the benefits of using this new model and scenarios where MPI point-to-point is better suited. Suggestions are provided to application developers on how to choose partition size for their application based on compute and message size. We evaluate MPI Partitioned communication with both a hot and cold CPU cache, system noise with different probability distributions, point-to-point communication directly, and with commonly used MPI communication patterns such as a halo exchange and Sweep3D.

CCS CONCEPTS

• **Computing methodologies** → **Parallel programming languages**; • **Networks** → *Network performance analysis*; Programming interfaces.

KEYWORDS

MPI, Partitioned Communication, Micro-Benchmarking, Halo Exchange, Sweep3D

ACM Reference Format:
Yıltan Hassan Temuçin, Ryan E. Grant, and Ahmad Afsahi. 2022. Micro-Benchmarking MPI Partitioned Point-to-Point Communication. In *51st International Conference on Parallel Processing (ICPP '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3545008.3545088

ICPP '22, August 29-September 1, 2022, Bordeaux, France

ACM ISBN 978-1-4503-9733-9/22/08…$15.00
https://doi.org/10.1145/3545008.3545088

1 INTRODUCTION

Applications using the Message Passing Interface (MPI) [3] programming model have traditionally used one process per CPU core. Over the past decade, there has been an increase in core count, but this "MPI everywhere" model is not always optimal for performance. Although the MPI standard supports multi-threaded communication, most MPI implementations and applications are poorly optimized.

Currently, the MPI standard supports three different threading modes: `MPI_THREAD_FUNNELLED`, `MPI_THREAD_SERIALIZED`, and `MPI_THREAD_MULTIPLE`. If the application uses multiple threads but only a 'main' thread makes MPI calls, then we should use the threading mode `FUNNELLED`. `SERIALIZED` allows for multiple threads to make MPI calls but only a single thread can make an MPI call at any given moment. That is, no two threads make MPI calls concurrently. With `MULTIPLE`, multiple threads can simultaneously enter the MPI library with no restrictions.

MPI point-to-point communication often faces challenges in multi-threaded environments. The commonly used fork-join model in conjunction with the `FUNNELLED` and `SERIALIZED` MPI threading modes results in the unnecessary synchronization of threads within the application. Using the `MULTIPLE` threading mode requires using many locks internally for critical regions within the MPI library causing lock contention. Lock contention arises from many sources such as message matching [13] and accessing network hardware [31], among others. Using multi-threaded Remote Memory Access (RMA) operations alleviates some of the issues that are observed with multi-threaded point-to-point communication such as MPI's message queues. However, RMA is notoriously difficult to program with. Although most MPI libraries provide an RMA implementation, it is not always performant.

A recent survey on MPI usage within the US Exascale computing project showed that 86% of application and system software developers plan to use MPI with multiple threads [8]. More importantly, 82% of users plan to make MPI calls within multi-threaded regions of their code. Although MPI developers have worked towards implementations that minimize the burden of multi-threaded point-to-point communication [31], many problems still persist. Therefore, it is critical that MPI improves its performance and programmability for the next generation of applications.

To address some of these issues with MPI, Partitioned Point-to-Point was proposed [17, 18] to provide semantics similar to point-to-point communication that can leverage

software-level optimizations and hardware-level triggered operations to enhance the performance of multi-threaded applications. In June of 2021, MPI Partitioned point-to-point communication was included in the MPI 4.0 standard [3]. This new communication model partitions the send and receive buffers of a point-to-point communication. This allows individual actors (e.g., threads) to mark separate partitions of data ready for transfer. This avoids using a thread barrier before a communication transfer and data can now be sent as soon as individual portions of it become available, rather than waiting for the entire buffer to be complete. This will be explained further in Section 2.1. For brevity, MPI Partitioned Point-to-Point communication may be referred to as MPI Partitioned throughout this paper.

This paper aims to understand the behaviour and performance of MPI Partitioned. Traditionally, micro-benchmarks have been used to evaluate the performance of MPI implementations on HPC systems. To the best of our knowledge, existing micro-benchmark suites [2, 4, 10, 28] do not study the new partitioned communication model of MPI.

Our micro-benchmark suite was created to provide insight into the MPI Partitioned interface in an isolated environment and provide a tool for developers to evaluate their designs. We explore the relationships between partition counts (thread counts) and message sizes to understand which types of workloads could benefit from using MPI Partitioned. We also take note of how the noise a workload generates impacts the improvement we expect to see from MPI Partitioned. Therefore we can infer which types of multi-threaded MPI applications could be good targets for porting to MPI Partitioned. We also demonstrate to application developers how they can use our micro-benchmark suite to assess the suitability of their application for Partitioned communication in their current state. Using our micro-benchmarks we create a projection of the performance benefits that are possible with Partitioned communication should they choose to make larger scale changes to their code. In this paper we make the following contributions:

- We present the first MPI micro-benchmark suite designed to evaluate MPI Partitioned implementations across a variety of metrics.
- We provide analysis of common MPI communication patterns and evaluate how applications using MPI Partitioned communication could potentially benefit from this new programming model.
- We evaluate MPI Partitioned using different types of system noise to create thread imbalances and highlight its noise tolerance.
- We provide application developers guidance on appropriate partition counts based on the message sizes, computation amount, system noise, and communication pattern.
- A new micro-benchmark suite that can be used in testing and development of MPI implementation native solutions.

The rest of this paper is organized as follows: In Section 2, we present background information on MPI Partitioned, MPI communication patterns, and system noise. We discuss the design of our micro-benchmark suite in Section 3. The performance results and analysis are presented in Section 4. In Section 5, we discuss our work in relation to existing research. Finally, we conclude our work and suggest future directions in Section 6.

2 BACKGROUND

In this section, we first introduce the MPI partitioned point-to-point communication model in greater depth using an example. We highlight which parts of the MPI interface are relevant for programming using this model. Then we explain two common communication patterns that are often used in MPI programs. These communication patterns will be used in our benchmark for evaluating MPI Partitioned. Finally, we briefly discuss why accounting for system noise is important when working with HPC systems and why we include it in our benchmarks.

2.1 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 [3]. A high level diagram is shown in Figure 1 to help explain this programming model. With partitioned communication, the send and receive buffers are segmented and actors (e.g., threads, in the context of this paper) mark data ready for transfer. Actors

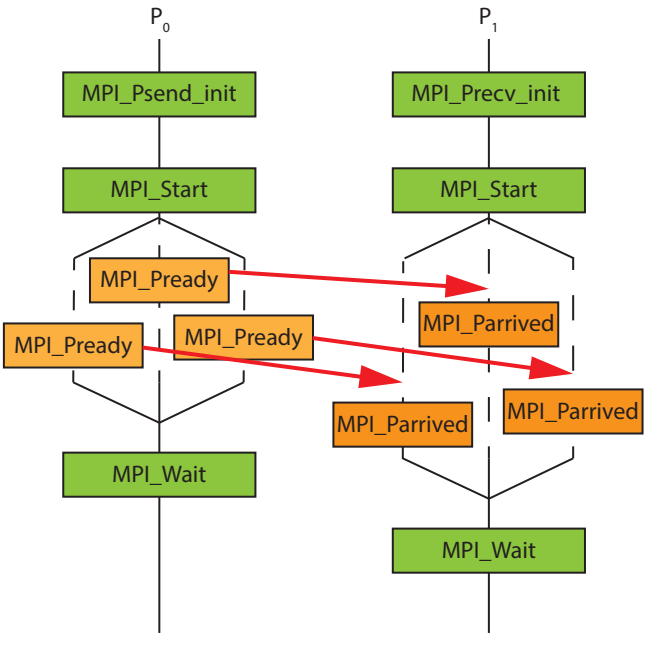


Figure 1: Overview of the MPI Partitioned Point-to-Point Communication Model

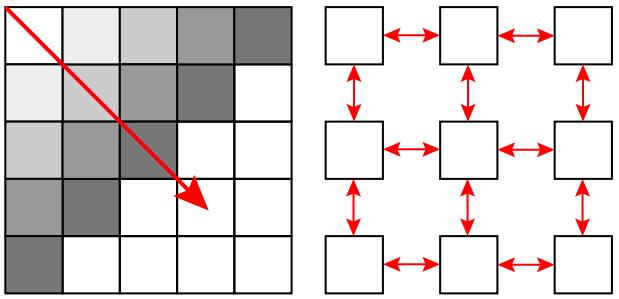
can take the form of OpenMP threads, POSIX threads, GPU work queues, and so on.

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 as if they matched during the init stage. Unlike MPI point-to-point communication, MPI Partitioned does not support wildcards as communication is initialized preemptively. This is beneficial for highly-threaded codes as it avoids matching list overheads when wildcards are allowed (preventing some high performance matching software designs). As matching is performed as if it happens at initialization time, tags must be decided on before communication (persistent) epochs start. 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.

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 thread. With MPI Partitioned, one or more partition can be assigned to each thread at run-time. Within the context of this paper we are assigning one thread to one partition. Once the sender exits a parallel region it must call `MPI_Test` or `MPI_Wait` to complete a partitioned communication transfer. In Figure 1, it is shown that the data is directly transferred between threads as we call `MPI_Pready`. The frequency and mechanism of transfer is ultimately dictated by the MPI implementation but this how an MPI user can reason their program.

On the receive side, the process can use `MPI_Test` to check if all partitions have arrived, or it can use `MPI_Parrived` which gives 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 [6].



(a) Communication Sweep **(b) 5-Point Halo Exchange**

Figure 2: Example MPI Communication Patterns

2.2 Communication Sweep

Sweeping algorithms are commonly used in HPC workload where data is decomposed and communication is sweeped across the processes [27]. A 2D sweep communication pattern is shown in Figure 2a. The computation starts on a corner and then sweeps out the data domain. Sweeps also exist in 3D problems where they are decomposed into 2D. In this paper we will focus on 3D sweeping patterns but a 2D example is shown for simplicity. These types of communication patterns stress the network as many messages are produced during their execution.

2.3 Halo Exchange

Halo exchanges are incredibly important in HPC applications. They have been previously studied using micro-benchmarks to help aid the design of system software with a focus on application domains [26]. In a typical workload, the problem is decomposed across processes and/or threads where each worker computes a part of the problem then communicates with its neighbours. In Figure 2b, we can see a visual representation of this communication pattern. A 5-point halo exchange pattern is shown, where each process communicates with four other processes. It is clear that the communication pattern differs for the edges and corners. In three dimensions this would be extended to seven points. Multi-threaded halo exchanges decomposes the problem per process and subdivides the work given to a process among threads.

2.4 System Noise

System noise on HPC clusters has been a long standing problem, especially for bulk synchronous parallel (BSP) programs which require synchronization at each step [24]. This imbalance between threads or processes results in a loss of performance during an application's execution due to waiting for barriers or data dependencies. In application or system software design, it is important to consider these sources of imbalance to develop more performant programs.

3 MICRO-BENCHMARK SUITE DESIGN

Our micro-benchmark suite contains two main parts; MPI Partitioned as point-to-point communication and its use in

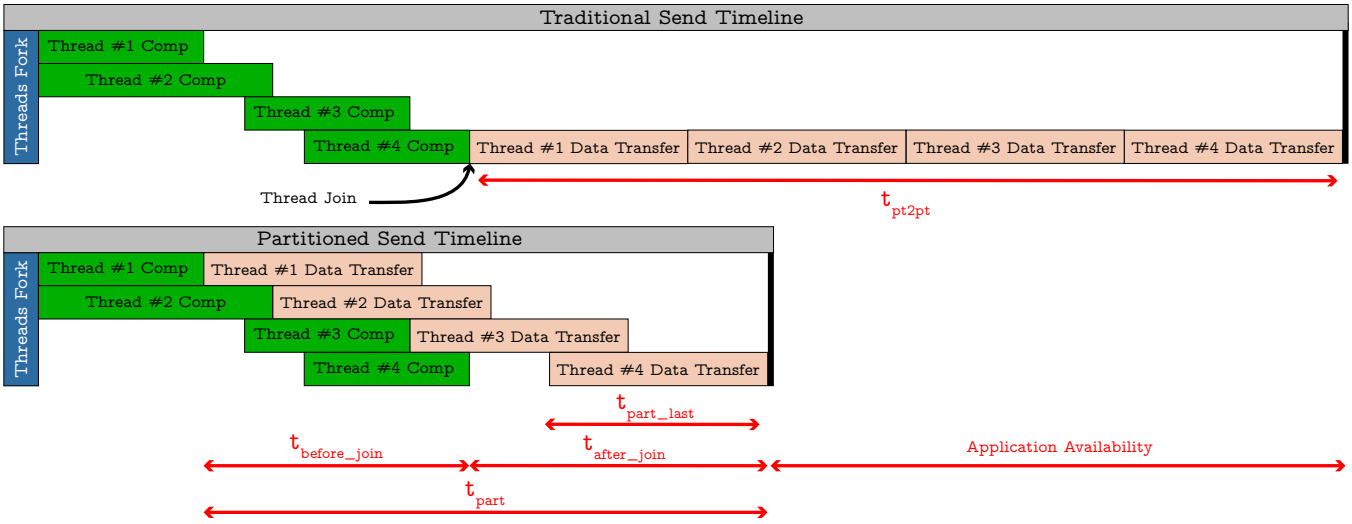


Figure 3: Assisting Visualization For Defined Metrics, adapted from [14]

.

communication patterns that mimic real world applications. The benchmark suite allow a user vary message size, partition count, system noise type and amount, and using hot cache vs cold-cache. These variables allow a user to understand the potential benefit of using MPI Partitioned in many different scenarios. The code used in this paper is available on GitHub[1].

3.1 Point-to-Point Metrics

In this paper, we have defined the following metrics for evaluating MPI Partitioned communication between two processes:

- Overhead
- Perceived Bandwidth
- Application Availability
- Early-Bird Communication

These metrics were chosen as traditional latency/bandwidth metrics do not provide the required information for us to study the MPI Partitioned interface in an isolated environment. Latency/bandwidth metrics give the user information about the performance of the underlying hardware, but the goal of MPI Partitioned are software level optimizations which will better utilize hardware.

3.1.1 Overhead. A point-to-point and a partitioned point-to-point communication could send the same data volume between two processes but it is likely that partition communication will do it with many data transfers. Sending n partitions of size k will have some overhead when compared to a single send model sending a message of size $m = n * k$. Therefore, it is important to understand what overhead partitioned communication has. We define the overhead $Overhead$ as:

$$Overhead = \frac{t_{part}}{t_{pt2pt}} \tag{1}$$

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

The parameter t_{pt2pt} is the time taken to send a single MPI send/receive operation of size m and this can be seen in Figure 3. The parameter t_{part} is the total time taken to send each individual partition measuring from the first `MPI_Pready` to the last `MPI_Parrived`.

In the case where the number of partitions is equal to 1, the communication is equivalent to a traditional persistent point-to-point communication (one thread communicating with one thread). Therefore, we would expect $t_{pt2pt} \approx t_{part}$ and have an overhead value close to one. Overhead is essentially a slowdown metric because we quantify to see how much extra load is placed on the network with multiple messages compared to a single message of the same size. It also shows how many MPI send/receive operations we could send in place of using multiple partitions.

3.1.2 Perceived Bandwidth. The concept of Perceived Bandwidth in MPI Partitioned was first proposed in [18]. We have included it in this benchmark suite for its importance and for completeness with prior work.

$$Perceived\ Bandwidth = \frac{m}{t_{part\_last}} \tag{2}$$

We define t_{part_last} as the time taken for the last partition to be sent to the receiving process. In Figure 3 this would be the data transfer referred to as *Thread #4 Data Transfer* in the Partitioned Send Timeline. This gives the bandwidth that would be required by a single send model to send the data after all tasks have completed. The perceived bandwidth will be significantly higher than what we see with the actual bandwidth from the network hardware. This is due to MPI Partitions ability to send data early before a joining of threads.

3.1.3 Application Availability. When switching from the MPI Point-to-Point to MPI Partitioned, we expect that an application using MPI Partitioned will perform better in noisy environments. We modify the metric for application availability presented in [11] to observe what MPI Partitioned can provide. We define the value of t_{after_join} to be the time that an MPI Partitioned communication still communicates after an equivalent thread join with the single-send communication. Essentially, we are measuring how much CPU time is freed to do additional work. This metric is explicitly labelled in Figure 3.

$$Application\ Availability = 1 - \frac{t_{after\_join}}{t_{pt2pt}} \tag{3}$$

3.1.4 Early-Bird Communication. With this metric we want to quantify how much of MPI Partitioned communication occurs before an equivalent thread join using an MPI send. In our benchmark, we first measure the amount taken for threads to be forked, the tasks completed, and then the threads to join. Then we calculate the amount of partitioned communication that has occurred before the join and label it t_{before_join}. In Figure 3, t_{before_join} would be the first two data transfers and a portion of the third transfer in the partitioned send timeline. The parameter t_{part} was defined in Section 3.1.1. Then from these values we obtain the percentage of communication which occurs before the thread join.

$$\%\ early\ bird = \frac{t_{before\_join}}{t_{part}} \tag{4}$$

With this metric it will be unlikely that a value of exactly 100% early-bird communication will be obtained but it is possible to asymptotically approach this value. A value close to 0% implies that an MPI Partitioned implementation does not have any features that provide early-bird communication.

3.2 MPI Communication Patterns

We chose two common MPI communication patterns to study in this paper; a communication sweep and a halo exchange. Further details on these communication patterns can be found in Section 2.3 and Section 2.2, respectively. For these communication patterns we have used the Ember module [19] from the structural simulation toolkit (SST) [23]. The *motifs* in this module are computation/communication patterns that are used to simulate MPI workloads. We use the Sweep3D motif and Halo3D, which is a 7-Point Halo Exchange. These motifs were modified to use OpenMP and MPI Partitioned communication for our micro-benchmark suite.

3.3 Noise Models

As stated in Section 2.4, HPC applications often experience a loss in performance from system noise. Therefore, to simulate a realistic workload we wanted the ability to to evaluate our benchmark using noise. Our benchmarks can evaluated with the following noise models:

- Single Thread Noise
- Uniform Noise
- Gaussian Noise

The *Single Thread Noise* model is when we delay a single thread by some delay amount. All other threads compute the expected compute amount. This is to mimic a context switch on a single CPU core [21]. This is the model that was used in [18] to evaluate Finepoints. With our *Uniform Noise* model, each thread samples a compute amount from a uniform probability distribution on the interval $[comp, comp + \%\ noise]$. For the *Gaussian Noise* model, we sample a compute amount from a normal probability distribution with the user defined computation amount as our mean and the % noise as our standard deviation. This is similar to the method used in [22]. For this noise model, we are ignoring edge cases where our delay amount is sampled from the tails of the normal distribution as we suspect that it will be sufficiently infrequent. For both the Uniform and Gaussian models, noise will be applied to all threads.

3.4 Hot vs Cold Cache Evaluation

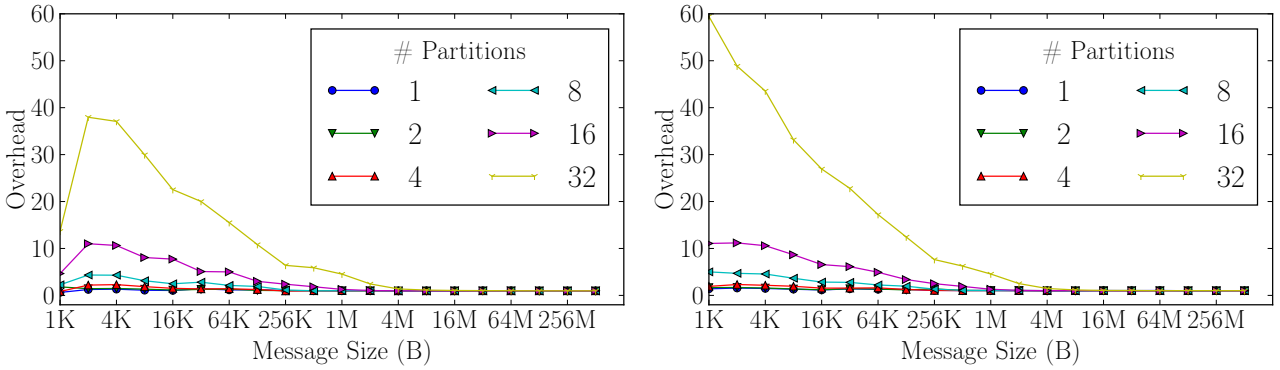
Most micro-benchmarks use a hot cache where the same memory location is repeatedly accessed. Thus, resulting in the data to be in CPU cache when needed. To better imitate real world usage, we also consider invalidating the CPU's cache and refer to this as our cold cache. For cache invalidation, we read/write from an 8MB buffer to clear the CPU's L1 and L2 cache using a similar method to the SMBs [10]. This results in CPU accessing memory not in its cache for each iteration of our benchmark.

4 PERFORMANCE RESULTS AND ANALYSIS

4.1 Experimental Setup

Experiments were conducted on the Niagara compute cluster at the SciNet HPC Consortium [25]. 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. In this paper we will limit our point-to-point tests to a single wing so there will only be a single switch between any two processes. As our system has one NUMA domain per socket, we have not considered NUMA effects aside from when threads are mapped to cores between sockets. That said, multiple NUMA domains shouldn't impact the results due to the design of this micro-benchmark suite.

Niagara uses the GNU/Linux distribution CentOS 7.6. To conduct our tests we only evaluated a single MPI Partitioned implementation, MPIPCL, due to the current experimental state of the MPI library native implementations such as Open MPI or MPICH. As MPIPCL is dependent on an MPI implementation's send and receive operations, we used the MPIPCL library on top of Open MPI from the master branch (7b177ce) and UCX v1.11.0. MPIPCL lets us have a stable layered library approach to partitioned communication that is helpful in these very early stages of MPI partitioned



(a) Cold Cache **(b) Hot Cache**

Figure 4: Overhead of Partitioned Point-to-Point Communication Relative to Point-to-Point Communication for 10ms of Compute

development. Previous work showing MPI partitioned communication in native libraries is preliminary [14, 29] and are still undergoing development. This would make them unsuitable for in-depth consistent testing for these micro-benchmarks. For our investigation into SNAP, we used a C port of SNAP from LANL [5]. This application was profiled using v3.5 of mpiP [1].

In our experiments, when we increase partition counts, we apply the same compute amount across all threads, thus using strong scaling unless otherwise stated. As we have injected noise using the noise models explained in Section 3.3, our results will appear somewhat noisy when presented. The results shown are averages over several trials, and we have pruned extreme noise samples from the dataset to avoid extreme outliers that do not often occur in practice. We also compare the effects of hot and cold caches when relevant.

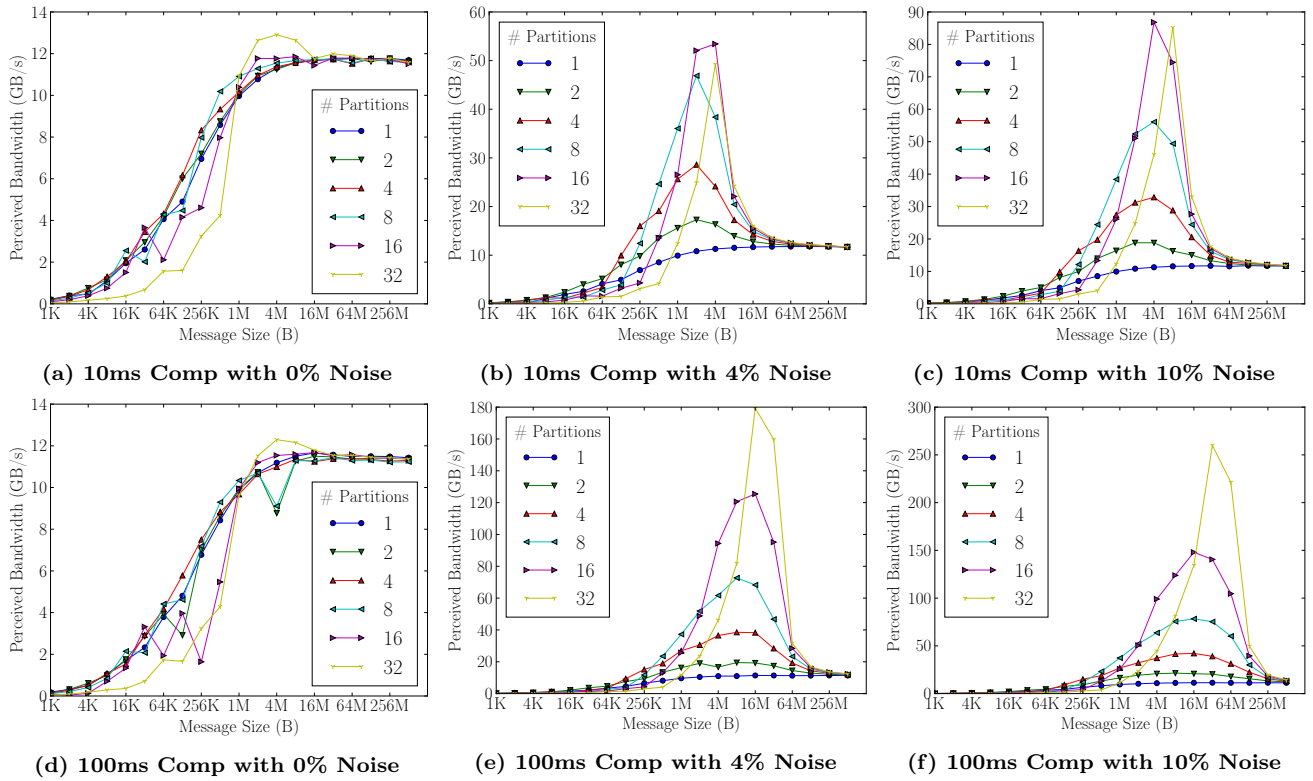
4.2 Overhead

Determining the overhead of MPI Partitioned (see Section 3.1.1) allows us to observe the loss in performance that could occur by switching to this programming model. We measure the overhead without the simulated noise to view it from an ideal environment in Figure 4. The cold cache (using cache invalidation) results shows a lower overhead than with the hot cache (without using cache invalidation). At first this seems surprising but the overhead metric is a ratio comparing a single MPI Point-to-Point to MPI Partitioned. Using a cold cache results in data not being present in memory when it is needed. Therefore, the CPU needs to read the required data from memory. So the cost of reading from memory is amortized during the benchmark. With one thread, our overhead value ranges between 1.6x and 1x compared to a single send, for both the hot and cold cache tests. For large messages, there is little cost (≈1x) associated with using MPI Partitioned compared to MPI point-to-point. This is ideal as we want to observe no loss in using MPI Partitioned. For two partitions, the overhead is also fairly low. Thus, one and two partitions are not clearly visible in Figure 4.

As thread count increases so does overhead, especially for small messages. These overheads only further increase with partition count. As small messages are latency bound, subdividing a message is more expensive since headers and other information is added as we transmit data across a wire. Although this is true for 1-16 partitions, there is a significant increase in overhead of up to 59.4x when using 32 partitions. In our design methodology we assigned each partition to a single thread. As our system has 20 cores per socket, using 32 partitions causes some of those threads to spill over to the second socket resulting in an increase in communication cost. Therefore, application designers should consider the platform to ensure that the partition counts are chosen to ensure that they are associated with a single socket. On systems with multiple NUMA domains per socket, this would be an important benchmark to refer to if any unexpected behaviour occurs when increasing partitions. For larger messages, the overhead of splitting the buffer into multiple partitions has a lower cost. It can be suggested that MPI Partitioned better suits application using medium to large messages.

4.3 Perceived Bandwidth

In Figure 5, we present the results for our perceived bandwidth (see Section 3.1.2) benchmark with different compute and noise amounts. When using one or two partitions a traditional bandwidth curve is shown, this can also be seen in the tests using 0% noise. For smaller messages, our perceived bandwidth is relatively low as these messages are latency bound. Our overhead results in Figure 4 showed that around the 1MB mark the overhead dropped significantly. At this message size our perceived bandwidth begins to reach its peak. As the message size increases the perceived bandwidth increases to a point, then there is a sharp decline. We see this initial sharp increase as the perceived bandwidth metric is a function of the time spent in the last transfer (send side partition) to arrive at the MPI_Pready call. Therefore, once a single partition saturates the network we see a decline in performance. As the partition count is increased we see the



(a) 10ms Comp with 0% Noise **(b) 10ms Comp with 4% Noise** **(c) 10ms Comp with 10% Noise**

(d) 100ms Comp with 0% Noise **(e) 100ms Comp with 4% Noise** **(f) 100ms Comp with 10% Noise**

Figure 5: Perceived Bandwidth of MPI Partitioned Point-to-Point Communication with Uniform Noise and a Hot Cache for Different Noise and Compute Amounts

peak increase; this is due to the total message size being further subdivided. When increasing partition count from 16 to 32 with 10ms computation a decline in performance is shown, the same is not true when compared to 100ms in noisy environments. Again, this is likely an issue of using more threads than available cores. With 100ms comp this overhead can be hidden as the computation time is sufficiently large.

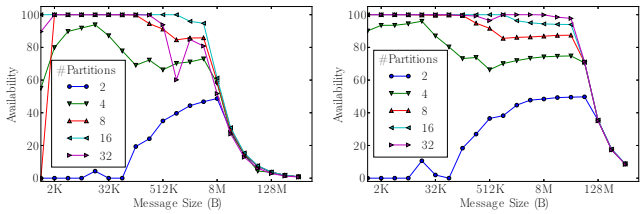
4.4 Application Availability

Our experiments on application availability (see Section 3.1.3) are shown in Figure 6 and Figure 7. The impact of varying partitions size in Figure 6 shows how a user can choose a partition size for their application. Generally, in a noisy environment more partitions result in more time for the CPU to compute more for small messages. The issue of thread spillover is also present in Figure 6a, as 16 partitions perform better than 32. After around 4MB application availability drops off; this seems to correlate with the peak we see in the perceived bandwidth. Increasing computation also results in a shift as to where the availability starts to drop off.

The results in Figure 6 are all presented using the single thread delay model but we would also like to observe availability with different noise distributions. The impact of thread arrival is presented in Figure 7. The best availability is shown with the single delay model as all other threads can continue their execution without being blocked and only the delayed thread will face consequences. With both the uniform and normal distributions, we see our application availability is worse for small messages than the single delay model as the imbalance between threads is much smaller and early-bird communication cannot be taken advantage of.

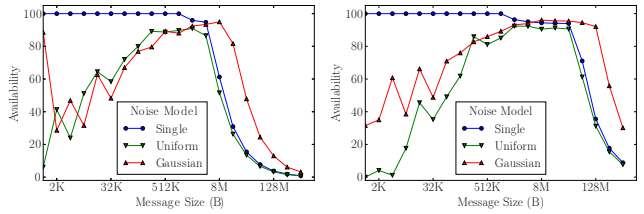
4.5 Early-Bird Communication

In Figure 8, we present the results for our investigation in quantifying early-bird communication using the Uniform noise model. Note, that results using this benchmark with 0% percent noise or with one partition would not be useful as those results would be the same as a single-send model. For both 10ms and 100ms compute, early-bird communication is taken advantage of as most messages are transferred before the single-send model would join its threads. This is especially true for small and medium sized messages as transfer time is significantly less than the computation time. Early-bird communication is better utilized as we increase the partition count with 100ms compute. That said, there is minimal difference between 8 and 32 partitions. The same cannot be said for the 10ms compute scenario because after 16 partitions utilization declines. It is likely 10ms compute is not sufficient to observe the desired properties of MPI Partitioned for messages larger than 2MB as the early-bird communication



(a) 10ms Comp (b) 100ms Comp

Figure 6: Application Availability When Using MPI Partitioned Point-to-Point Communication With a Hot Cache and Our Single Thread Delay Model With 4% Noise



(a) 10ms Comp (b) 100ms Comp

Figure 7: The Impact of Noise Models on Application Availability. 16 Partitions are Used with 4% Noise and a Hot Cache

window of opportunity is too small. However, it could still be useful for an application with small compute to use MPI Partitioned if they transfer smaller data.

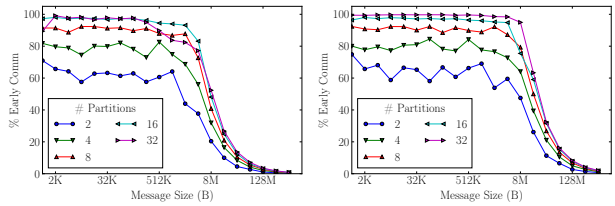
Another observation that can be seen is that with two partitions we effectively utilize early-bird communication. When comparing to our availability results for small messages it seems that the noise is sufficiently small that sending messages does not improve application availability as much.

4.6 Sweep3D

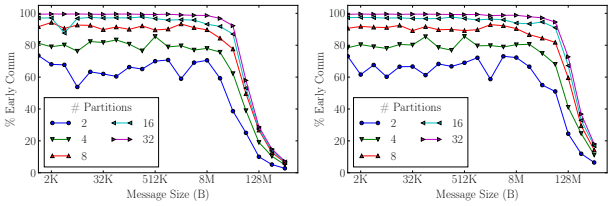
Using our communication pattern micro-benchmarks we can also gain insight into how MPI Partitioned communication performs with many processes, as previous sections in this paper have only covered point-to-point performance between two processes. In these tests, we have compared the communications using MPI point-to-point with a single thread, multiple threads, and with multiple threads using MPI Partitioned.

Figure 9 and Figure 10 show the results for the Sweep3D communication pattern's throughput for 10ms and 100ms of computation, respectively. For the most part, we observe that the performance between MPI Partitioned and MPI point-to-point is relatively similar for small and medium messages. We suspect that this is largely due to MPIPCL being implemented with MPI point-to-point. Therefore, these comparisons are limited as we are not observing what a well optimized MPI Partitioned implementation could provide for applications.

As we are using weak scaling for the data size, when increasing the partition count from four to 16 there is a 4x



(a) 10ms Comp with 4% Uniform Noise (b) 10ms Comp with 10% Uniform Noise



(c) 100ms Comp with 4% Uniform Noise (d) 100ms Comp with 10% Uniform Noise

Figure 8: Percentage of Early-Bird Communication with MPI Partitioned Point-to-Point Communication

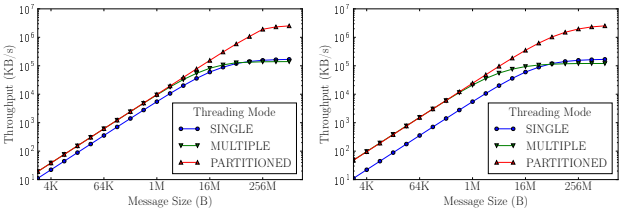
increase in compute per process (strong scaling is used for computation). We can see that the increase in thread count results in improved throughput for the Sweep3D communication pattern for both `MULTIPLE` and `PARTITIONED`. As thread count increases, the data size per thread decreases so less data is sent per thread across the network.

For larger messages, it can be seen that the difference in throughput between MPI point-to-point with multiple threads and MPI Partitioned grows. With 10ms computation, the multi-threaded throughput falls below what we measure with the single-threaded implementation and MPI Partitioned implementation has a 15.1x higher throughput than single-threaded. The trends for 10ms and 100ms computation results are relatively similar. As expected, due to the larger compute the throughput drops with 100ms. Also the divergence between MPI point-to-point and MPI Partitioned occurs at a larger message size changes.

4.7 Halo Exchange

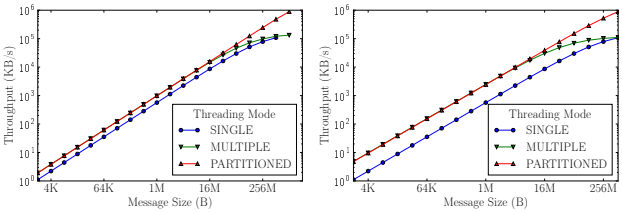
Figure 11 and Figure 12 show the results for a 3D halo exchange with 10ms and 100ms computation, respectively. Due to the nature of how a 3D halo exchange can be implemented we are required to use thread counts that are cubed numbers. We have chosen thread counts of eight and 64; only the eight-thread configuration fits onto a single socket, whereas 64 threads require over-subscription. Eight threads results in four partitions as each face has 2x2 threads. For our 64-thread experiment, each face of the cube has 16 partitions (4x4).

The results using four partitions perform relatively the same for all threading modes which we studied. Therefore, in the figures it is difficult to distinguish the different lines. Due to weak scaling we have 8x more compute while providing a similar throughput to a single-threaded implementation.



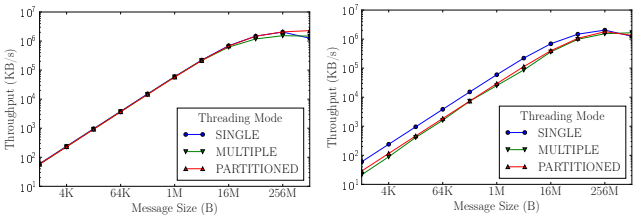
(a) 4 Partitions **(b) 16 Partitions**

Figure 9: Sweep3D Communication Throughput For 10ms, 4% Single Noise with a Hot Cache



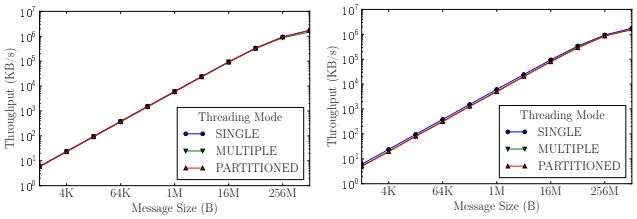
(a) 4 Partitions **(b) 16 Partitions**

Figure 10: Sweep3D Communication Throughput For 100ms, 4% Single Noise with a Hot Cache



(a) 4 Partitions - 8 Threads **(b) 8 Partitions - 64 Threads (Oversubscribed)**

Figure 11: Halo3D Communication Throughput For 10ms, 4% Single Noise with a Hot Cache



(a) 4 Partitions - 8 Threads **(b) 8 Partitions - 64 Threads (Oversubscribed)**

Figure 12: Halo3D Communication Throughput For 100ms, 4% Single Noise with a Hot Cache

A larger difference between threading modes is shown for 64 threads. The performance of multi-threaded MPI point-to-point is close to what we see with MPI Partitioned. Again, we suspect that is due to using MPIPCL. For a 16MB message size with 10ms computation, we see 42.6% decrease in throughput with a 64x increase in total computation. With 100ms computation, we see only a 16.8% decrease for the same increase in computation. Oversubscription could be useful in applications with very large compute where the workload can be dynamically distributed among threads.

4.8 Proxy Application Projection

In Section 4.6, we saw promising results for using MPI Partitioned with sweeping communication patterns. To further investigate how porting an application to MPI Partitioned could yield performance improvements, we profile SNAP application [5] using the mpiP profiler [1]. SNAP is a proxy application that models the performance of discrete ordinates neutral particle transport applications. The application is modelled based on PARTISN, which solves the linear Boltzmann transport equation. SNAP was chosen as it uses a 3 dimensional sweeping pattern.

The projected performance by porting SNAP to MPI Partitioned can be seen in Figure 13. In this figure, we projected the performance based on the 15.1x performance improvement we saw in Section 4.6. For smaller process counts, the expected performance improvement is relatively small; this is due to MPI send/receives contributing to only 1-6% of application run-time. At 128 and 256 nodes, MPI send/receives contributes to 20.4% and 54.5% of application run-time, respectively. This results in the large speedups that we expect to see for these process counts.

5 RELATED WORK

MPI Endpoints was initially proposed at the MPI Forum suggesting that assigning ranks to threads could be one possible solution to handling hybrid MPI codes [9]. 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. Challenges with multi-threaded MPI performance usually occurs with threads requiring access to mutexes which can cause starvation [6], contention with the message queue and message progression, [13], or issues handling the MPI_Request object [16].

Another approach to hybrid programming with MPI is to use the MPI Partitioned Point-to-Point communication model. MPI Partitioned tries to form a hybrid between RMA and point-to-point APIs of MPI. It provides similar semantics to point-to-point in the form of send/receives but it intends to deliver the performance of multi-threaded RMA. It decouples the control of communication with the data transfer itself [17, 18]. The partitioned API accommodates multi-threaded and other hybrid codes by the concept of partitions. Each partition of data is committed by individual actors; i.e., threads, and their arrival can also be checked by the receiver [12, 18]. It also avoids many of the MPI message matching issues, as send/receives are matched once in a serial portion of

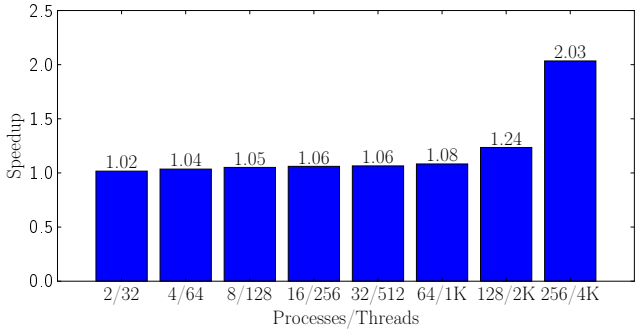


Figure 13: Expected Speedup From Porting SNAP-C to MPI Partitioned.

the code, then the buffers can be reused during an application execution.

A few MPI Partitioned implementations exist but the remaining optimization space is still relatively large [7, 14, 29, 30]. In these works, the authors have thus far investigated portability [7], using helper threads for communication [29], and using an RMA based implementation [14]. Although these works have studied MPI Partitioned, they have not used a public benchmarking suite.

System noise often causes unnecessary delays and performance degradation for HPC applications [24]. In [15] the authors showed how to quantify system noise and the impact on applications. Noise has been found to follow a Gaussian distribution with a mean value of 100ms with a standard deviation of 3ms [22]. Noise has also been used to predict the performance degradation of HPC applications in noisy environments [22].

Currently, there are no publicly available micro-benchmarks for partitioned communication. This includes some of the most common MPI benchmarks such as SMB [10], the Ohio State Micro-benchmark suite (OMB) [4], and Intel MPI Benchmarks [2]. Some benchmarks do exist to evaluate multi-threaded MPI communication [26], but they do not consider system noise or any other metrics that we have studied [28]. Micro-benchmarking MPI communication patterns for system design has also been previously studied but again not within the context of partitioned communication.

6 CONCLUSION

In this paper, we addressed the lack of open-source micro-benchmarks for MPI Partitioned communication with a proposal for methods in which we can evaluate new designs. Our micro-benchmark design provides the user with ability to adjust many constraints within their measurements, such as hot vs cold caching, different probability distributions for thread noise and arrival imbalance, compute amounts, noise amounts, and partition counts. These parameters allow users to search the parameter space for optimal partition communication usage for their application using a set of predefined metrics. For applications that use common communication patterns such as halo exchanges or sweeping patterns we have provided analysis on how partition communication can be used. At this stage, MPI Partitioned implementations are new and not highly optimized, so the performance difference is small relative to using multi-threaded MPI Point-to-Point.

Our results show the benefits of using MPI Partitioned communication in noisy environments. We found that the amount of application compute time also greatly impacts what performance we could gain. We also have shown that it is important to consider partition count based on the message size that a user intends to use and the hardware platform to ensure that threads are mapped appropriately.

6.1 Future Work

This paper provides a baseline for future work conducted on partitioned communication and its extensions and run-time improvements to MPI implementations can be compared. As we only evaluated MPIPCL, once other MPI implementations are sufficiently mature, it would be useful to compare them amongst themselves to see the trade-offs for different types of partitioned workloads. Another limitation of MPIPCL is that send and receive partitions must have equal counts. Therefore, we have not explored different partition sizes between processes. Porting applications to use MPI Partition based of the projections we have presented in the paper could be useful to emphasize real world usage and application of this new MPI API.

The proposed micro-benchmark suite only handles the scenario where buffers are located in host memory. With the growth of accelerators and MPI Partitioned proposals to handle invocation of `MPI_Pready` from compute kernels or task queues (e.g., `sycl::queue` or `cudaStreams_t`), it is important to consider how this may impact our application design. Finally, the extension to MPI Partitioned collectives [20] could be necessary as it will face some similar motives as to the work that has been conducted in this paper.

ACKNOWLEDGMENTS

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. We thank the developers of MPIPCL for their preliminary MPI partitioned implementation that allowed us to conduct this research.

REFERENCES

[1] 2022. A light-weight MPI profiler (mpiP). https://github.com/LLNL/mpiP
[2] 2022. Intel® MPI Benchmarks. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-mpi-benchmarks.html
[3] 2022. Message Passing Interface. http://www.mpi-forum.org
[4] 2022. OSU Micro-Benchmarks. https://mvapich.cse.ohio-state.edu/benchmarks/
[5] 2022. SNAP: SN (Discrete Ordinates) Application Proxy. https://github.com/lanl/SNAP/tree/main/ports/snap-c
[6] Abdelhalim Amer, Huiwei Lu, Yanjie Wei, Pavan Balaji, and Satoshi Matsuoka. 2015. MPI+Threads: Runtime Contention and Remedies. In *Proceedings of the 20th ACM SIGPLAN*

Symposium on Principles and Practice of Parallel Programming (San Francisco, CA, USA) *(PPoPP 2015)*. Association for Computing Machinery, New York, NY, USA, 239–248. https://doi.org/10.1145/2688500.2688522

[7] Purushotham V. Bangalore, Andrew Worley, Derek Schafer, Ryan E. Grant, Anthony Skjellum, and Sheikh Ghafoor. 2020. A Portable Implementation of Partitioned Point-to-Point Communication Primitives. In *Poster: Proceedings of the 27th European MPI Users' Group Meeting* (Austin, TX, USA) *(EuroMPI/USA '20)*. Association for Computing Machinery, New York, NY, USA, 1–3.

[8] David E. Bernholdt, Swen Boehm, George Bosilca, Manjunath Gorentla Venkata, Ryan E. Grant, Thomas Naughton, Howard P. Pritchard, Martin Schulz, and Geoffroy R. Vallee. 2020. A survey of MPI usage in the US exascale computing project. *Concurrency and Computation: Practice and Experience* 32, 3 (2020), 1–16. https://doi.org/10.1002/cpe.4851 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4851 e4851 cpe.4851.

[9] James Dinan, Ryan E Grant, Pavan Balaji, David Goodell, Douglas Miller, Marc Snir, and Rajeev Thakur. 2014. Enabling Communication Concurrency through Flexible MPI Endpoints. *Int. J. High Perform. Comput. Appl.* 28, 4 (Nov. 2014), 390–405. https://doi.org/10.1177/1094342014548772

[10] Doug Doefler, Brian W Barrett, RE Grant, MG Dosanjh, and T Groves. 2009. Sandia MPI microbenchmark suite (SMB). *Sandia National Laboratories* 2020 (2009).

[11] Douglas Doerfler and Ron Brightwell. 2006. Measuring MPI Send and Receive Overhead and Application Availability in High Performance Network Interfaces. In *Proceedings of the 13th European PVM/MPI User's Group Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface* (Bonn, Germany) *(EuroPVM/MPI'06)*. Springer-Verlag, Berlin, Heidelberg, 331–338. https://doi.org/10.1007/11846802_46

[12] Matthew Dosanjh and Ryan Grant. 2019. Receive-Side Partitioned Communication. (9 2019). https://doi.org/10.2172/1763213

[13] Matthew G.F. Dosanjh, Ryan E. Grant, Whit Schonbein, and Patrick G. Bridges. 2020. Tail queues: A multi-threaded matching architecture. *Concurrency and Computation: Practice and Experience* 32, 3 (2020), 1–13. https://doi.org/10.1002/cpe.5158 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.5158 e5158 cpe.5158.

[14] Matthew G.F. Dosanjh, Andrew Worley, Derek Schafer, Prema Soundararajan, Sheikh Ghafoor, Anthony Skjellum, Purushotham V. Bangalore, and Ryan E. Grant. 2021. Implementation and evaluation of MPI 4.0 partitioned communication libraries. *Parallel Comput.* 108 (2021), 102827. https://doi.org/10.1016/j.parco.2021.102827

[15] Kurt B. Ferreira, Patrick Bridges, and Ron Brightwell. 2008. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. 1–12. https://doi.org/10.1109/SC.2008.5219920

[16] David Goodell, Pavan Balaji, Darius Buntinas, Gábor Dózsa, William Gropp, Sameer Kumar, Bronis R. de Supinski, and Rajeev Thakur. 2010. Minimizing MPI Resource Contention in Multithreaded Multicore Environments. In *2010 IEEE International Conference on Cluster Computing*. 1–8. https://doi.org/10.1109/CLUSTER.2010.11

[17] Ryan Grant, Anthony Skjellum, and Purushotham V. Bangalore. 2015. Lightweight threading with MPI using Persistent Communications Semantics.. In *Workshop on Exascale MPI (ExaMPI). Held in conjunction with the 2015 International Conference-for High Performance Computing, Networking, Storage and Analysis (SC15)*. 1–3. https://www.osti.gov/biblio/1328651

[18] Ryan E. Grant, Matthew G. F. Dosanjh, Michael J. Levenhagen, Ron Brightwell, and Anthony Skjellum. 2019. Finepoints: Partitioned Multithreaded MPI Communication. In *High Performance Computing*, Michèle Weiland, Guido Juckeland, Carsten Trinitis, and Ponnuswamy Sadayappan (Eds.). Springer International Publishing, Cham, 330–350.

[19] Simon D. Hammond, K. Scott Hemmert, Michael J. Levenhagen, Arun F. Rodrigues, and G.R. Voskuilen. 2015. Ember: Reference Communication Patterns for Exascale. (12 2015).

[20] Daniel J. Holmes, Anthony Skjellum, Julien Jaeger, Ryan E. Grant, Purushotham V. Bangalore, Matthew G.F. Dosanjh, Amanda Bienz, and Derek Schafer. 2021. Partitioned Collective Communication. In *2021 Workshop on Exascale MPI (ExaMPI)*. 9–17. https://doi.org/10.1109/ExaMPI54564.2021.00007

[21] Chuanpeng Li, Chen Ding, and Kai Shen. 2007. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*. 2–es.

[22] Oscar H. Mondragon, Patrick G. Bridges, Scott Levy, Kurt B. Ferreira, and Patrick Widener. 2016. Understanding Performance Interference in Next-Generation HPC Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '16)*. IEEE Press, Article 33, 12 pages.

[23] Richard Murphy, Arun F. Rodrigues, Peter Kogge, and Keith Douglas Underwood. 2004. The structural simulation toolkit : a tool for bridging the architectural/microarchitectural evaluation gap. (12 2004). https://www.osti.gov/biblio/1088092

[24] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. 2003. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing* (Phoenix, AZ, USA) *(SC '03)*. Association for Computing Machinery, New York, NY, USA, 55. https://doi.org/10.1145/1048935.1050204

[25] Marcelo Ponce, Ramses van Zon, Scott Northrup, Daniel Gruner, Joseph Chen, Fatih Ertinaz, Alexey Fedoseev, Leslie Groer, Fei Mao, Bruno C. Mundim, Mike Nolta, Jaime Pinto, Marco Saldarriaga, Vladimir Slavnic, Erik Spence, Ching-Hsing Yu, and W. Richard Peltier. 2019. 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)* (Chicago, IL, USA) *(PEARC '19)*. Association for Computing Machinery, New York, NY, USA, Article 34, 8 pages. https://doi.org/10.1145/3332186.3332195

[26] Whit Schonbein, Scott Levy, W. Pepper Marts, Matthew G. F. Dosanjh, and Ryan E. Grant. 2020. Low-cost MPI Multithreaded Message Matching Benchmarking. In *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 170–179. https://doi.org/10.1109/HPCC-SmartCity-DSS50907.2020.00022

[27] Robert Searles, Sunita Chandrasekaran, Wayne Joubert, and Oscar Hernandez. 2018. Abstractions and Directives for Adapting Wavefront Algorithms to Future Architectures. In *Proceedings of the Platform for Advanced Scientific Computing Conference* (Basel, Switzerland) *(PASC '18)*. Association for Computing Machinery, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3218176.3218228

[28] Rajeev Thakur and William Gropp. 2009. Test suite for evaluating performance of multithreaded MPI communication. *Parallel Comput.* 35, 12 (2009), 608–617. https://doi.org/10.1016/j.parco.2008.12.013 Selected papers from the 14th European PVM/MPI Users Group Meeting.

[29] Andrew Worley, Prema Prema Soundararajan, Derek Schafer, Purushotham Bangalore, Ryan Grant, Matthew Dosanjh, Anthony Skjellum, and Sheikh Ghafoor. 2021. Design of a Portable Implementation of Partitioned Point-to-Point Communication Primitives. In *50th International Conference on Parallel Processing Workshop* (Lemont, IL, USA) *(ICPP Workshops '21)*. Association for Computing Machinery, New York, NY, USA, Article 35, 11 pages. https://doi.org/10.1145/3458744.3474046

[30] Andrew Preston Worley. 2021. *A Portable Implementation of Partitioned Point-To-Point Communication Primitives*. Master's thesis. Tennessee Technological University.

[31] Rohit Zambre, Aparna Chandramowliswharan, and Pavan Balaji. 2020. *How I Learned to Stop Worrying about User-Visible Endpoints and Love MPI*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3392717.3392773

A PAPER ARTIFACT DESCRIPTION APPENDIX

A.1 Software Artifact Availability:

All code used in this paper will be publicly available. These micro-benchmarks were designed to measure MPI Partitioned's capability. The linked GitHub repository should explain how to setup and run these tests.

A.2 Hardware Artifact Availability:

Experiments were conducted on the Niagara compute cluster at the SciNet HPC Consortium.

A.3 Data Artifact Availability:

N/A

A.4 Proprietary Artifacts:

N/A

A.5 Artifact 1:

A.5.1 Persistent ID:. https://github.com/Yiltan/MPI-Partitioned-Microbenchmarks.

A.5.2 Artifact name: MPI-Partitioned-Microbenchmarks

A.5.3 Citation of artifact: This paper.

A.5.4 Relevant hardware details: 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.

A.5.5 Operating systems and versions: GNU/Linux distribution CentOS 7.6.

A.5.6 Compilers and versions: gcc (9.4.0)

A.5.7 Applications and versions:

- Open MPI from the master branch (7b177ce)
- UCX v1.11.0
- SNAP-C from https://github.com/lanl/SNAP/tree/main/ports/snap-c
- mpiP 3.5

A.5.8 Key algorithms: N/A