Efficient Process Arrival Pattern Aware Collective Communication for Deep Learning

Pedram Alizadeh
ECE Department
Queen's University
Kingston, ON, Canada
18pm7@queensu.ca

Amirhossein Sojoodi
ECE Department
Queen's University
Kingston, ON, Canada
amir.sojoodi@queensu.ca

Yıltan Hassan Temuçin
ECE Department
Queen's University
Kingston, ON, Canada
yiltan.temucin@queensu.ca

Ahmad Afsahi
ECE Department
Queen's University
Kingston, ON, Canada
ahmad.afsahi@queensu.ca

ABSTRACT

MPI collective communication operations are used extensively in parallel applications. As such, researchers have been investigating how to improve their performance and scalability to directly impact application performance. Unfortunately, most of these studies are based on the premise that all processes arrive at the collective call simultaneously. A few studies though have shown that imbalanced Process Arrival Pattern (PAP) is ubiquitous in real environments, significantly affecting the collective performance. Therefore, devising PAP-aware collective algorithms that could improve performance, while challenging, is highly desirable. This paper is along those lines but in the context of Deep Learning (DL) workloads that have become maintstream.

This paper presents a brief characterization of collective communications, in particular MPI_Allreduce, in the Horovod distributed Deep Learning framework and shows that the arrival pattern of MPI processes is indeed imbalanced. It then proposes an intra-node shared-memory PAP-aware MPI_Allreduce algorithm for small to medium messages, where the leader process is dynamically chosen based on the arrival time of the processes at each invocation of the collective call. We then propose an intra-node PAP-aware algorithm for large messages that dynamically constructs the reduction schedule at each MPI_Allreduce invocation. Finally, we propose a PAP-aware cluster-wide hierarchical algorithm, which is extended by utilizing our intra-node PAP-aware designs, that imposes less data dependency among processes given its hierarchical nature compared to flat algorithms. The proposed algorithms deliver up to 58% and 17% improvement at the micro-benchmark and Horovod with TensorFlow application over the native algorithms, respectively.

EuroMPI/USA'22, September 26–28, 2022, Chattanooga, TN, USA

ACM ISBN 978-1-4503-9799-5/22/09…$15.00
https://doi.org/10.1145/3555819.3555857

CCS CONCEPTS

• **Computing methodologies** → **Parallel programming languages**.

KEYWORDS

MPI, Distributed Deep Learning, Collective Communication, Process Arrival Pattern, MPI_Allreduce

ACM Reference Format:
Pedram Alizadeh, Amirhossein Sojoodi, Yıltan Hassan Temuçin, and Ahmad Afsahi. 2022. Efficient Process Arrival Pattern Aware Collective Communication for Deep Learning. In *EuroMPI/USA'22: 29th European MPI Users' Group Meeting (EuroMPI/USA'22), September 26–28, 2022, Chattanooga, TN, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3555819.3555857

1 INTRODUCTION

The Message Passing Interface (MPI) [2] is the de facto standard for parallel programming in HPC applications. The MPI standard provides different communication semantics such as point-to-point, partitioned point-to-point, one-sided, and collective operations. Collective communications have been extensively used in HPC applications including distributed DL applications, and their performance is critical to the performance of such applications [16].

Optimizing collective operations has been an active area of research [4, 14, 18, 19, 25, 32, 33, 36, 39]. Studies have shown that optimizing reduction collective operations, specifically MPI_Allreduce, the most exploited collective in DL applications [5], can significantly improve the performance of distributed DL frameworks [8–10, 37], such as Horovod [35], TensorFlow [3] and CNTK [34]. Unfortunately, however, similar to other collective operations, reduction algorithms have been optimized mainly under the premise that all processes start the operation at the same time. Research conducted on the arrival time of processes at the collective calls has shown that this is rarely the case, and that imbalanced *Process Arrival Pattern* are ubiquitous in HPC applications [12, 17, 23, 29]. It has been shown that the well-performing collective algorithms for the balanced micro-benchmarks, which are usually the algorithms of choice in MPI implementations, perform poorly under imbalanced process arrival patterns [12]. Therefore, it is important

to propose new algorithms capable of exploiting process imbalance to deliver high performance when there is an asynchrony among the processes arriving at the collective operation. We study the PAP behavior of the Horovod + TensorFlow application, and then propose different PAP-aware designs capable of delivering high-performance under imbalanced PAPs. This paper makes the following contributions:

- We characterize the communication pattern of Horovod + TensorFlow and provide evidence that MPI processes arrive asynchronously at the MPI_Allreduce collective calls.
- We propose an intra-node PAP-aware shared-memory aware MPI_Allreduce algorithm for small to medium messages that dynamically chooses the leader of the operation based on the arrival time of the processes at each invocation of the collective call to achieve performance. We observe up to 56% improvement over native algorithms under different process arrival patterns.
- We propose an intra-node PAP-aware MPI_Allreduce for large messages that dynamically constructs the reduction schedule based on the arrival order of the processes. This algorithm lets the arriving processes contribute their data and leave the collective as soon as possible without a priori knowledge of the arrival pattern. The experimental results show up to 73% and 44% improvement for MPI_Reduce and MPI_Allreduce operations over the best performing native algorithms, respectively.
- Based on node-wide and cluster-wide PAP for MPI_Allreduce, we show that hierarchical cluster-wide algorithms outperform their flat counterparts due to imposing less data dependency on the processes. We extend our PAP-aware hierarchical cluster-wide algorithm with the proposed PAP-aware intra-node collectives. Micro-benchmarks results show up to 57% performance improvement under imbalanced process arrival patterns. Horovod with TensorFlow application results show up to 17% performance gain.

The rest of this paper is organized as follows. Section 2 provides background information for MPI collectives and process arrival pattern. Section 3 discusses the related works on PAP-aware collective algorithms. Section 4 presents the MPI_Allreduce PAP in Horovod as the motivation behind our work. Section 5 presents our PAP-aware proposals for MPI_Allreduce. Our experimental results and analysis will be presented in Section 6. Finally, Section 7 concludes the paper and comments on future directions.

2 BACKGROUND

2.1 MPI Collective Communication

Collective communication operations involve multiple processes and perform one-to-many, many-to-one, and many-to-many communications in an optimized, yet convenient way. MPI_Allreduce and MPI_Reduce are among the most commonly used collective communication operations in HPC applications [6, 11, 30]. Various algorithms and techniques have been proposed in literature to improve their performance for different topologies, network interconnects, hardware technologies, process count, and message sizes [13, 15, 20, 26, 31, 38]. Most existing algorithms have been designed and evaluated under an impractical premise that all the processes arrive at the collective call simultaneously. While this assumption is correct for microbenchmark analysis, even in programs with perfectly balanced workloads the processes arrive at the collective site at notably different times [12].

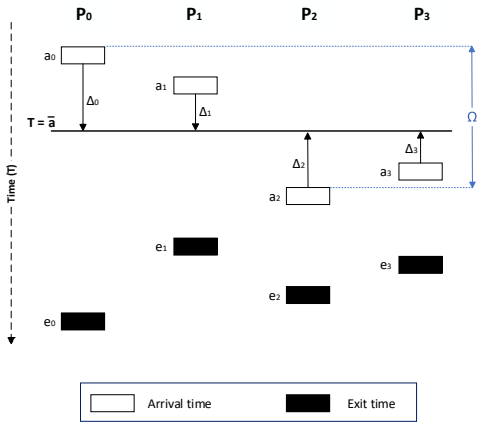


Figure 1: Process arrival pattern parameters

2.2 Process Arrival Pattern

The process arrival pattern can significantly affect the performance of collective communications in that it determines the time each process can start its contribution to the operation. Therefore, it is important to propose new algorithms to deliver high performance for a broad range of process arrival patterns. In this section, we introduce the metrics used in the literature to describe the arrival pattern of processes in the applications. With a given world size of n, processes $P_0, P_1, ..., P_{n-1}$, the PAP can be presented by the tuple $(a_0, a_1, ..., a_{n-1})$ as illustrated in Figure 1. The PAP is considered to be balanced if all the processes arrive at the collective site simultaneously, and imbalanced otherwise. The imbalance in the PAP can be described by *average-case imbalance time* and *worst-case imbalance time* metrics that were originally defined in [12].

The *worst-case (maximum) imbalance time* (Ω), defined in Equation (2.1), denotes the difference in time between the earliest arriving process and the last process entering the collective. The *average-case imbalance time* (Δ), on the other hand, is the average time difference between the arrival time of each process and the average of arrival times. The average of arrival times ($\bar{a}$), and the *average-case imbalance time* have been illustrated in Equation (2.2 (a)) and Equation (2.3), respectively.

$$\Omega = max_i\{a_i\} - min_i\{a_i\} \tag{2.1}$$

$$\bar{a} = \frac{a_0 + a_1 + ... + a_{n-1}}{n} \tag{2.2 (a)}$$

$$\Delta_i = |a_i - \bar{a}| \tag{2.2 (b)}$$

$$\bar{\Delta} = \frac{\Delta_0 + \Delta_1 + ... + \Delta_{n-1}}{n} \tag{2.3}$$

The *average-case imbalance factor* and the *worst-case (maximum) imbalance factor* are two other metrics that are very useful for

characterizing the imbalanced PAPs. Let α be the time it takes to communicate a message (equal to the size of the message in the collective). The *average-case imbalance factor* $\frac{\bar{\Delta}}{\alpha}$, and *worst-case imbalance factor* $\frac{\Omega}{\alpha}$ are defined as the *average-case imbalance time* and *worst-case imbalance time* normalized by the time α.

3 RELATED WORK

Faraj et al. [12] studied the process arrival pattern characteristics of FT, LAMMPS, NBODY, and NTUBE MPI applications on two HPC platforms. The authors introduced the average and worst-case imbalance factors described in Section 2.2 to measure the imbalance in the *Process Arrival Time* (PAT). They showed that the differences between the PATs at a collective operation are usually significant, even for applications with perfectly balanced workloads.

Patarasuk and Yaun [24] investigated the performance of various MPI_Bcast algorithms under different PAPs and showed that they could not achieve performance for most PAPs. They proposed two PAP-aware algorithms for broadcast with large message sizes, where the root sends the data to the processes as they enter the collective call. However, if multiple processes arrive at the collective call simultaneously, the root initiates a sub-group broadcast among the newly-arrived processes and assigns a process within the sub-group to forward the data to the rest of the sub-group processes. This work does not study reduce and allreduce operations. Furthermore, the proposed algorithm is not built on hierarchical algorithms to take advantage of the fast intra-node shared memory on modern systems with hierarchical architectures to reduce the communication latency. Instead, they use MPI point-to-point primitives to communicate control messages as well as data.

Qian and Afsahi [29] proposed RDMA-based PAP-aware algorithms for MPI_Allgather and MPI_Alltoall routines for different message sizes on InfiniBand clusters. Instead of sending/receiving additional control messages, the authors used the RDMA control registers for notifying each process of the arrival of other processes. They also extended their work for having a better performance for small messages by making their design shared-memory aware. Although the algorithms proposed in this work achieve good performance in the presence of imbalance, they are aimed for the systems supporting Infiniband RDMA. Otherwise, they would require alternative synchronization/control mechanisms.

Proficz [27] proposed two PAP-aware algorithms for the MPI Allreduce operation, namely Sorted Linear Tree (SLT) and Pre-Reduced Ring (PRR) algorithms. The author introduced a background thread for each process to monitor the progress of the computation phase. The background thread estimates the remaining computation time for its process and communicates this information to other processes. Using the gathered data, each background thread is able to approximate the PAP for all processes, which is then used by the proposed algorithms. The SLT algorithm is based on the linear tree algorithm. Using the estimated PAP, the SLT algorithm sorts the processes based on their arrival times. Then, the algorithm allows the faster processes to start the communication before the later ones arrive. It is shown that the SLT algorithm can deliver a speedup of up to 1.16 over the standard linear tree algorithm in specific cases. The PRR algorithm is an extension of the ring algorithm. In the PRR algorithm, the number of reducing pre-steps by the faster processes is calculated based on the predicted process arrival times. This way, a speedup of up to 1.14 could be achieved over the regular ring algorithm for certain PAP. The proposed SLT and PRR algorithms delivered 4.2% and 4.0% improvement over a ring algorithm for training a convolutional neural network with the CIFAR-10 [1] dataset. One problem with this method is that introducing an extra thread would lead to oversubscription of the processing cores and consequently, performance penalties. In addition, having the background threads to monitor the progress of the computation phase may require instrumenting the source code of applications incurring additional performance overheads.

Marendic et al. [22] studied the performance of reduction algorithms under imbalanced PAPs. They propose two load balancing reduction algorithms, static and dynamic. The static algorithm depends on a priori knowledge of PATs of all the participating processes and is shown to achieve near-optimal latency. This algorithm is based on the unrealistic assumption that the PATs can be predicted before the call to the collective operation. The dynamic algorithm is an extension of the binomial tree algorithm and does not require a priori information about the PATs. The authors use small control messages to signal the PATs between the involved processes so the early arriving processes can reduce their data and redirect their results to the later arriving ones. One major problem with this work is that it does not consider all the possible PAPs in their design. In the proposed algorithm, the imbalance in the PAP can only be absorbed if the neighbor processes exhibit similar arrival pattern; otherwise, the communication progression will be hindered or at least will not be optimal. It was shown that for specific PAPs and when there is only a single slow process, the proposed algorithm outperforms the other algorithms.

There are other works on improving collective performance in DL applications based on their inherent load imbalance [21, 28], however, they use a different approach in tackling them than our work. Pumma et al. study the effect of load imbalance in scalability limitations of TensorFlow and Horovod. Their proposals improve the performance for training various neural networks by reducing the resource contention under imbalanced workloads. Li et al. propose eager Stochastic Gradient Descent (SGD) which outperforms traditional synchronous SGDs by reducing the number of global synchronizations required for decentralized gradient accumulation.

4 MOTIVATION: COMMUNICATION CHARACTERIZATION OF HOROVOD

The communication characterization of parallel applications is essential in that the findings can be used to improve their performance. For that, we study the frequency and run-time contribution of different collective operations in Horovod with TensorFlow. We introduce the most critical collective and further investigate the frequency of message sizes and their contribution to the collective's overall run-time, and its process arrival pattern.

4.1 Experimental Setup

Experiments were conducted on Cedar, a heterogeneous cluster at Compute Canada with 192 nodes, each having two Intel Silver 4216 Cascade Lake for a total of 186 GB of memory and 32 CPU cores, running at 2.1 GHz. Each node also has four NVIDIA V100 32GB

HBM2 GPUs connected via NVLink. A 100 Gb/s Intel Omni-Path interconnect connects all the nodes together.

We use the CUDA-aware implementation of MVAPICH2-2.3 with CUDA 10.0.130, Horovod-0.18.0 with synthetic benchmarks and TensorFlow-1.13.0. The benchmark runs for 10 iterations and uses 10 batches of size 32 as default, and for all the tests we assigned one process per GPU and distributed the available CPU cores evenly among processes. Using the PMPI interface [2], we developed a custom profiler to study the MPI collective characteristics and process arrival patterns. We use *MPI_Wtime* for timing measurement and a barrier synchronization after *MPI_Init* to synchronize the clocks globally between the processes across the nodes, with an accuracy equal to the latency of sending a few small messages.

4.2 Collective Communication Characterization

Figure 2 shows that for the ResNet50 model, MPI_Allreduce accounts for 75% of the number of collective calls and 96% of the communication run-time on Cedar cluster. Broadcast is the second important operation which makes up 17% of the collectives and contributes up to 4% to the communication run-time. Allgather, gather, and gatherv operations account for the remaining 8% of the collectives and less than 1% of the communication run-time. The results for other DL models, VGG16 and DenseNet201, and GPU counts follow the same trend, and so we do not report them.

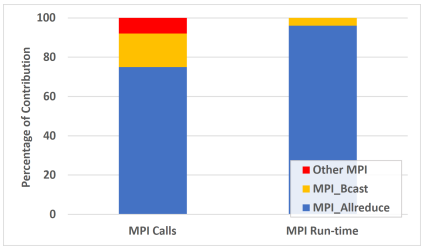


Figure 2: Frequency and run-time contribution of different MPI collectives for Horovod + TensorFlow with ResNet50 on Cedar (256 GPUs, 64 nodes)

We further studied the frequency and run-time contribution of MPI_Allreduce among different message sizes. Our profiling results show that Horovod uses allreduce with messages in the 4B-64MB range. Figure 3 shows that allreduce with small message sizes account for the majority of all the allreduce calls and their communication run-time by 89% and 65% for ResNet50 DL model, respectively. Medium message sizes account only for 1% of the allreduce calls which translates to less than 1% of the allreduce communication time. The second important message size sub-range in terms of frequency of calls is the large message allreduce which makes up 10% of all the allreduce calls, contributing to 35% of the allreduce run-time for ResNet50. These results suggest that small and large message sizes should be the focus of our study.

4.3 MPI_Allreduce Process Arrival Pattern

In this section, we investigate the cluster-wide and then the node-wide PAP metrics for MPI_Allreduce in Horovod + TensorFlow for

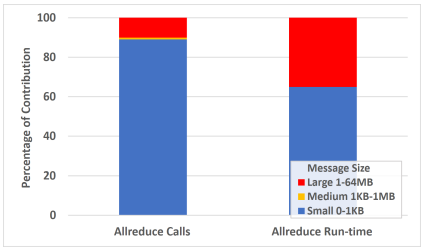


Figure 3: Contribution of different message sizes to the MPI_Allreduce frequency and run-time for Horovod + TensorFlow with ResNet50 on Cedar (256 GPUs, 64 nodes)

ResNet50. Similar results are observed for VGG16 and DenseNet201, but for space reasons we do not report them here.

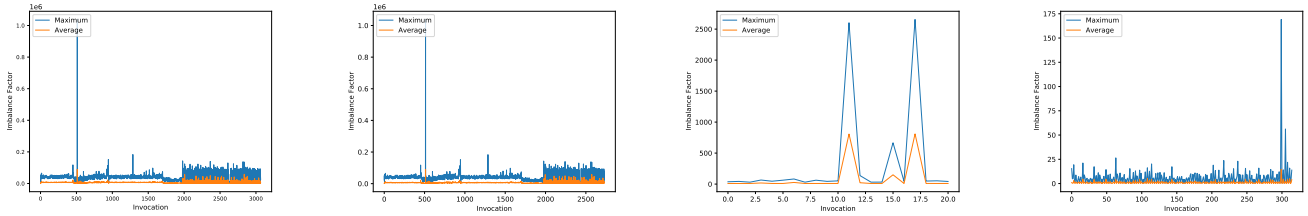
4.3.1 Cluster-wide Study. Table 1 presents the worst-case and average-case imbalance factors, averaged across the invocations by all processes in the cluster for allreduce in Horovod for ResNet50. As it can be seen, both the average worst-case and average-case imbalance factors are very large, particularly for small messages, which suggests that the process arrival pattern is noticeably imbalanced. In other words, the PAP for medium/large messages are generally less imbalanced than those with small messages. However, it should be mentioned that the communication latency for large messages (e.g., 64MB) is much larger than small messages.

Table 1: The average cluster-wide *worst-case* and *average-case* imbalance factors for MPI_Allreduce for Horovod + TensorFlow with ResNet50 on Cedar (256 GPUs, 64 nodes)

Message Range	Imbalance Factor	
	Cluster-wide Metrics	
	Worst	Average
All Message Sizes	33631.77	5516.57
Small Messages	37767.87	6194.69
Medium Messages	327.21	94.41
Large Messages	5.85	1.07

Figure 4 depicts the maximum and average imbalance factors at each invocation of allreduce in Horovod for ResNet50 on Cedar cluster. It is evident that the majority of invocations have similar maximum/average imbalance factors, while there are only a few periods of spikes that occur from time to time. Although the imbalance pattern for each of the invocations seems random, a phased behavior can be noted in the process arrival patterns. In other words, the process arrival patterns tend to remain in a roughly steady range for a long span of time before they fluctuate drastically.

4.3.2 Node-wide Study. We extract the node-wide worst-case and average-case imbalance factors for the processes and report the average across the nodes. Table 2 shows that the average imbalance factors on each node are significantly lower than their cluster-wide counterparts. In other words, processes on the same node arrive at the collective calls with considerably less delay with respect to each other, compared to all other processes on the cluster. This is likely



(a) All message sizes (b) Small message sizes (c) Medium message sizes (d) Large message sizes

Figure 4: The maximum/average imbalance factors for MPI_Allreduce in Horovod + TensorFlow for ResNet50 with 256 GPUs, evenly distributed among 64 nodes on Cedar

because intra-node processes suffer from similar hardware/software related issues resulting in their imbalanced arrival time.

Overall, the node-wide and cluster-wide results motivate us to design PAP-aware collective algorithms for MPI_Allreduce to enhance the performance of Horovod +TenorFlow DL application.

Table 2: The average node-wide *worst-case* and *average-case* imbalance factors for MPI_Allreduce for Horovod + TensorFlow with ResNet50 on Cedar (256 GPUs, 64 nodes)

Message Range	Imbalance Factor	
	Node-wide Metrics	
	Worst	Average
All Message Sizes	7747.23	2848.16
Small Messages	8698.83	3197.90
Medium Messages	220.12	92.99
Large Messages	1.84	0.77

5 PROPOSED PAP-AWARE MPI_ALLREDUCE ALGORITHMS

5.1 Node-wide PAP-aware MPI_Allreduce for Small and Medium Messages

As the communication characterization of Horovod in Section 4.3 showed, the process arrival time of the MPI processes can be significantly imbalanced, particularly for small messages in MPI_Allreduce in Deep Learning workloads. In this section, we first introduce the native intra-node MVAPICH algorithm for small message MPI_Allreduce and then propose a PAP-aware adaptive design. In MVAPICH, the algorithm of choice for small message sizes is a two-step shared-memory algorithm. In the first phase, the leader process, the process with the intra-node rank of zero, waits for all the processes on its node to arrive at the collective call. Once all the processes arrive and copy their data into the shared-memory, the leader process reduces the data in the shared-memory. In the second phase, the leader process sends the reduced data to all processes on the node via a shared-memory broadcast operation. One major problem with the aforementioned algorithm is that it does not take into account the arrival pattern of the processes. In fact, it performs poorly when some processes enter the MPI_Allreduce well after the other processes. This is because the leader process will not start the intra-node reduce operation until all processes have already entered the collective operation. We address this issue by having the leader process poll on the arrival of processes. This way, the leader can reduce the data as soon as the processes arrive. However, the proposed approach does not provide an opportunity for enhancement if the leader process arrives last, or close to last. This led us to consider process arrival time in our design and choose the leader process dynamically at run-time. This way, we let the early arriving processes contribute to the progression of the reduce operation without the need to wait for all the processes to arrive.

Our proposed intra-node PAP-aware MPI_Allreduce algorithm for small and medium messages, *Small-PAP-aware*, consists of two steps, as follows:

- **Phase 1**: Intra-node PAP-aware shared-memory reduce by the leader process of the node
- **Phase 2**: Intra-node shared-memory broadcast

However, unlike MVAPICH, in our design for the reduce step, at each invocation of the MPI_Allreduce, we dynamically assign the earliest arriving process of each node as the leader of that node, responsible for the reduction operation. Other processes only need to copy their data into the shared-memory upon their arrival and set a flag to make the leader process aware that their data is in the shared-memory and ready to be reduced. The leader process polls on the flags and executes the reduction operation whenever data is ready. For the broadcast phase, we use a shared-memory broadcast by the leader process, where the leader simply writes its data into the shared-memory and all the other processes on the node copy the data from the shared-memory into their own address space.

Algorithm 1 presents our proposed design. In order to implement the synchronization between processes on each node for assigning the earliest process as the leader process of that node, we use two variables *Leader_Defined_Flag* and *Is_Leader*. *Leader_Defined_Flag* is a shared variable among the processes residing on the same node. This flag is protected by lock/unlock to avoid race conditions, so that the first arriving process can be safely determined. *Is_Leader*, on the other hand, is a local variable to each process which demonstrates whether the process has been assigned as the leader responsible for the execution of the reduction operation. In addition to the variables stated above, we use two other shared buffers, *Data_Buffer* and *Data_Ready_Flags*. This way, the processes could share their data and their availability with the leader process through the shared-memory. *Data_Buffer* is the shared buffer where the processes copy their data into their preallocated segments so that the leader process

Algorithm 1: Node-wide PAP-aware MPI_Allreduce for Small and Medium Messages

Input : The data residing in send buffers (*sendbuf*)
Output: The data residing in receive buffers (*recvbuf*)
Variables:
***Leader_Defined_Flag*:** A shared flag protected by lock/unlock to determine the first arrived (the leader) process.
***Data_Buffer*:** A shared buffer populated by processes with their data.
***Data_Ready_Flags*:** A shared buffer consisting of flags, each dedicated to one process, demonstrating whether the process's data in the *Data_Buffer* is ready.
***Is_Leader*:** A local variable determining the leader process.

```
begin
1   if Leader_Defined_Flag == 0 then
2       if trylock(&mutex) then
3           if Leader_Defined_Flag == 0 then
4               Leader_Defined_Flag = 1;
5               Is_Leader = 1;
            end
6           unlock(&mutex);
        end
    end
7   if Is_Leader == 1 then
8       while Operation_Progression < (local_size − 1) do
9           for (i = 0; i < local_size; i++) do
10              if Data_Ready_Flags[i] == 1 then
11                  Operation_Progression++;
12                  Data_Ready_Flags[i] = 0 ;
13                  reduce(recvbuf, Data_Buffer[i]);
14                  if Operation_Progression == (local_size − 1)
                     then
15                      Copy(recvbuf, ..., Result,...);
16                      for (j = 0; j < local_size; j++) do
17                          Completion_flags[j] = 1;
                        end
                    end
                end
            end
        end
18  else
19      Copy(sendbuf, ..., Data_Buffer[local_rank],...);
20      Data_Ready_Flags[local_rank] = 1;
21      while Completion_flags[local_rank] == 0 do
22          Wait for own Completion flag to be set
        end
23      Copy(Result, ..., recvbuf,...);
24      Completion_flags[local_rank] = 0;
    end
end
```

can execute the reduction operation on them. *Data_Ready_Flags* is the shared buffer filled with flags, each dedicated to one process. These flags demonstrate whether the data has been successfully copied into the *Data_Buffer* by the corresponding processes.

Once a process enters the collective operation, it first reads the *Leader_Defined_Flag* to check whether the leader process has been defined. If the leader process has not been defined yet, it means that the arrived process is among the earliest arriving processes and hence could be assigned as the leader. Therefore, the process tries to lock the lock variable. Upon acquiring the lock, the process re-validates the *Leader_Defined_Flag* to ensure that no process has been assigned as the leader so far. It then assigns itself as the leader and informs other intra-node processes by setting the *Leader_Defined_Flag* and releases the lock, as shown in Lines 1 to 6.

The processes can now perform the actions assigned to them based on whether they are the leader process or not. Non-leader processes copy the data in their *sendbuf* into the appropriate segment of *Data_Buffer* in the shared-memory and inform the leader of its availability by setting its dedicated flag in *Data_Ready_Flags* (Lines 18 to 20). The leader process, on the other hand, is responsible for performing the reduction operation by polling on the *Data_Ready_Flags*. Whenever a data is ready in the *Data_Buffer*, the leader reduces it into its *recvbuf*. This operation continues until there is no more data to be reduced. At this point, the result of the intra-node reduce step of the hierarchical MPI_Allreduce is available in the *recvbuf* of the leader process (Lines 7 to 13).

The second phase (shared-memory broadcast) begins by the leader process copying the result of the reduce operation from its own *recvbuf* into the dedicated area in *Data_Buffer* defined as *Result* and sets the *Completion_flags* for each of the processes to inform them that the collective result is ready in the shared-memory (Lines 14 to 17). Then, the leader process resets the appropriate flags *Is_Leader* and *Leader_Defined_Flag* for the next invocation of the collective. The non-leader processes, on the other hand, poll on their dedicated *Completion_flags* and once it is set, they copy the MPI_Allreduce result from shared-memory into their own *recvbuf* (Lines 21 to 22). Then, they reset their own *Completion_flags* for the next MPI_Allreduce collective call (Line 23). Figure 5 presents an example execution of the proposed PAP-aware MPI_Allreduce for four processes. It should be mentioned that our design is thread-safe, and hence it can be used in multi-threaded environments.

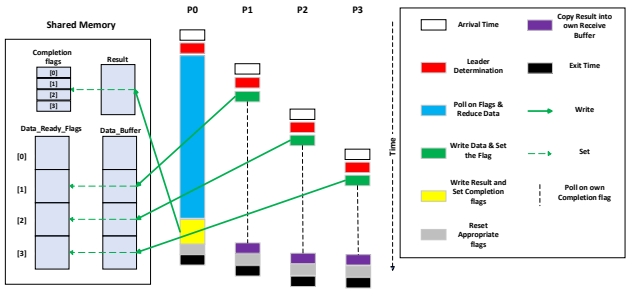


Figure 5: Example run of the proposed small/medium message PAP-aware MPI_Allreduce algorithm for four processes

5.2 Node-wide PAP-aware MPI_Allreduce for Large Messages

Copying large messages in and out of shared-memory is expensive. Therefore, we propose a two-step node-wide PAP-aware Allreduce

Algorithm 2: Node-wide PAP-aware MPI_Reduce for Large Messages

Input : The data residing in send buffers (*sendbuf*)
Output : The data residing in the root's receive buffer (*recvbuf*)
Variables:
***Process_Counter*:** A shared counter protected by lock/unlock to keep the track of the number of arrived processes.
***Sorted_Ranks_Buffer*:** A shared buffer filled with ranks of the processes based on their arrival order.
***Arrival_Rank*:** A local variable for storing the value of *Process_Counter*.

```
begin
1   lock(&mutex);
2       Arrival_Rank = Process_Counter;
3       Process_Counter + = 1;
4   unlock(&mutex);
5   Sorted_Ranks_Buffer[Arrival_Rank] = MPI_Rank;
6   if First Process to Arrive then
7       while (Sorted_Ranks_Buffer[Arrival_Rank + 1] = = NULL);
8       Send(... , Sorted_Ranks_Buffer[Arrival_Rank + 1] , ...);
9   else if Last Process to Arrive then
10      Receive(... , Sorted_Ranks_Buffer[Arrival_Rank - 1] , ...);
11      Reduce the received data with your own data.
12      if MPI_Rank ! = root then
13          Send(... , root , ...);
        end
14  else
15      Receive(... , Sorted_Ranks_Buffer[Arrival_Rank - 1] , ...);
16      Reduce the received data with your own data.
17      while (Sorted_Ranks_Buffer[Arrival_Rank + 1] = = NULL);
18      Send(... , Sorted_Ranks_Buffer[Arrival_Rank + 1] , ...);
    end
19  if (MPI_Rank = = root) && (!Last Process) then
20      Receive(... , Last Process , ...);
    end
end
```

algorithm for large messages, *Large-PAP-aware*, consisting of a PAP-aware reduce algorithm followed by a broadcast operation, that achieves zero-copy data transfer by copying the data directly from the send buffer of a process to the receive buffer of the next arriving process. In our PAP-aware reduce algorithm, we allow the early arriving processes to start the reduce operation before the later processes arrive, and leave the collective as soon as they make their contribution. One challenge with all PAP-aware algorithms is to develop a way to inform every other process of the ranks that have already arrived at the collective. One method for doing so is to exchange point-to-point control messages between the processes [24]. However, this introduces an extra overhead that affects the performance. To minimize the overhead of control messages, we use a shared-memory structure between the processes on each node.

Algorithm 2 presents our proposed PAP-aware MPI_Reduce for medium to large messages that minimizes the data dependency between the processes. Whenever a process arrives, it receives the reduced data from the last process that has arrived. After performing its part to the reduce operation, the process passes the updated reduced data to the next arriving process and leaves the collective call. In order to implement the synchronizations between the processes necessary for the execution of our algorithm, we use two shared-memory variables, *Process_Counter* and *Sorted_Ranks_Buffer*. *Process_Counter* is a shared counter that is protected by lock/unlock so the processes can safely access its value even at the presence of race conditions. *Sorted_Ranks_Buffer* is a shared buffer containing as many cells as the number of processes on the node. Each cell is dedicated to a process based on its arrival order. The first cell of the *Sorted_Ranks_Buffer* is assigned to the earliest arriving process, while the final arriving process occupies the last cell. Each process writes its *MPI_Rank* in its designated cell when it arrives. Since *Sorted_Ranks_Buffer* is shared, each process will have access to *Arrival_Rank* (determined by the position of the cell) and *MPI_Rank* (determined by the value of the cell) of all processes on the node.

Once a process arrives at the collective call, it tries to lock the lock variable. Upon acquiring the lock, the process reads the *Process_Counter*'s value, recognizes its rank among the already arrived processes, and stores it into a local variable called *Arrival_Rank*, then increments the counter and releases the lock (Lines 1 to 4 of Algorithm 2). Next, the process writes its *MPI_Rank* in the appropriate cell of *Sorted_Ranks_Buffer* that its *Arrival_Rank* suggests (Line 5). Now that the process is aware of its arrival rank, it can perform the actions assigned to it based on its arrival time. For the earliest process, since there is no predecessor, it only needs to wait for the next process to arrive (through polling on the *Sorted_Ranks_Buffer*) to pass its data and leave the collective call (Lines 6 to 8). For the processes that arrive between the first and the last processes, the first step is to receive the reduced data from their predecessor (the predecessor process can be recognized through the *Sorted_Ranks_Buffer*), and then reduce their own data with the received data and wait for their successor to arrive. Upon their arrival, they will send the updated data to it. At this point, they can leave the collective (Lines 14 to 18). When the last process arrives, it reduces its data with the reduced data from the previous process and sends the result of the final reduce operation to the root process (process zero, without loss of generality) for the broadcast operation (Lines 9 to 13). The reduce operation will be completed upon reception of the result by the root process (Lines 19 and 20). Figure 6 presents an example execution of the proposed node-wide PAP-aware MPI_Reduce algorithm for four processes. It should be mentioned that although our design has an $O(n)$ time complexity for the reduction operation, the minimum data dependency between the involved processes results in better performance in imbalanced PAPs, compared to algorithms with better time complexities. In addition, this is not too detrimental given the per node GPU count in modern clusters.

For the broadcast operation, we utilize two broadcast algorithms used frequently in MPI implementations. A shared-memory broadcast is used for up to 64KB messages, where each process copies the data from shared-memory to its receive buffer, as explained in Section 5.1. For larger messages, the commonly used Binomial-Tree broadcast algorithm is used.

5.3 Cluster-wide PAP-aware MPI_Allreduce

In MVAPICH, the algorithm of choice for MPI_Allreduce for medium to large messages (larger than 64KB) is a flat Reduce-Scatter followed

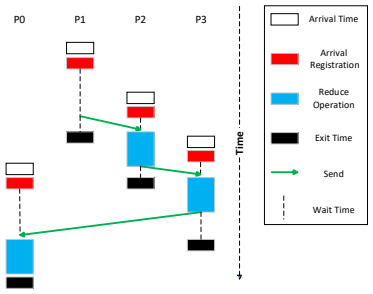


Figure 6: Example run of the proposed large message PAP-aware reduce algorithm for four processes

by an Allgather (RSA), as it delivers the best performance among the other algorithms according to the microbenchmarks used to derive its tuning table. These microbenchmarks only measure the performance under perfectly balanced workloads when processes arrive at the collective call simultaneously. In practical scenarios with real application workloads on different environments, however, the performance of flat algorithms are susceptible to imbalanced PAPs because of the inherent data dependency that they introduce among all the processes, which acts like an unnecessary implicit synchronization between the processes in the cluster. Hierarchical algorithms, on the other hand, introduce less data dependency among the processes, where the first and the last phases of the algorithm are usually an intra-node collective operation that only requires synchronization between the processes residing on the same node. Therefore, we argue that the hierarchical algorithms are less prone to performance degradation in the presence of imbalanced PAPs and hence, they are a better algorithm for applications with imbalanced PAPs. This is supported by the imbalance factors shown in Table 1 and Table 2, where the node-wide imbalanced factors are much less than their cluster-wide counterparts for the Horovod with TensorFlow workload.

For this study, we consider a three-phase cluster-wide hierarchical RSA algorithm, *Hierarchical-RSA*, for MPI_Allreduce, as follows. We then extend the *Hierarchical-RSA* algorithm with our proposed node-wide PAP-aware algorithms in Section 5.1 and Section 5.2 for its intra-node phases, and refer to them as *Hierarchical-RSA+Small-PAP-aware* and *Hierarchical-RSA+Large-PAP-aware*, accordingly.

- Phase 1: Intra-node reduce by the leader process
- Phase 2: Inter-node MPI_Allreduce among the leader processes (RSA)
- Phase 3: Intra-node broadcast by the leader process

6 EVALUATION RESULTS AND ANALYSIS

In this section, we evaluate the performance of our proposed designs for PAP-aware MPI_Allreduce collective against state-of-the-art algorithms under imbalanced PAPs with different Maximum Imbalance Factors (MIF). The experimental platform for our tests in this section is the Cedar cluster described in Section 4.1.

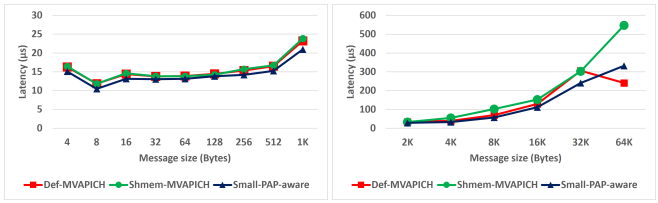
6.1 Micro-benchmark Results

6.1.1 Node-wide PAP-aware MPI_Allreduce for Small and Medium Messages. In this section, we evaluate our intra-node PAP-aware MPI_Allreduce algorithm, *Small-PAP-aware*, for message sizes up to 64KB. For larger messages, the shared-memory-aware algorithms in MVAPICH as well as our algorithm lose to non-shared-memory-aware algorithms by a large margin. Therefore, we do not present the results past 64KB messages. As discussed earlier, the default intra-node MPI_Allreduce algorithm in MVAPICH for messages up to 1KB is a two-step shared-memory algorithm. However, for message sizes from 1KB to 64KB, MVAPICH switches to a flat recursive doubling MPI_Allreduce algorithm. We call the set of two aforementioned algorithms *Def-MVAPICH* in the rest of this section. In order to evaluate our PAP-aware shared-memory algorithm fairly, we also compare its performance against an algorithm that uses the same two-step shared-memory algorithm in MVAPICH, but for messages up to 64KB. We call this algorithm *shmem-MVAPICH*. Performance comparison is done under imbalanced PATs. For this purpose, we use the balanced Ohio State University Micro-Benchmark (OMB) suite [7] and induce a random computation before MPI_Allreduce for each process to create an imbalanced workload. The upper bound for the random computation is determined by the MIF. For all micro-benchmark studies in this paper, we assume that the data for the collective operation resides in the host buffers to show the algorithmic impact of our design over the native algorithms.

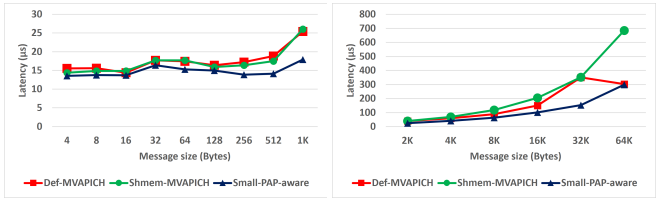
Figure 7 compares our proposed PAP-aware algorithm against the *Def-MVAPICH* and the *shmem-MVAPICH* with 32 processes and imbalanced workloads with MIFs equal to 10, 20, and 50 on Cedar. It should be mentioned that our design provides comparable results with the default algorithms with 4 processes per node (PPN) due to the overhead induced by locking/unlocking mechanism. However, with increasing process count, our design outperforms the studied algorithms by a larger margin. For that, we present the results with 32 PPN in this section. As it can be seen in the figure, for a small MIF of 10, our PAP-aware algorithm outperforms both the *Def-MVAPICH* and the *shmem-MVAPICH* algorithms almost for all message sizes, with a 22% improvement over the best performing algorithm. As we increase the MIF to 20 and 50, we observe the maximum performance improvement of 56% and 37% over the other two algorithms, respectively. The improvement for MIF 50 is smaller because the delay in the arrival time of the processes is so large that it dominates the latency of the whole collective call. We observe the same pattern for MIFs larger than 50, where the PAP-aware algorithm outperforms the other algorithms for all message sizes.

6.1.2 Node-wide PAP-aware MPI_Allreduce for Large Messages. Figure 8 compares our intra-node PAP-aware MPI_Allreduce algorithm, *Large-PAP-aware*, for large messages against the default RSA algorithm, *Def-MVAPICH (RSA)*, the Binomial Tree + Broadcast algorithm, *BT+Bcast*, and the Reduce Scatter/Gather + Broadcast algorithm, *RSG+Bcast*, under imbalanced PAPs with MIFs equal to 10, 20, and 50 on Cedar with 4 processes per node.

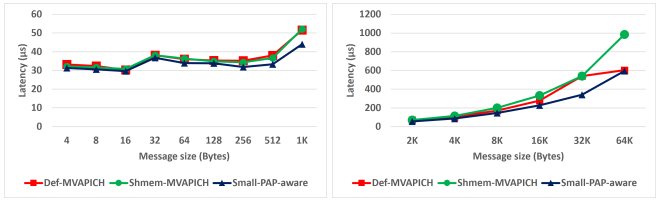
For a small MIF of 10, our PAP-aware algorithm outperforms all the other algorithms for all message sizes larger than 64KBs, with an average improvement of 18% over the best performing algorithm for the message range of 64KB - 64MB. The maximum improvement of 41% was observed for the message size of 1MB. As we increase the MIF to 20, we observe 20% average improvement over the best



(a) MIF = 10

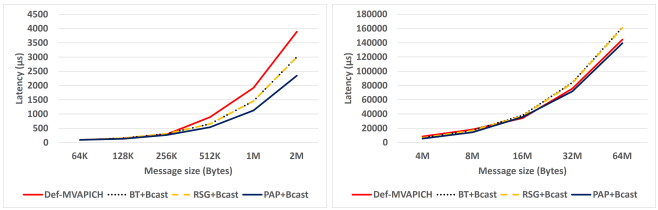


(b) MIF = 20

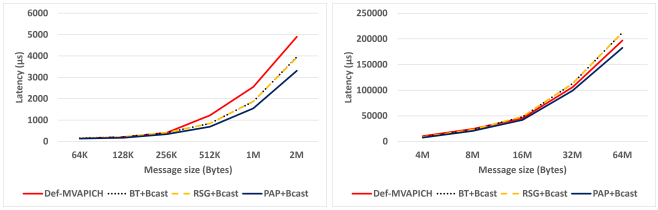


(c) MIF = 50

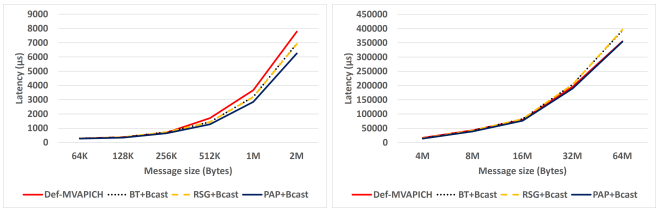
Figure 7: Comparison of the proposed *Small-PAP-aware* MPI_Allreduce against the Def-MVAPICH and shmem-MVAPICH algorithms under imbalanced workload, different MIFs and 32 processes on a single node on Cedar



(a) MIF = 10



(b) MIF = 20



(c) MIF = 50

Figure 8: Comparison of the proposed *Large-PAP-aware* MPI_Allreduce against the Def-MVAPICH (RSA), "BT+Bcast", and "RSG+Bcast" algorithms under imbalanced workload, different MIFs and 4 processes on a single node on Cedar

performing algorithm, with a maximum improvement of 44% for 512KB message. With a MIF of 50, the average improvement is 11%, and the maximum improvement is 25% for the 512KB message over the best performing algorithm. For significantly larger MIFs, although our PAP-aware algorithm outperforms all the other studied algorithms, the speedup decreases as we increase the MIF.

6.1.3 Cluster-wide PAP-aware MPI_Allreduce for Large Messages. We compare the performance of our *Hierarchical-RSA* algorithm, against the default flat RSA algorithm in MVAPICH under an imbalanced workload and for medium to large messages (larger than 64KB) on 2 to 32 nodes with 4 PPN on Cedar cluster. The MIF is chosen in a way to mimic the cluster-wide imbalanced process arrival pattern of Horovod for ResNet50 for large messages that was presented in Section 4.3. The results show that for all experiments, the hierarchical algorithm has a smaller latency than the flat RSA algorithm for most of the message sizes between 64KB to 64MB. Table 3 shows the average, minimum, and maximum improvement of the hierarchical algorithm over the flat algorithm, along with the corresponding message size for the minimum and maximum improvements. The results show that the average improvement among all message sizes is greater than 20% for all the scenarios. This confirms the superior performance of the hierarchical algorithm over the flat algorithm in the presence of imbalanced PAP. This is because in the hierarchical algorithm the early arriving processes will only need to wait for other processes on their own node to arrive to start their communication, whereas in the flat algorithm the communication progression is hampered until all the processes across the cluster arrive.

Table 3: Performance improvement of *Hierarchical-RSA* over *Def-MVAPICH* for up to 128 GPUs on Cedar, considering MIF of Horovod's ResNet50

GPU Count	Improvement				
	Average	Min	Message Size	Max	Message Size
8	20.01%	16.36%	32MB	49.5%	2MB
16	28.97%	8.15%	64MB	54.01%	128KB
32	28.64%	11.15%	64MB	55.81%	256KB
64	26.94%	19.43%	64MB	57.64%	128KB
128	23.17%	22.23%	32MB	56.76%	256KB

6.2 Horovod Application Results

In this section, we compare the performance of the proposed hierarchical algorithm *Hierarchical-RSA* against the flat RSA algorithm *Def-MVAPICH* for Horovod + tensorFlow. We use the Horovod synthetic benchmarks, which provide the throughput of the image classification task by the number of images processed per second

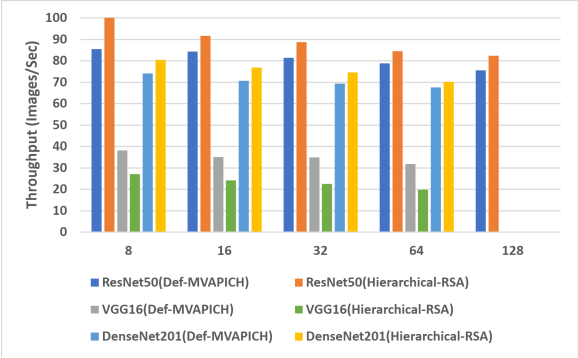


Figure 9: Horovod + TensorFlow per-GPU throughput comparison between *Def-MVAPICH* and *Hierarchical-RSA* algorithms for different DL models and 8 to 128 GPUs, evenly distributed among 2 to 32 nodes on Cedar

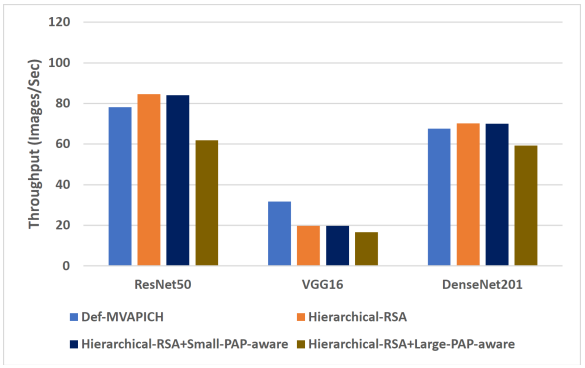


Figure 10: Horovod + TensorFlow total per-GPU throughput comparison of *Def-MVAPICH*, *Hierarchical-RSA*, *Hierarchical-RSA+Small-PAP-aware*, and *Hierarchical-RSA+Large-PAP-aware* algorithms for different DL models and 64 GPUs, evenly distributed among 16 nodes on Cedar

(Images/Sec). Figure 9 exhibits the per-GPU throughput of the flat RSA and the *Hierarchical-RSA* algorithms for different models for 8 to 128 GPUs, except for VGG16 and DensetNet201 datapoints for 128 GPUs that are missing. The results show that for the ResNet50 and the DenseNet201 models, the *Hierarchical-RSA* outperforms the *Def-MVAPICH* algorithm for all the GPU counts by up to 17% and 9%, respectively. For the VGG16 model, on the other hand, we see that the flat RSA outperforms the *Hierarchical-RSA* algorithm. To investigate the reasons behind this, we measured the MIF of the VGG16 and the DenseNet201 and observed that the MIF of large messages for the VGG16 model is less than one, whereas the MIF of large messages for DenseNet201 is similar to ResNet50. This indicates that the PAP for the VGG16 is quite balanced and therefore, the flat RSA algorithm is a better choice for this model.

As the communication characterization of Horovod in Section 4.3 showed, the cluster-wide imbalance factor is significantly larger than the node-wide imbalance factor, which suggests that in Horovod the processes on the same node arrive at the collective call with much less delay with respect to each other than to all the other processes on the cluster. Consequently, Deep Learning applications with the aforementioned characteristics could benefit from the hierarchical algorithms, where the early arriving processes on each node will not need to wait long for the processes on their node to arrive to start the intra-node step; whereas with flat algorithms, the waiting time between all the cluster-wide processes to start the operation is much larger. Hence, the hierarchical algorithm delivers much better performance compared to its flat counterpart under the inherently imbalanced PAP of Horovod.

We study the performance of our *Hierarchical-RSA+Small-PAP-aware* and *Hierarchical-RSA+Large-PAP-aware* algorithms, which extend our *Hierarchical-RSA* algorithm with our node-wide PAP-aware algorithms for its intra-node phases, against the *Def-MVAPICH* and *Hierarchical-RSA* algorithms for Horovod + TensorFlow. From the results presented in Figure 10, it is evident that the *Hierarchical-RSA+Small-PAP-aware* algorithm is on par with the *Hierarchical-RSA* algorithm and cannot enhance its performance. This is because, as discussed in Section 6.1.1, our node-wide *Small-PAP-aware* algorithm delivers comparable performance to the native algorithms for 4 PPN (one process per GPU). However, on computing nodes with more than 4 GPUs or GPUs with the MIG/MPS feature, we would expect to see a superior performance for our *Hierarchical-RSA+Small-PAP-aware* algorithm over the *Hierarchical-RSA* algorithm. We will report our findings in a near future.

The results for *Hierarchical-RSA+Large-PAP-aware* shows performance degradation compared to our proposed *Hierarchical-RSA* and *Hierarchical-RSA+Small-PAP-aware* algorithms, as well as the *Def-MVAPICH*. This is because, as discussed in Section 6.1.2, our node-wide *Large-PAP-aware* algorithm for large messages starts to outperform the other algorithms with MIFs larger than 10, whereas the intra-node MIF of Horovod for large messages is close to two.

7 CONCLUSION AND FUTURE WORK

Researchers have long been investigating to improve the performance of MPI collective communication operations from different aspects. Most of these studies, however, are based on the premise that all processes arrive at the collective call at the same time. Research has shown that such an assumption is impractical in HPC platforms and that the process arrival pattern, even in MPI applications with perfectly balanced workloads, is sufficiently imbalanced to affect the performance adversely. In this work, we study the communication characterization of Horovod distributed Deep Learning framework with TensorFlow, including MPI_Allreduce and its PAP. We propose two intra-node PAP-aware MPI_Allreduce algorithms for different message sizes. The proposed algorithms achieve up to 56% and 44% performance improvement over the native algorithms under different imbalanced PAPs, respectively. We then propose cluster-wide hierarchical PAP-aware MPI_Allreduce algorithms, which induce less data dependency among processes compared to flat algorithms and are capable of achieving high performance under imbalanced workloads. These algorithms achieve up to 58% and 17% improvement over native algorithms at the micro-benchmark level and for Horovod + TensorFlow, respectively.

As for future work, we would like to show the effectiveness of our PAP-aware designs on clusters with larger GPU counts per node and with more modern GPUs, as well as other Deep Learning workloads. We plan to optimize our algorithms to minimize the

overhead of dynamically constructing the reduction schedule to achieve performance even for applications with small MIFs. We would also like to consider other communication libraries such as Open MPI with UCX, UCC, and NCCL for our PAP-aware collective ideas. We plan to explore PAP-aware algorithms for other collectives such as MPI_Alltoall and MPI_Alltoallv that are used in other Deep Learning applications. Finally, we would like to develop a mechanism that detects/predicts the PAP to invoke/disable appropriate PAP-aware algorithms based on the imbalanced PAP.

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 Cedar compute clusters at Simon Fraser University.

REFERENCES

[1] (June, 2022). CIFAR-10 and CIFAR-100 datasets. https://www.cs.toronto.edu/~kriz/cifar.html
[2] (June, 2022). Message Passing Interface (MPI 4.0). http://www.mpi-forum.org
[3] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/
[4] Q. Ali, S. P. Midkiff, and V. S. Pai. 2009. Efficient high performance collective communication for the Cell blade. In *Proceedings of the 23rd international conference on Supercomputing (ICS)*. 193–203.
[5] T. Ben-Nun and T. Hoefler. 2019. Demystifying parallel and distributed Deep Learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)* 52, 4 (2019), 1–43.
[6] D. E. Bernholdt, S. Boehm, G. Bosilca, M. Gorentla Venkata, R. E. Grant, T. Naughton, H. P. Pritchard, M. Schulz, and G. R. Vallee. 2020. A survey of MPI usage in the U.S. exascale computing project. *Concurrency and Computation: Practice and Experience (CCPE)* 32, 3 (2020), e4851.
[7] D. Bureddy, H. Wang, A. Venkatesh, S. Potluri, and D. K. Panda. 2012. OMB-GPU: A Micro-Benchmark Suite for Evaluating MPI Libraries on GPU Clusters. In *Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface* (Vienna, Austria) *(EuroMPI'12)*. Springer-Verlag, Berlin, Heidelberg, 110–120. https://doi.org/10.1007/978-3-642-33518-1_16
[8] A. Castell'o, E. S. Quintana-Ort'i, and J. Duato. 2021. Accelerating distributed deep neural network training with pipelined MPI allreduce. *Cluster Computing* 24, 4 (2021), 3797–3813.
[9] C.-H. Chu, P. Kousha, A. A. Awan, K. S. Khorassani, H. Subramoni, and D. K. Panda. 2020. NV-Group: Link-Efficient Reduction for Distributed Deep Learning on Modern Dense GPU Systems. In *Proceedings of the 34th ACM International Conference on Supercomputing* (Barcelona, Spain) *(ICS '20)*. Association for Computing Machinery, New York, NY, USA, Article 6, 12 pages. https://doi.org/10.1145/3392717.3392771
[10] C.-H. Chu, X. Lu, A. A. Awan, H. Subramoni, J. Hashmi, B. Elton, and D. K. Panda. 2017. Efficient and scalable multi-source streaming broadcast on GPU clusters for deep learning. In *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, 161–170.
[11] S. Chunduri, S. Parker, P. Balaji, K. Harms, and K. Kumaran. 2018. Characterization of MPI usage on a production supercomputer. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 386–400.
[12] A. Faraj, P. Patarasuk, and X. Yuan. 2008. A study of process arrival patterns for MPI collective operations. *International Journal of Parallel Programming* 36, 6 (2008), 543–570.
[13] I. Faraji and A. Afsahi. 2015. Hyper-Q aware intranode MPI collectives on the GPU. In *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*. 47–50.
[14] I. Faraji and A. Afsahi. 2018. Design considerations for GPU-aware collective communications in MPI. *Concurrency and Computation: Practice and Experience (CCPE)* 30, 17 (2018), e4667.
[15] K. Hasanov and A. Lastovetsky. 2017. Hierarchical redesign of classic MPI reduction algorithms. *The Journal of Supercomputing* 73, 2 (2017), 713–725.
[16] T. Hoefler, T. Schneider, and A. Lumsdaine. 2008. Accurately measuring collective operations at massive scale. In *2008 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 1–8.
[17] T. Hoefler, T. Schneider, and A. Lumsdaine. 2010. Characterizing the influence of system noise on large-scale applications by simulation. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.
[18] G. Inozemtsev and A. Afsahi. 2012. Designing an offloaded nonblocking MPI_Allgather collective using CORE-Direct. In *2012 IEEE International Conference on Cluster Computing (Cluster)*. IEEE, 477–485.
[19] A. Jocksch, N. Ohana, E. Lanti, E. Koutsaniti, V. Karakasis, and L. Villard. 2021. An optimisation of allreduce communication in message-passing systems. *Parallel Comput.* 107, 102812 (2021).
[20] K. Kandalla, A. Venkatesh, K. Hamidouche, S. Potluri, D. Bureddy, and D. K. Panda. 2013. Designing optimized MPI broadcast and allreduce for Many Integrated Core (MIC) InfiniBand clusters. In *21st Annual IEEE Symposium on High-Performance Interconnects (HotI)*. IEEE, 63–70.
[21] S. Li, T. Ben-Nun, S. D. Girolamo, D. Alistarh, and T. Hoefler. 2020. Taming unbalanced training workloads in deep learning with partial collective operations. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 45–61.
[22] P. Marendić, J. Lemeire, T. Haber, D. Vučinić, and P. Schelkens. 2012. An investigation into the performance of reduction algorithms under load imbalance. In *European Conference on Parallel Processing (Euro-Par)*. Springer, 439–450.
[23] B. S. Parsons. 2015. Accelerating MPI collective communications through hierarchical algorithms with flexible inter-node communication and imbalance awareness. Ph.D. Dissertation. Purdue University.
[24] P. Patarasuk and X. Yuan. 2008. Efficient MPI Bcast across different process arrival patterns. In *2008 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 1–11.
[25] F. Petrini, S. Coll, E. Frachtenberg, and A. Hoisie. 2001. Hardware-and software-based collective communication on the Quadrics network. In *Proceedings IEEE International Symposium on Network Computing and Applications (NCA)*. IEEE, 24–35.
[26] J. Pjesivac-Grbovic. 2007. Towards automatic and adaptive optimizations of MPI collective operations. Ph.D. Dissertation. University of Tennessee.
[27] J. Proficz. 2018. Improving all-reduce collective operations for imbalanced process arrival patterns. *The Journal of Supercomputing* 74, 7 (2018), 3071–3092.
[28] S. Pumma, D. Buono, F. Checconi, X. Que, and W.-C. Feng. 2020. Alleviating load imbalance in data processing for large-scale deep learning. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 262–271.
[29] Y. Qian and A. Afsahi. 2011. Process arrival pattern aware alltoall and allgather on InfiniBand clusters. *International Journal of Parallel Programming* 39, 4 (2011), 473–493.
[30] R. Rabenseifner. 1999. Automatic MPI counter profiling of all users: First results on a CRAY T3E 900-512. In *Proceedings of the message passing interface developer's and user's conference*, Vol. 1999. 77–85.
[31] R. Rabenseifner. 2004. Optimization of collective reduction operations. In *International Conference on Computational Science*. Springer, 1–9.
[32] B. Ramesh, K. K. Suresh, N. Sarkauskas, M. Bayatpour, J. M. Hashmi, H. Subramoni, and D. K. Panda. 2020. Scalable MPI Collectives using SHARP: Large Scale Performance Evaluation on the TACC Frontera System. In *2020 Workshop on Exascale MPI (ExaMPI)*. 11–20.
[33] H. Ritzdorf and J. L. Traff. 2006. Collective operations in NEC's high-performance MPI libraries. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 1–10.
[34] F. Seide and A. Agarwal. 2016. CNTK: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2135–2135.
[35] A. Sergeev and M. Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).
[36] S. Sur, U. K. R. Bondhugula, A. Mamidala, H.-W. Jin, and D. K. Panda. 2005. High performance RDMA based all-to-all broadcast for InfiniBand clusters. In *International Conference on High-Performance Computing (HiPC)*. Springer, 148–157.
[37] Y. H. Temucin, A. Sojoodi, P. Alizadeh, and A. Afsahi. 2021. Efficient Multi-Path NVLink/PCIe-Aware UCX based Collective Communication for Deep Learning. In *28th Annual IEEE Symposium on High-Performance Interconnects (HotI)*. IEEE, 1–10.
[38] M. G. Venkata, P. Shamis, R. Sampath, R. L. Graham, and J. S. Ladd. 2013. Optimizing blocking and nonblocking reduction operations for multicore systems: Hierarchical design and implementation. In *2013 IEEE International Conference on Cluster Computing (Cluster)*. IEEE, 1–8.
[39] M.-S. Wu, R. A. Kendall, and K. Wright. 2005. Optimizing collective communications on SMP clusters. In *2005 International Conference on Parallel Processing (ICPP)*. IEEE, 399–407.