PiP-MColl: Process-in-Process-based Multi-object MPI Collectives

Jiajun Huang,* Kaiming Ouyang,† Yujia Zhai,* Jinyang Liu,* Min Si,‡
Ken Raffenetti,§ Hui Zhou,§ Atsushi Hori¶ Zizhong Chen* Yanfei Guo§ Rajeev Thakur§
*University of California, Riverside
†NVIDIA Corporation
‡Meta Platforms, Inc.
§Argonne National Laboratory
¶National Institute of Informatics

***Abstract*—In the era of exascale computing, the adoption of a large number of CPU cores and nodes by high-performance computing (HPC) applications has made MPI collective performance increasingly crucial. As the number of cores and nodes increases, the importance of optimizing MPI collective performance becomes more evident. Current collective algorithms, including kernel-assisted inter-process data exchange techniques and data sharing based shared-memory approaches, are prone to significant performance degradation due to the overhead of system calls and page faults or the cost of extra data-copy latency. These issues can negatively impact the efficiency and scalability of HPC applications. To address these issues, we propose PiP-MColl, a Process-in-Process-based Multi-object Interprocess MPI Collective design that maximizes small message MPI collective performance at scale. We also present specific designs to boost the performance for larger messages, such that we observe a comprehensive improvement for a series of message sizes beyond small messages. PiP-MColl features efficient multiple sender and receiver collective algorithms and leverages Process-in-Process shared memory techniques to eliminate unnecessary system call, page fault overhead and extra data copy, which results in improved intra- and inter-node message rate and throughput. Experimental results demonstrate that PiP-MColl significantly outperforms popular MPI libraries, including Open-MPI, MVAPICH2, and Intel MPI, by up to 4.6X for the MPI collectives `MPI_Scatter`, `MPI_Allgather`, and `MPI_Allreduce`.**

***Index Terms*—MPI Collective, Message Passing Interface, Process-in-Process, Parallel Algorithms, Distributed Systems**

I. Introduction

MPI collectives have been embraced across a range of research fields [1]–[8] due to their ability to provide low latency and high throughput on distributed-memory systems. As we enter the era of exascale computing, the number of cores per node and nodes being utilized by MPI applications has reached unprecedented levels, leading to greater demands for scalability in MPI collectives. Even small or medium-message collectives can incur significant communication overhead, making it essential to carefully design them for optimal performance [9].

In general, MPI collectives involve both intra- and inter-node communication. While inter-node collective communication is often the primary source of overhead and has been the focus of optimization efforts, the increased number of cores per node means that the overhead associated with intranode collective communication is no longer insignificant [10]. As a result, various methods have been proposed to optimize both intranode and inter-node collective communication in order to achieve maximal communication performance.

Kernel-assisted data copy approaches have been long demonstrated effective to speed-up intra-node communications. In [11], Jin proposed LiMiC, a high-performance and portable kernel module interface integrated with MVAPICH, which is a high-performance open source MPI. In [12], Ma et al. published a Linux kernel module, KNEM, enabling high-performance intra-node MPI communication for large messages. In [13], Chakraborty proposed Cross Memory Attach (CMA) which allows efficient intra-node communication without introducing redundant data copy. These approaches, however, suffer from significant performance degradation for small or medium-message collectives due to the overhead of expensive system calls and page faults. Parsons et al. demonstrated efficient MPI collective algorithms with a POSIX shared-memory (POSIX-SHMEM) [14] multisender design. However, POSIX-SHMEM can limit the efficiency of algorithms that are unable to achieve high performance for medium- and large-message collective communication due to the inherent double copy overhead.

In addition to these data copy methods, researchers have also explored techniques that leverage data sharing based shared-memory (SHMEM) to reduce intra-node data transfer overhead. Hashmi et al. proposed strategies to reduce shared address space and accelerate `MPI_Allreduce` and `MPI_Reduce` communication using the interprocess SHMEM capability of XPMEM [15]. Nevertheless, XPMEM has system call overhead for buffer expose and attachment which limits its performance in small- and medium-messages. Apart from these previous shared-memory approaches, Hori et al. proposed Process-in-Process (PiP), a programming environment that allows all processes on a node to be loaded into the same virtual memory space and enables them to access each other's private memory as if they were threads in userspace [16]. PiP is able to facilitate aforementioned extra copy and expensive system-related overhead. A direct application of PiP in current MPI collectives, however, is unable to fully saturate the network bandwidth and is limited by the potential negative

impact of unnecessary process synchronization on overall performance. To address these limitations, we introduce PiP-MColl, a Process-in-Process-based multi-object interprocess MPI collective design that maximizes intra- and inter-node message rate and network throughput. More specifically, our contributions include:

- We design and implement PiP-MColl, the first-ever multi-object MPI collective that leverages the capabilities of PiP to avoid both extra data copies and expensive system-related overhead, resulting in improved message rate and network throughput.
- We present delicate multi-object designs for popular MPI collective algorithms that enables multiple concurrent data send and receive operations without introducing extra data copy or system-related overhead.
- We explicitly optimize PiP-MColl by overlapping intra-node communication with inter-node communication to better utilize the network bandwidth. This optimization is especially beneficial for medium- and large-message collectives.
- We evaluate the performance of PiP-MColl for three MPI collectives, `MPI_Scatter`, `MPI_Allgather`, and `MPI_Allreduce`. Our results show that PiP-Mcoll outperforms Intel MPI, OpenMPI, MVAPICH2, and the baseline MPICH integrated with PiP over a range of message sizes and execution scales (up to 128 Xeon Broadwell nodes) by up to 4.6X.

The rest of the paper is organized as follows: we introduce background in Section II, and then detail our designs and optimizations in Section III. Evaluation results are given in Section IV. We conclude our paper in Section V.

II. Background and related works

MPI collectives involve both intra- and internode communication. As the number of cores per node has reached tens and even hundreds, the overhead of intranode collective communication has become significant. To optimize the performance of intranode collectives, various shared-memory techniques have been adopted. In this section, we provide an overview of the widely adopted shared-memory techniques in the design of MPI collectives.

A. Shared-memory-based techniques

Traditionally, share-memory-based methods are utilized to reduce intranode communication overheads.

1) POSIX Shared Memory: The POSIX Shared Memory (POSIX-SHMEM) interface is a widely adopted method in the design of Message Passing Interface (MPI) systems. It is integrated into MVAPICH2 for its portability and efficiency [17]. Besides, it is natively supported by the Linux kernel. To exchange data, processes must collectively allocate shared-memory buffers through system calls. The sender process then copies the user data into the shared-memory buffer, while the receiver process copies the data out of the shared-memory buffer into its own receive buffer. This exchange mechanism brings fast communication when the message size is small since process synchronization is not required. However, the double data copy overhead causes lower performance for medium- and large-message communication, compared with other kernel-assisted shared-memory techniques. In contrast, we have resolved the double data copy overhead in our PiP-Mcoll.

2) Data-exchange-based shared memory: LiMiC and KNEM are Linux kernel modules that enable direct data copy between processes. They are integrated into MVAPICH2 and Open MPI respectively [17] [18]. Both of them utilize system calls to operate user-level buffer segments through the kernel. The sender process registers a user-level buffer in the kernel space, obtains a kernel-created buffer key, and sends the key to the receiver process. The receiver process then retrieves the data through a kernel system call. The CMA mechanism, integrated into the Linux kernel, allows for data exchange between processes using the system calls `process_vm_writev` or `process_vm_readv`. CMA is integrated into Open MPI because it provides a simple and native way for interprocess data exchange [18]. Nevertheless, it still incurs system calls for every data transmission. LiMiC, KNEM, and CMA are all designed for *data exchange* rather than *data sharing*. As a result, they are unable to eliminate unnecessary data copies during collective communication. For instance, in the `MPI_Allreduce` operation, each process must exchange data with all other processes before performing the reduction. This data exchange leads to additional data copies that can be avoided through data sharing. For small-message cases, system calls involved in these techniques bring additional overheads into communication, which are avoided in our PiP-Mcoll. Furthermore, unnecessary data copies slow down the collective communication for medium and large messages, while we do not need these data copies in PiP-Mcoll.

3) Data-sharing-based shared memory: XPMEM is a Linux kernel module that enables data sharing among processes in the user space. It is intergrated into MVAPICH2 and Open MPI [17] [18]. Different from LiMiC, KNEM, and CMA, which are designed for data exchange, XPMEM allows for data sharing among processes. The sender process can expose its user buffer, while the receiver process attaches the buffer to its own address space and performs the data exchange. This allows the receiver process to access the private buffer of the sender process without incurring an extra copy operation. This feature brings significant benefits for reduce-based MPI communication. However, similar to LiMiC, KNEM, and CMA, XPMEM requires system calls for buffer expose and attachment. Thus, its performance in small messages is limited. On the contrary, heavy system calls are not needed in our PiP-Mcoll. For larger messages, the data-sharing nature of XPMEM could bring remarkable benefits compared with those data-exchange-based approaches.

B. Process-in-Process-based techniques

Two parallel execution models are broadly adopted by many HPC systems. They are multiprocess and multithread approaches. However, the multiprocess method can not guarantee

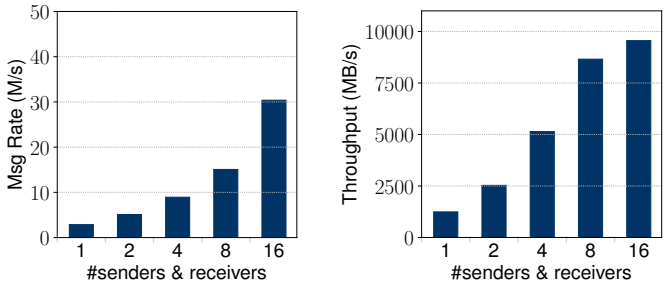
the efficient inter-process MPI communications when there are many processes on a node due to the additional data copy and OS-related overheads. As for the multithread technique, it is effective for inter-core communications. Nevertheless, in the MPI environment, race contention between multiple threads usually brings significant overheads. The third parallel execution model absorbs the strengths of the two models and Process-in-Process (PiP) is a portable implementation that supports this execution model.

PiP is a userspace shared address space technique that does not need system calls in order to achieve interprocess data exchange. In the PiP environment, MPI processes are loaded into the same virtual memory space. Each process has its own separate context (e.g., static variables) but can access the private data of other processes like threads. No system call overhead is involved throughout the communication. Unlike the conventional shared memory techniques, this shared address space technique allows us to access any data structures including pointers. This feature brings the possibility to deliver optimal performance for all collective communication sizes compared with other shared-memory techniques. However, PiP itself can cause huge performance degradation especially in small messages because processes need to synchronize message sizes before any communications. PiP-Mcoll has mitigated this synchronization overhead with carefully designed algorithms. Besides, the network bandwidth or message rate can not be saturated by small and medium messages if we directly integrate PiP into current MPI collectives, while they can be saturated by PiP-Mcoll. For larger messages, the shared-address feature of PiP could bring notable performance boost with the suitable collective algorithm design, which is PiP-MColl.

III. Multiobject Interprocess MPI Collective Design And Optimizations

In this section, we first provide an overview of the key factors that impact message rate and network throughput to motivate our multi-objective (i.e., multiple sender and receiver) design approach. We then delve into the communication cost model used for theoretical performance analysis. Furthermore, we present a detailed examination of the PiP-based multi-objective collective algorithm designs and specific optimizations for a variety of message sizes.

Evaluating the efficiency of MPI communication involves assessing two key metrics: message rate and network throughput. For small-message communication, a higher message rate implies better parallelism, while for medium- or large-message communication, a higher network throughput implies better utilization of network bandwidth. With large-message communication, it is typical for network bandwidth to be saturated by a single process, which highlights the importance of improving internode collective performance through algorithmic means to reduce overall communication volume. However, for small- and medium-message communication, it is unlikely for data transmission to saturate the corresponding hardware, such as memory and network interface card, with a single process alone. We demonstrate the feasibility of our multi-object design by showcasing its performance on the widely-deployed network interconnect Intel Omni-Path [19]. Figure 1 illustrates the performance of point-to-point communication with 4KB and 128 KB message sizes, and varying numbers of senders and receivers on two separate nodes.



(a) Message rate: 4KB message (b) Throughput: 128KB message

Fig. 1: Inter-node message rate and network throughput for point-to-point communication using Intel Omni-Path network interconnect, with varying numbers of senders and receivers and message sizes of 4KB and 128KB.

Figure 1 illustrates the effectiveness of adopting multiple senders and receivers for point-to-point communication, as it results in higher message rate and better network throughput due to improved hardware utilization. These results provide the foundation for our choice to use as many objects as possible in PiP-MColl to achieve optimal collective communication performance.

The Hockney cost model [20] is a simple yet popular model that expresses data transmission performance. It is represented by the equation $\alpha + M * \beta$, where α denotes start-up latency per message, M is the message size in bytes, and β is the transmission speed (s/byte). In the context of MPI collectives that involve both intra- and internode communication, we extend this model to more accurately capture theoretical performance by including both intra- and internode runtime. In this study, we define several symbols to represent system performance characteristics. Specifically, α_r denotes intra-node start-up latency, α_e denotes internode start-up latency, β_r and β_e denote intra-node and internode transmission time per byte, respectively, γ denotes reduction speed (s/byte), M is the message size in bytes, P is the number of processes per node, and N is the number of nodes on which MPI collectives are executed.

A. PiP-MColl optimizations for small messages

An explicit multi-object design, as shown in Figure 1, achieves higher message rate and network throughput compared to previous single-object techniques for internode communication on small messages. This motivates the performance-oriented designs of our PiP-MColl collective algorithms.

1) MPI_Scatter: In `MPI_Scatter`, the global root process designated by the user evenly distributes data to other processes. Conventional MPI often uses a binomial tree algorithm [21] which selects only one pair of sender and receiver per

node for internode data exchange. In contrast, PiP-MColl utilizes all processes on a node as senders and receivers to scatter and receive data among nodes in parallel for all message sizes. Here we define the local rank of a process on a node as R_l ranging from 0 to $P-1$. We also assume that the user-designated global root process is a local root process for convenience. N is the power of $P+1$, C_b is the number of bytes each process should receive into the destination buffer in scatter, and N_{id} is the node id ranging from 0 to $N-1$.

In Figure 2, we present the high-level design of PiP-MColl's `MPI_Scatter` with overlapped intranode scatter. The algorithm proceeds as follows. ❶ The global root node or nodes that just receive the data share the sending buffer address. If the local root process has not shared the source buffer address A_s, it posts the address to all processes on the node and moves on to the next step. ❷ The data is scattered asynchronously through the network. Each process on a node with data finds its paired process with global rank $((R_l+1)*\frac{N}{P+1}+N_{id})*P$ and asynchronously sends data from address $A_s+(R_l+1)*\frac{N}{P+1}*C_b*P$ with $\frac{N}{P+1}*C_b*P$ bytes. ❸ The intranode scatter is performed. For all processes on a node containing data, each process finds offset address $A_s+R_l*C_b$ and copies C_b bytes into the receiving buffer. ❹ Wait until the internode scatter completes. Each process waits until its own internode sending requests issued in step 2 complete and the paired processes wait until receiving all data. ❺ The process recursively executes steps ❶ to ❹. For each round, we update $N=\frac{N}{P+1}$. If $N==1$, the algorithm completes; otherwise, it goes back to ❶.

Our algorithm overlaps intra- and internode scatter, resulting in an overall runtime that is determined by the longer of $T_{intrascatter}$ and $T_{interscatter}$. $T_{intrascatter}$ is defined as $\alpha_r+P*C_b*\beta_r$, and $T_{interscatter}$ is defined as $\alpha_e*\lceil log_{P+1}N\rceil+C_b*(N-1)*P*\beta_e$. As message size C_b increases, the total running time T also increases linearly. Similarly, as the number of nodes N increases, $T_{interscatter}$ becomes more significant and T increases linearly as well. This demonstrates that our algorithm has good scalability with respect to C_b and N. Additionally, the explicit overlap improves bandwidth utilization for medium and large messages, allowing for consistent use of the `MPI_Scatter` algorithm across different message sizes.

2) MPI_Allgather: We further extend our algorithmic designs to the another MPI collective algorithm `MPI_Allgather`. Traditionally, for small messages, the Bruck algorithm [22] is used for non-power-of-two cases and the recursive doubling algorithm [23] for power-of-two cases. To achieve high-performance in our allgather routine, we first design PiP-MColl allgather algorithm for small message sizes. Figure 3 illustrates a high-level overview of the PiP-MColl allgather algorithm for small message cases, which can be described as follows: ❶ We begin by performing intranode gather to the local root process. Local processes perform `MPI_Gather` to gather data into the local root process destination buffer A_d. ❷ Next, we initialize parameters. The multi-object Bruck algorithm step is initialized as $S_p=1$ and the base of the multi-object Bruck algorithm as $B_k=P+1$. ❸ We find the paired source and destination process. Each process sets $N_{offset}=(R_l+1)*S_p$ and finds the paired source node $N_{src}=(N_{id}+N_{offset})\%N$ and destination node $N_{dst}=(N_{id}-N_{offset})\%N$. The paired source process rank is $N_{src}*N+R_l$, and the destination process rank is $N_{dst}*N+R_l$. ❹ We then perform send and receive. We define C_b as the number of bytes received from each process in allgather and A_d as the starting address of the destination buffer of the local root process. Each process sends C_b*S_p bytes from the local root process buffer to the destination process and receives C_b*S_p bytes from the source process into address $A_d+C_b*S_p*(R_l+1)$. For each process, we update $S_p=S_p*B_k$. If S_p is less than or equal to $\frac{N}{B_k}$, we repeat steps ❸ to ❹. If not, we proceed to step ❺. ❺ If N is not a power of B_k, we have remaining $N-S_p$ nodes for the final step. Each process takes $Rem=Max(Min(S_p,N-S_p*R_l),0)$ remainder. If $Rem>0$, the process will send and receive $Rem*C_b$ bytes from the paired destination and source process. ❻ Finally, the local root process shifts data into the correct sequence [22] and broadcasts to other processes.

According to the small-message allgather algorithm mentioned above, we have equation $T_{intra-gathers}=\alpha_r+(1+N*P)*(P-1)*C_b*\beta_r$, which shows the intranode gather runtime, and equation $T_{inter-allgathers}=\alpha_e*\lceil log_{P+1}N\rceil+(C_b*P-1)*C_b*P*\beta_e$, which shows the internode allgather runtime. Since we do not overlap intra- and internode communication in this case, the overall runtime is $T=T_{intra-gathers}+T_{inter-allgathers}$. Referring to the equations, we can see that as the number of nodes, N, increases, the runtime of $T_{intra-gathers}$ grows linearly and $T_{inter-allgathers}$ even increases less significantly, which demonstrates the scalability of our algorithm in terms of N. However, as the message size, C_b, increases, $T_{inter-allgathers}$ has a quadratic growth. Therefore, this algorithm is not well-suited for larger messages as it prioritizes reducing latency over minimizing the total data transferred. To improve allgather performance for medium and large messages, a new algorithm must be developed.

3) MPI_Allreduce: To optimize performance, we designed the recursive PiP-MColl Bruck algorithm for small-message allreduce. The design in allreduce is similar to allgather, but it requires an additional reduction operation after each data transmission. In addition, when the number of nodes is not a power of $P+1$, we need to compute the reduction results of the reminder recursively. ❶ All processes on a node perform intranode binomial reduce and store final results into destination buffer A_d of the local root process. ❷ We set S_p to 1 and B_k as $P+1$ as the base of the multiobject Bruck algorithm. ❸ Each process assigns $N_{offset}=(R_l+1)*S_p$ and finds the paired source node $N_{src}=(N_{id}+N_{offset})\%N$ and destination node $N_{dst}=(N_{id}-N_{offset})\%N$. The paired source process rank is $N_{src}*N+R_l$, and the destination process rank is $N_{dst}*N+R_l$. ❹ Each process sends C_b bytes from the local root process buffer A_d to the destination process, receives

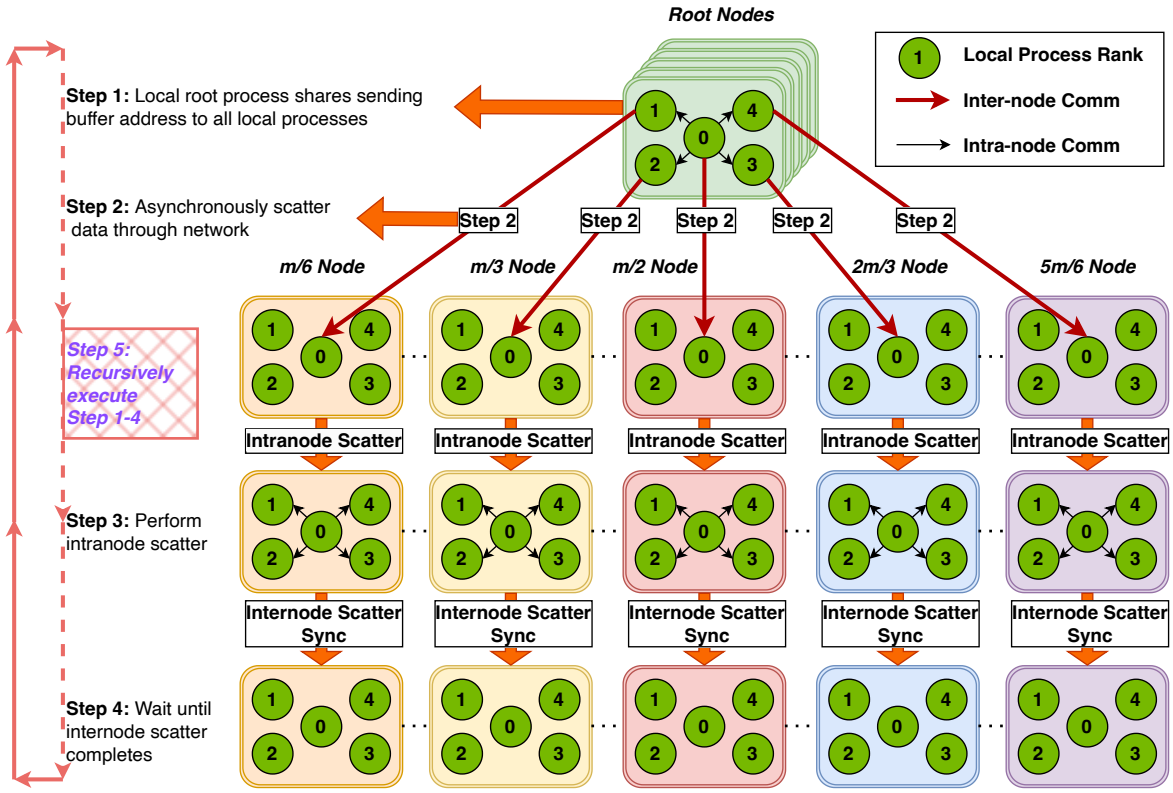


Fig. 2: High-level design of PiP-MColl `MPI_Scatter` with overlapped intranode scatter. Shown as an example are 5 senders and 1 receiver on a node.

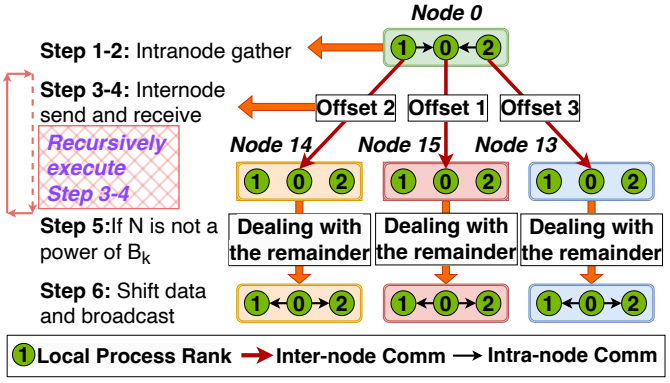


Fig. 3: Example of PiP-MColl allgather algorithm for small-message communication with 16 nodes and 3 objects per node.

C_b bytes from the source process in a temporary buffer and performs an intranode reduce. ❺ We handle the current stage remainder. Every process sets $S_p = S_p * B_k$. If $Rem = N\%S_p$ is not zero, we need to perform an intranode reduction for the reminder. If $S_p == B_k$, then Rem number of processes perform intranode reduction using the received data and store the results in a new remainder buffer A_r. If $S_p > B_k$, then $\lceil \frac{Rem * B_k}{S_p} \rceil$ number of processes perform intranode reduction using the received data and previous reminder results and store the results in a new reminder buffer A_r. If S_p is smaller than or equal to $\frac{N}{B_k}$, go back to ❸; otherwise, go to ❻. ❻ We then start the final stage for remainder. If N is not a power of B_k, each process sets $Rem = Min((N - S_p) - R_l * S_p, S_p)$. If $Rem > 0$ and $Rem == S_p$, the process sends C_b bytes from A_s to the destination process and receives C_b bytes from the source process. On the other hand, if $Re > 0$ and $Re \neq S_p$, the process sends C_b bytes from A_r to the destination process and receives C_b bytes from the source process. The processes with $Re > 0$ on the node perform an intranode reduce. ❼ Eventually, we broadcast results. The local root process broadcasts the global reduction results to all processes on the node, and the algorithm completes.

The overall runtime of the `All_reduce` function for small-message inputs can be calculated as the sum of $T_{intra-reduces}$ and $T_{inter-allreduces}$. $T_{intra-reduces}$ can be expressed as $\alpha_r * \lceil log_2 P \rceil + C_b * \lceil log_2 P \rceil * \beta_r + C_b * \lceil log_2 P \rceil * \gamma$, while $T_{inter-allreduces}$ is $\alpha_e * \lceil log_{P+1} N \rceil + C_b * P * \lceil log_{P+1} N \rceil * \beta_e + C_b * \lceil log_{P+1} N \rceil * \gamma$. From these calculations, we can see that an increase in message size C_b will linearly increase the total runtime, while the node size N has a logarithmic relationship with $T_{inter-allreduces}$. This indicates that we have good scalability in N. However, the overall internode communication volume is $C_b * P * \lceil log_{P+1} N \rceil$, which does not scale well when we have a large N, C_b, and P. Similar to allgather, this is because we focused on minimizing latency when designing the algorithm. Therefore, there is still room to reduce the total

transferred data in order to improve performance for medium and large messages.

B. PiP-MColl based Optimization on Larger Messages

As we conclude in the last section, our MPI_Scatter algorithm is already highly scalable in message size C_b, but our current small-message algorithms for MPI_Allgather and MPI_Allreduce are not able to deliver high performance in terms of larger messages. Accordingly, we still need to specifically optimize these two routines for achieving a smaller running time for medium- and large-message cases.

1) MPI_Allgather: In medium- and large-message allgather, we adopt the multiobject ring algorithm to maximize network bandwidth utilization. The algorithm, as shown in Figure 4, proceeds as follows: ❶ Intranode gather is applied to the local root process, where all processes on a node gather data into the destination buffer A_d of the local root process. ❷ Paired processes are identified and parameters are initialized, with each process finding the left source node $N_{src} = (N_{id} - 1)\%N$ and right destination node $N_{dst} = (N_{id} + 1)\%N$. The paired source process rank is thus $P * N_{src} + R_l$, while the destination process rank is $P * N_{dst} + R_l$. After that, each process sets $N_{doffset} = N_{id}$ and $N_{soffset} = N_{src}$ for node reference. Thereafter, address offsets $A_{doffset} = C_b * R_l + N_{doffset} * P * C_b$ and $A_{soffset} = C_b * R_l + N_{soffset} * P * C_b$ are calculated. ❸ Multiobject send and receive in a ring pattern is conducted, with each process asynchronously sending C_b bytes starting from $A_d + A_{doffset}$ to the destination process and receiving C_b bytes in $A_d + A_{soffset}$. ❹ Overlapped intranode broadcast is performed, with the local root process broadcasting $P * C_b$ bytes at address $A_d + A_{doffset}$ to other processes on the node. ❺ Parameters are checked and updated, with the process setting $N_{doffset} = N_{soffset}$, $N_{soffset} = (N_{soffset} - 1)\%N$ and repeating from step 3 if the ring communication step is smaller than $N - 1$, otherwise the algorithm completes.

The overall runtime of medium- and large-message allgather can be calculated by adding the intranode gather (shown in equation $T_{intra-gatherl} = \alpha_r + (P-1) * C_b * \beta_r$) to the larger one of the overlapped intranode broadcast runtime and internode multiobject ring communication runtime , as shown in equations $T_{intra-bcastl} = \alpha_r * (N-1) + (P-1) * N * P * C_b * \beta_r$ and $T_{inter-allgatherl} = \alpha_e * (N-1) + P * C_b * (N-1) * \beta_e$. We are able to observe that the total execution time T is now in a linear relationship with message size C_b. Compared with the quadratic relationship in the previous algorithm, a tremendous time has been saved for medium or large messages, which ensures high scalability with regard to C_b. This is due to the fact that we try to minimize the size of transferred data when we design this new algorithm. To get even better performance, we apply overlapping between $T_{intra-bcastl}$ and $T_{inter-allgatherl}$. As shown in the equations above, the internode allgather and intranode broadcast time scale linearly with the number of nodes N, while the intranode gather runtime does not change with N. We thus maintain the linear property considering the rise of node size N.

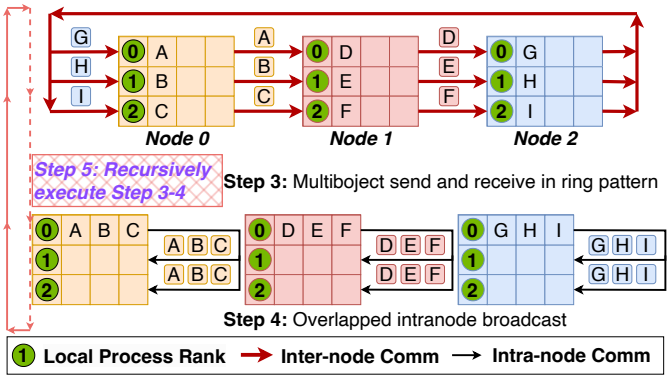


Fig. 4: High-level PiP-MColl allgather algorithm for medium- and large-message size. The figure shows an example with 3 nodes and 3 objects per node and one step of the ring communication. The intra- and internode communications run in parallel for overlapping.

2) MPI_Allreduce: In order to achieve high performance for medium- and large-message allreduce operations, our proposed PiP-MColl algorithm utilizes a multi-object reduce-scatter approach, followed by a PiP-MColl based allgather. In contrast to the traditional Rabenseifner's algorithm [24], where a reduce-scatter followed by an allgather are performed, we assume that N is divisible by P, C_b is divisible by N, and the data chunk size is S_c. The algorithm is described as follows: ❶ We begin by performing an intranode reduce operation, where all processes on a node perform the intranode reduce (the detailed algorithm is explained in Section III-C), and store the final results in the local root process destination buffer. ❷ Next, we post the buffer address. Each process posts the source data buffer address to all other processes on the node and gets the local root process destination buffer address A_d. ❸ Afterwards, we find paired node range and process. Each process finds the paired node range from $\frac{N*R_l}{P}$ to $\frac{N*(R_l+1)}{P}$. For a paired node N_p, the paired destination process rank is $N_p * P + R_l$. ❹ Subsequently, we perform internode reduce-scatter. For each paired node, a process finds the data chunk starting from $A_d + \frac{C_b*N_p}{N}$ to $A_d + \frac{C_b*(N_p+1)}{N}$ and sends it to the paired process. Then, if $N_p == N_{id}$, the process receives $N - 1$ chunks from the paired source processes and reduce in the corresponding chunk; otherwise, it sends the chunk to the paired destination process. ❺ Finally, we conduct internode allgather with intranode broadcast. After step ❹, each node owns the partial results of allreduce. All processes need to perform internode allgather followed by intranode broadcast to obtain the complete global results, and the algorithm completes.

Our algorithm for medium- and large-message allreduce includes intranode reduce, internode reduce-scatter, internode allgather, and intranode broadcast. The runtime for intranode reduce and internode reduce-scatter can be found in equations $T_{intra-reducel} = \alpha_r * (P-1) + C_b * P * \gamma$ and $T_{inter-rscatterl} = \alpha_e * (P-1) + \frac{(N-1)*C_b}{N} * \beta_e + \frac{C_b}{N} * (N-1) * \gamma$. As previously discussed in Section III-A2, the runtime for internode allgather

and intranode broadcast have been analyzed. The overall runtime for the algorithm is calculated as $T = T_{intra-reducel} + T_{inter-rscatterl} + Max(T_{intra-bcastl}, T_{inter-allgatherl})$. By analyzing the equations, we can see that our medium- and large-message allreduce algorithm has improved scalability. By reducing the internode transferred data from $C_b * P * \lceil log_{P+1} N \rceil$ to $\frac{C_b}{N} * (N-1)$, we have significantly decreased the running time T with larger message sizes C_b and node numbers N. Additionally, we have implemented overlapping to mitigate network congestion, resulting in improved performance for medium- and large-message cases compared to our small-message algorithm.

C. Auxiliary MPI Collectives

The primary MPI collectives that we have focused on in this work are `MPI_Scatter`, `MPI_Allgather`, and `MPI_Allreduce`. However, in order to achieve optimal performance for these collectives, we also designed several auxiliary intranode MPI collectives, including `MPI_Bcast`, `MPI_Gather`, and `MPI_Reduce`. These building blocks play a crucial role in our PiP environment and are essential for the efficient execution of the primary MPI collectives.

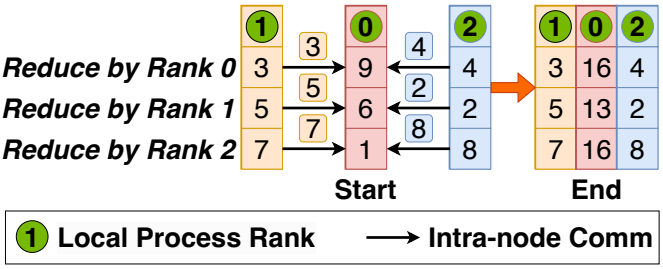


Fig. 5: PiP-MColl–based large-message intranode reduce communication with 3 processes on a node. Each buffer is chunked based on the number of processes, and all data will be reduced into the root process destination buffer.

The MPI collective `MPI_Bcast` is a one-to-all operation where one root process broadcasts the data to all processes in the group. For small-message broadcasts, the root process first copies the source data into a temporary buffer and then posts the address of the temporary buffer to all processes. These processes then copy the data into their destination buffer. For large-message broadcasts, the root process posts its source buffer address to all processes in the beginning. The processes then copy the data into their destination buffer. In this case, the root process needs to wait until all processes have completed the data copy. `MPI_Gather` is an operation where all processes send their data to a designated root process. The same algorithm is applied for both small and large message communication. The root process first posts its destination buffer address to all processes. Then, each process copies its source data into the designated position in the root's buffer. In this case, the root process needs to wait until all processes have completed the data copy. For `MPI_Reduce`, the binomial algorithm is used for small message reduction. For large messages, the root process posts its destination buffer, while every other process posts its source data buffer. Each process is then responsible for reducing a specific chunk of the buffer. If there are N processes, each buffer is evenly divided into N chunks, and process i will reduce the ith chunk from all source data buffers into the ith chunk of the destination buffer. This is illustrated in Figure 5. The root process must wait for all other processes to complete their data reduction.

IV. Experimental Results

In this section, we first compare the scalability and effectiveness of PiP-MColl with our baseline (PiP-MPICH) for both small and medium message sizes. We then evaluate the large-scale microbenchmark performance of PiP-MColl against PiP-MPICH, as well as ubiquitous libraries such as Intel MPI, MVAPICH2, and OpenMPI.

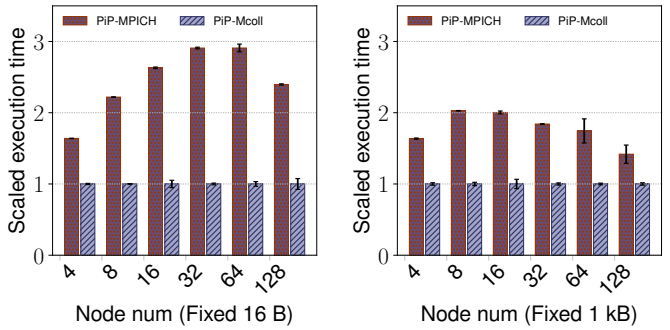
A. Experimental Setup

In our experiments, we use a 128-node cluster with 18 processes per node, resulting in a total of 2304 processes. Each node is equipped with two Intel Xeon E5-2695v4 Broadwell processors, yielding a total of 36 cores per node. Additionally, each NUMA node is equipped with 64 GB of DDR4 memory, resulting in a total of 128 GB of memory per node. The nodes are connected through the Intel OPA interconnect, with a maximum message rate of 97 million per second and a bandwidth of 100 Gbps. Hyperthreading is disabled on all nodes.

We use PiP-based MPICH, extended from the commit bb595ca0 of the MPICH main branch, as the baseline implementation. It includes single-copy-based optimization for medium and large intranode message communication [16]. All source codes are compiled using gcc/gfortran version 4.8.5. The low-level libraries used include Libfabric version 1.10.1, Psm2 version 0201, and Glibc version 2.17. The Glibc libraries are patched to support PiP task spawn. For comparison, we use Intel-MPI version 2017.3, OpenMPI version 4.1.2, and MVAPICH2 version 2.3.6 to compare with PiP-MColl. In our microbenchmark experiments, we use a two-stage approach: a warm-up stage and an execution stage. Both stages run the same number of iterations. For small message sizes (ranging from 16 bytes to 1 kilobyte), we run 10,000 iterations and calculate the average time per iteration as the final runtime. For medium message sizes (ranging from 1 kilobyte to 128 kilobytes), we run 1,000 iterations for sizes between 1 and 8 kilobytes, and 100 iterations for sizes between 8 and 128 kilobytes. For large message sizes (128 kilobytes or larger), we run 10 iterations. To ensure accuracy, we repeat all microbenchmarks 10 times and measure their standard deviation.

B. Scalability Evaluation

1) MPI_Scatter: Figure 6 presents the performance of `MPI_Scatter` for both fixed small message sizes (16 B) and medium message sizes (1 kB) as the number of nodes increases. As shown in the figure, PiP-MColl outperforms the baseline for both small and medium message communication. This is due to the multi-object design of PiP-MColl, which is able to fully utilize network bandwidth even for small



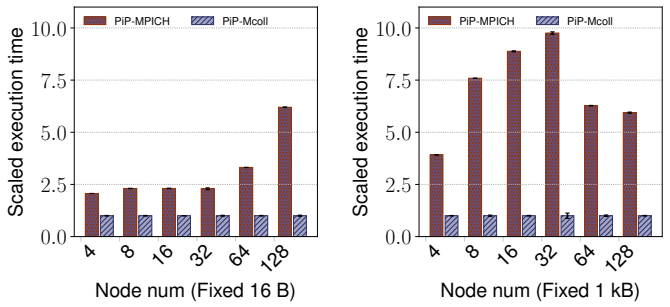
(a) Node-based test with 16 B message size.

(b) Node-based test with 1 kB message size.

Fig. 6: MPI_Scatter performance with increasing numbers of nodes.

and medium messages. Additionally, it is evident that the speedup for small messages is generally larger than that for medium messages. This is because PiP requires message size synchronization before communication between processes can occur, resulting in a higher performance degradation for small messages compared to medium ones. However, the PiP-MColl algorithm design minimizes unnecessary process synchronization overhead, resulting in less performance degradation compared to PiP-MPICH.

2) MPI_Allgather: Figure 7 presents the performance of `MPI_Allgather` with various numbers of nodes at fixed small message size (16 B) and medium message size (1 kB) per process. PiP-MColl outperforms the baseline (PiP-MPICH) in all cases. For small message (16 B), more than 6 times of performance improvement can be realized using PiP-MColl at 128 nodes (2304 processes). Furthermore, we do not witness the expected decreased speed-ups in the medium message (1 kB). This is because the slow down of the PiP-MPICH itself in message sizes between 64 B and 1 kB as shown in Figure 10 and 13. The performance degradation here emphasizes the need to design new algorithms instead of directly using exiting algorithms for achieving a higher speed MPI collectives with PiP.

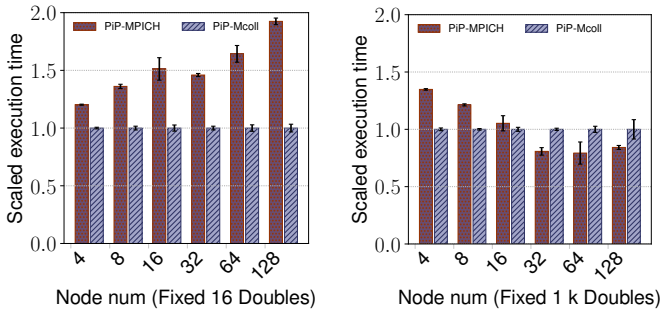


(a) Node-based test with 16 B message size.

(b) Node-based test with 1 kB message size.

Fig. 7: MPI_Allgather performance with increasing numbers of nodes.

3) MPI_Allreduce: In evaluating the scalability of our implementation, Figure 8 benchmarks the performance of `MPI_Allreduce` with fixed small message count (16) and medium message count (1k) as the number of nodes increases. As shown, PiP-MColl outperforms the baseline (PiP-MPICH) in small-message cases due to the benefits of its multi-object design, which leads to higher message rates. However, for medium-message sizes, the benefits of PiP-MColl decrease as the number of nodes increases. This is because the inherent process synchronization overheads in PiP become less prominent with larger data sizes, while our PiP-Mcoll experiences some performance degradation. This is because the multi-object design in `MPI_Allreduce` requires synchronization among processes per node, and the synchronization overhead is comparable to the communication runtime when the message size is not large enough. If the number of nodes is larger than $P+1$ (in our case, $P+1$ is equal to 19), this overhead persists in each repeated step, thus limiting overall speedup. It is important to note that this multi-object synchronization overhead is distinct from the inherent process synchronization overhead in PiP, and as such, our baseline does not experience such slow down in medium message counts (i.e., 1k).



(a) Node-based test with 16 doubles.

(b) Node-based test with 1 k doubles.

Fig. 8: MPI_Allreduce performance with increasing numbers of nodes.

C. PiP-Mcoll for Small Messages

1) MPI_Scatter: We then evaluated the performance of the `MPI_Scatter` function for small message sizes. Figure 9 illustrates the scatter performance for small message sizes per process (i.e., overall data size on the root process is $M_{size} * \#process$) on a 128-node cluster with 18 processes on each node. The execution time is normalized based on the time spent by PiP-Mcoll, with lower bars indicating higher performance. To better distinguish performance differences between the MPI implementations, we have excluded execution times that are more than 4 times larger than that of PiP-Mcoll. As shown in the figure, PiP-MColl consistently outperforms the other MPI implementations, achieving the best speedup of 65% when the message size is 256 bytes. This demonstrates the effectiveness of our multiobject design for small message sizes, as it maximizes the message rate and results in higher performance than other MPI implementations.

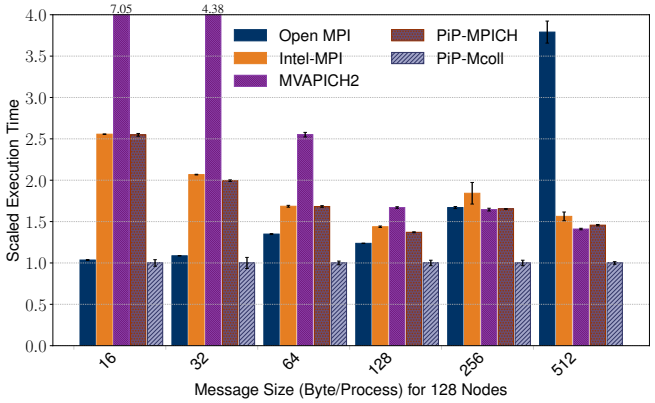


Fig. 9: MPI_Scatter performance with small message sizes.

2) MPI_Allgather: On the other hand, Figure 10 shows the `MPI_Allgather` performance with small message sizes from 16 B to 512 B on 128 Xeon Broadwell nodes. Theoretically, `MPI_Allgather` generates more data movements compared to other two MPI collectives, as processes receive more data than they send, which benefits the most from PiP-MColl. From the experimental data, we find that PiP-MColl outperforms other MPI implementations in all cases. Similarly, our multi-object design brings a noticeable performance improvement for small messages (i.e., 64 B), where PiP-Mcoll is over 4.6X as fast as the fastest MPI implementation. We also observe that our baseline (PiP-MPICH) sometimes has the worst performance among all the MPI implementations. This is due to the synchronization overhead inside PiP, which requires message size synchronization before communications. As `MPI_Allgather` is more communication-intensive compared to `MPI_Scatter`, the overhead here is more serious than in `MPI_Scatter`. As a result, it is important to redesign MPI collective algorithms for similar cases.

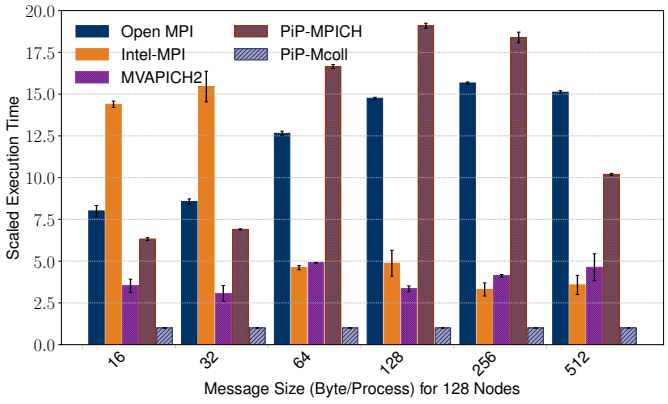


Fig. 10: MPI_Allgather performance with small message sizes.

3) MPI_Allreduce: Figure 11 presents the performance of `MPI_Allreduce` with small message counts on 128 nodes. Our multi-object design in PiP-MColl allows for maximum message rate, resulting in the best performance among all MPI implementations. According to the experimental data, PiP-MColl shows up to 31% speedup compared to the fastest MPI implementation with small message counts (i.e., 256 B). This high performance of PiP-MColl's `MPI_Allreduce` further demonstrates the effectiveness of our multi-object design.

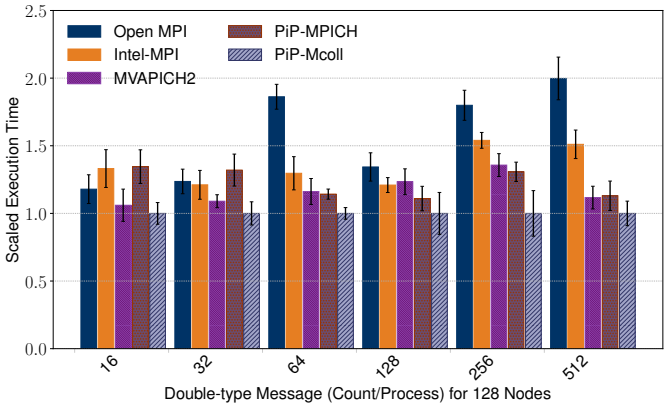


Fig. 11: MPI_Allreduce performance with small double-type message counts.

D. PiP-Mcoll for Larger Messages

1) MPI_Scatter: The performance of `MPI_Scatter` with medium and large message sizes is shown in Figure 12. To enhance clarity, the scaled execution time is capped at twice the time of PiP-Mcoll. In the analysis in Section III-A1, we found that the PiP-Mcoll Scatter algorithm works well for both small and larger message sizes. Therefore, we use the same `MPI_Scatter` algorithm for different message sizes. As shown in the results, PiP-MColl outperforms other MPI implementations in all cases. Compared to the fastest implementation among all other MPI libraries, PiP-MColl achieves the highest speedup (32%) when the message size is small (i.e., 1 kB), and the speedup gradually decreases as the message size increases until 64 kB. This is because as the message size increases, the network bandwidth becomes saturated, resulting in less significant benefits from the multi-object scatter. For large messages, although PiP-MColl does not bring about significant speedup, it still provides some benefits from better network utilization through overlapping inter- and intranode communications. For instance, we can improve the performance of `MPI_Scatter` by 17% compared to the fastest MPI implementation for a 512 kB large message. We also notice an abnormal speedup increment compared to PiP-MPICH when the message size is 512 kB. This is due to performance degradation of the baseline rather than PiP-MColl. In theory, the baseline should be slightly slower than PiP-MColl in these cases.

2) MPI_Allgather: On the other hand, Figure 13 shows the performance of the `MPI_Allgather` function with different message sizes from 1 kB to 512 kB on 128 Xeon Broadwell nodes. Similarly to `MPI_Scatter`, we have cut off the scaled execution time when it exceeds 6 times that of PiP-Mcoll to clearly illustrate the performance differences. To present the

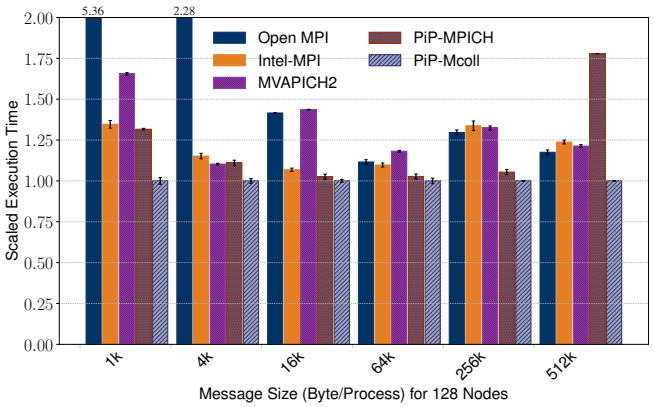


Fig. 12: MPI_Scatter performance with medium and large message sizes. We use the same algorithm as for small message sizes.

benefits of our specific optimizations for larger messages, we add PiP-Mcoll-small into the figure, which adopts the small-message algorithm across the message sizes. After PiP-Mcoll switches to the explicitly designed large-message algorithm, we observe a 146% performance better than PiP-Mcoll-small at 256 kB. As depicted in the figure, PiP-MColl outperforms the baseline (PiP-MPICH) in all cases and generally outperforms other MPI implementations. For medium-size messages (e.g., 4 kB), the benefits of the multiobject design are negligible and the major performance improvement is from intranode communication, which results in an improvement of up to 32% compared to the fastest MPI library. For large messages, PiP-Mcoll has better network utilization due to the overlapped intranode broadcast in the large-message allgather algorithm ❹. As a result, more than 10% speed-up can be achieved in these cases.

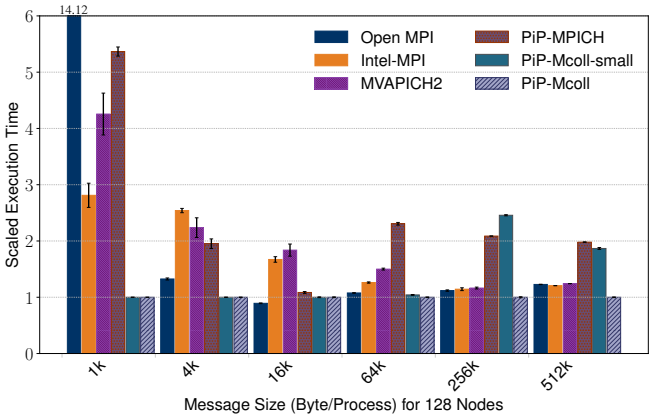


Fig. 13: MPI_Allgather performance with medium and large message sizes. PiP-MColl allgather uses small-message algorithm when message sizes are smaller than 64 kB and the large-message algorithm is switched at 64 kB.

3) MPI_Allreduce: Figure 14 presents the performance with larger message counts on 128 nodes. Similar to the allgather case, we have implemented specific optimizations for larger messages at the algorithm level and the large-message algorithm in PiP-MColl's `MPI_Allreduce` is switched on at 8k message counts (i.e., 64 kB). When the message count is greater than 8k, we observe a 91% speedup on average over the PiP-Mcoll-small variant at 16k. As shown in the figure, the performance of PiP-MColl falls behind other MPI implementations when message counts are between 1k and 16k. This is due to the fact that the small-message algorithm is still being used when message counts are below 8k. As message sizes increase, the benefits of the multiobject design become less significant and cannot offset the synchronization overhead. Additionally, the results differ from those of `MPI_Allgather` because the message size does not increase along with communication, resulting in a smaller ratio of communication to synchronization overhead. As a result, PiP-MColl `MPI_Allreduce` is not able to provide the same level of performance boost as PiP-MColl `MPI_Allgather` there.

On the other hand, when PiP-MColl switches to the large-message algorithm, it is not able to achieve better performance with message counts of 16k. This is because the message sizes are still relatively small, and the benefits of PiP-MColl's large-message algorithm cannot offset the synchronization overhead. However, when the message count is larger than or equal to 64k, PiP-MColl performs better than the baseline due to better network utilization provided by the large-message algorithm. At even larger message counts (i.e., 512k), PiP-Mcoll provides up to 25% performance improvement compared to the fastest MPI implementation.

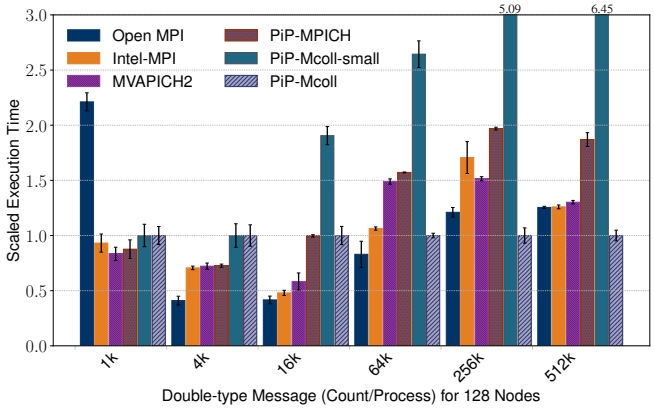


Fig. 14: MPI_Allreduce performance with medium and large double-type message counts. To have a higher performance, MPI_Allreduce switches to the large-message algorithm when message count is larger than or equal to 8k.

V. Conclusion

MPI collective performance is a popular research topic that has been studied for years. The state-of-the-art works adopt various methods to improve intra- and internode collective communication performance. For example, XPMEM, CMA, KNEM, and POSIX shared-memory techniques are widely used for efficient MPI collective design. However, these de-

signs involve heavy system overhead or double copy overhead, which results in suboptimal performance of collective algorithms. In this paper we propose PiP-MColl, a PiP-based multiobject interprocess MPI collective design, to improve performance for small- and medium-message collectives. PiP-MColl utilizes a PiP shared-memory technique to load MPI processes into the same virtual memory space and allows us to perform data copy at the userspace and avoid system and double copy overhead. Multiobject design at large scale for small- and medium-message communication in the PiP environment enables us to obtain maximal message rate and network throughput in the majority of instances. We apply PiP-MColl to three widely used MPI collective routines: `MPI_Scatter`, `MPI_Allgather`, and `MPI_Allreduce`. The experimental results show that PiP-MColl performs much better than the baseline PiP-MPICH and also beats the widely used MPI libraries Intel MPI, OpenMPI, and MVAPICH2 in most cases. In summary, PiP-MColl is an efficient method that utilizes multiobject and the PiP technique and is able to maximize performance at a large scale for small- and medium-message MPI collectives mostly.

ACKNOWLEDGMENT

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, and the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357. We gratefully acknowledge the computing resources provided on Bebop, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

REFERENCES

[1] A. A. Awan, K. Hamidouche, J. M. Hashmi, and D. K. Panda, "S-caffe: Co-designing mpi runtimes and caffe for scalable deep learning on modern gpu clusters," in Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2017, pp. 193–205.

[2] Y. Wang and M. Borland, "Pelegant: A parallel accelerator simulation code for electron generation and tracking," in AIP Conference Proceedings, vol. 877, no. 1. American Institute of Physics, 2006, pp. 241–247.

[3] J. Huang, S. Di, X. Yu, Y. Zhai, J. Liu, Y. Huang, K. Raffenetti, H. Zhou, K. Zhao, Z. Chen, F. Cappello, Y. Guo, and R. Thakur, "gzccl: Compression-accelerated collective communication framework for gpu clusters," 2023.

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard et al., "Tensorflow: A system for large-scale machine learning," in 12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16), 2016, pp. 265–283.

[5] A. Ayala, S. Tomov, X. Luo, H. Shaeik, A. Haidar, G. Bosilca, and J. Dongarra, "Impacts of multi-gpu mpi collective communications on large fft computation," in 2019 IEEE/ACM Workshop on Exascale MPI (ExaMPI). IEEE, 2019, pp. 12–18.

[6] J. Huang, K. Ouyang, Y. Zhai, J. Liu, M. Si, K. Raffenetti, H. Zhou, A. Hori, Z. Chen, Y. Guo, and R. Thakur, "Accelerating mpi collectives with process-in-process-based multi-object techniques," in Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing, ser. HPDC '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 333–334. [Online]. Available: https://doi.org/10.1145/3588195.3595955

[7] A. Jain, A. A. Awan, H. Subramoni, and D. K. Panda, "Scaling tensorflow, pytorch, and mxnet using mvapich2 for high-performance deep learning on frontera," in 2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS). IEEE, 2019, pp. 76–83.

[8] J. Huang, S. Di, X. Yu, Y. Zhai, J. Liu, K. Raffenetti, H. Zhou, K. Zhao, Z. Chen, F. Cappello, Y. Guo, and R. Thakur, "C-coll: Introducing error-bounded lossy compression into mpi collectives," 2023.

[9] S. Chunduri, S. Parker, P. Balaji, K. Harms, and K. Kumaran, "Characterization of mpi usage on a production supercomputer," in SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2018, pp. 386–400.

[10] S. Jain, R. Kaleem, M. G. Balmana, A. Langer, D. Durnov, A. Sannikov, and M. Garzaran, "Framework for scalable intra-node collective operations using shared memory," in SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2018, pp. 374–385.

[11] H.-W. Jin, S. Sur, L. Chai, and D. K. Panda, "Limic: Support for high-performance mpi intra-node communication on linux cluster," in 2005 International Conference on Parallel Processing (ICPP'05). IEEE, 2005, pp. 184–191.

[12] T. Ma, G. Bosilca, A. Bouteiller, B. Goglin, J. M. Squyres, and J. J. Dongarra, "Kernel assisted collective intra-node mpi communication among multi-core and many-core cpus," in 2011 International Conference on Parallel Processing. IEEE, 2011, pp. 532–541.

[13] S. Chakraborty, H. Subramoni, and D. K. Panda, "Contention-aware kernel-assisted mpi collectives for multi/many-core systems," in 2017 IEEE International Conference on Cluster Computing (CLUSTER). IEEE, 2017, pp. 13–24.

[14] B. S. Parsons and V. S. Pai, "Accelerating mpi collective communications through hierarchical algorithms without sacrificing inter-node communication flexibility," in 2014 IEEE 28th International Parallel and Distributed Processing Symposium. IEEE, 2014, pp. 208–218.

[15] J. M. Hashmi, S. Chakraborty, M. Bayatpour, H. Subramoni, and D. K. Panda, "Designing efficient shared address space reduction collectives for multi-/many-cores," in 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2018, pp. 1020–1029.

[16] A. Hori, M. Si, B. Gerofi, M. Takagi, J. Dayal, P. Balaji, and Y. Ishikawa, "Process-in-Process: Techniques for Practical Address-Space Sharing," in Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing. ACM, 2018, pp. 131–143.

[17] M. TEAM. (2020) MVAPICH2-X 2.3 User Guide. [Online]. Available: https://mvapich.cse.ohio-state.edu/static/media/mvapich/mvapich2-x-userguide.pdf

[18] T. O. M. Community. (2023) Documentation for Open MPI. [Online]. Available: https://docs.open-mpi.org/en/v5.0.x/release-notes/networks.html

[19] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak, "Intel® omni-path architecture: Enabling scalable, high performance fabrics," in 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects. IEEE, 2015, pp. 1–9.

[20] R. W. Hockney, "The communication challenge for mpp: Intel paragon and meiko cs-2," Parallel computing, vol. 20, no. 3, pp. 389–398, 1994.

[21] M. Shroff and R. A. Van De Geijn, "Collmark: Mpi collective communication benchmark," in International Conference on Supercomputing. Citeseer, 2000, p. 10.

[22] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby, "Efficient algorithms for all-to-all communications in multiport message-passing systems," IEEE Transactions on parallel and distributed systems, vol. 8, no. 11, pp. 1143–1156, 1997.

[23] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," The International Journal of High Performance Computing Applications, vol. 19, no. 1, pp. 49–66, 2005.

[24] R. Rabenseifner, "Optimization of collective reduction operations," in International Conference on Computational Science. Springer, 2004, pp. 1–9.