Parameterized Algorithms for Non-uniform All-to-all

Ke Fan[1], Jens Domke[2], Seydou Ba[2], and Sidharth Kumar[1]
[1]University of Illinois Chicago, Chicago, IL, USA
[2]RIKEN Center for Computational Science, Kobe, Japan
{kfan23,sidharth}@uic.edu,{jens.domke,seydou.ba}@riken.jp

ABSTRACT

`MPI_Alltoallv` generalizes the uniform all-to-all communication (`MPI_Alltoall`) by enabling the exchange of data-blocks of varied sizes among processes. This function plays a crucial role in facilitating many computational tasks, such as FFT calculations and graph mining operations. Popular MPI libraries, such as MPICH and OpenMPI, implement `MPI_Alltoall` using a combination of linear and logarithmic algorithms. However, `MPI_Alltoallv` typically relies only on variations of linear algorithms, missing the benefits of logarithmic approaches. Furthermore, current algorithms also overlook the intricacies of modern HPC system architectures, such as the significant performance gap between intra-node (local) and inter-node (global) communication. To address these problems, this paper presents two novel algorithms: *Par*ameterized *Log*arithmic non-uniform *A*ll-to-all (*ParLogNa*) and *Par*ameterized *Lin*ear non-uniform *A*ll-to-all (*ParLinNa*). *ParLogNa* is a tunable logarithmic time algorithm for non-uniform all-to-all, and *ParLinNa* is a hierarchical and tunable near-linear-time algorithm for non-uniform all-to-all. These algorithms efficiently address the trade-off between bandwidth maximization and latency minimization that existing implementations struggle to optimize. We show a performance improvement over the state-of-the-art implementations by factors of 42x and 138x on Polaris and Fugaku, respectively.

ACM Reference Format:
Ke Fan[1], Jens Domke[2], Seydou Ba[2], and Sidharth Kumar[1]. 2025. Parameterized Algorithms for Non-uniform All-to-all. In *The 34th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '25), July 20–23, 2025, Notre Dame, IN, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3731545.3731590

1 INTRODUCTION

Motivation: Optimizing data movement remains a critical challenge in the era of exascale. Collective communication that involves data exchange among (almost) all processes is an important class of data movement. Owing to its global scope, collectives are typically difficult to scale and can consume a substantial portion of the overall execution time in applications, often accounting for between 25% and 50%, or more [3]. Machine learning (ML) applications, in particular, depend heavily on all-reduce and all-to-all (both *uniform* and *non-uniform*) collectives, which are crucial in efficiently shuffling data and synchronizing parameters during the parallel training process [7, 37]. Beyond ML, various HPC workloads heavily utilize *non-uniform* all-to-all communication. These include graph algorithms like PageRank [4], Fast Fourier Transform (FFT) computations [27], quantum computer simulations [35], and certain advanced preconditioners and solvers [8].

Limitation of state-of-art approaches: State-of-the-art implementations of *non-uniform* all-to-all communication typically rely on linear-time algorithms. In contrast, *uniform* all-to-all collective implementations utilize either linear-time algorithms (e.g., the scattered algorithm [1]) or logarithmic-time algorithms (e.g., the Bruck algorithm [32]), depending on the message sizes. Adapting logarithmic-time algorithms for *non-uniform* all-to-all communication is a challenging task and has only recently been explored [11]. Both existing log and linear time approaches for non-uniform all-to-all exchanges have limitations.

The log time approach presented in [11] works only for a fixed radix of 2, where it prioritizes low latency by requiring only $\log_2 P$ communication rounds, where P represents the number of processes. However, this approach significantly increases the total volume of data exchanged, making it most suitable for only small-sized messages that are dominated by latency. The inability to vary the radix and, therefore, tune the total number of communication rounds and the amount of data exchanged renders the approach usable for limited message sizes. An adjustable communication pattern with *tunable radix* could offer better performance by making calculated trade-offs: slightly increasing the overall workload to better utilize available bandwidth while simultaneously reducing latency. Our work aims to address this gap.

While the existing log approach lacks the ability to be tuned, the existing linear time approach (such as scattered) lacks the ability to take advantage of the hierarchical architecture of existing HPC systems. Today's HPC infrastructure consists of computational nodes containing multiple CPU sockets, each equipped with several processing cores. Data exchanges between cores within the same node experience significantly lower latency than those between cores on different nodes. By not leveraging this architectural hierarchy, existing linear approaches overlook opportunities for substantial performance optimization. Our work also aims to address this gap.

Key insights and contributions: Our research focuses on improving the efficiency of all-to-all communication for *non-uniform* data distributions, addressing the limitations of existing approaches. We introduce two novel algorithms called *ParLogNa* (*Par*ameterized *Log*arithm *N*on-uniform *A*ll-to-all) and *ParLinNa* (*Par*ameterized *Lin*ear *N*on-uniform *A*ll-to-all) specifically designed for *non-uniform all-to-all* workloads.

ParLogNa's key innovation lies in its ability to adjust the radix, which determines the base of the logarithmic complexity. This radix can be set anywhere between 2 and P. By allowing fine-grained control over the radix, *ParLogNa* enables users to optimize the

HPDC '25, July 20–23, 2025, Notre Dame, IN, USA

ACM ISBN 979-8-4007-1869-4/2025/07
https://doi.org/10.1145/3731545.3731590

trade-off between the number of communication rounds and the size of data exchanges, leading to improved performance scalability. *ParLogNa* is a first-of-its-kind parameterized algorithm designed for non-uniform all-to-all data exchanges that uniquely allows users to adjust the volume of data transferred and the number of communication rounds, enabling optimization of bandwidth usage and latency reduction.

ParLinNa represents an evolution of the scattered algorithm, operating as a near-linear-time solution with two distinctive capabilities. The algorithm's primary innovation lies in utilizing HPC system architecture through a decoupled communication structure. This structure separates data exchanges into two distinct phases: intra-node communication utilizing shared memory within nodes and inter-node communication facilitating message transfer across the network. Additionally, *ParLinNa* incorporates dual parametric controls, with separate tunable parameters governing the intra-node and inter-node communication phases, enabling precise performance optimization at both levels.

In summary, our paper makes the following contributions:

(1) We develop *ParLogNa*, capable of adjusting the radix (r) from 2 to P. Evaluation of *ParLogNa* reveals: small radices work for small messages, a radix close to $\sqrt{P}$ improves mid-sized communication, and large radices work for large messages.
(2) We develop a near-linear-time algorithm, *ParLinNa*, which decouples communication into local intra-node and global inter-node data exchange phases to improve performance further.
(3) We perform a detailed evaluation of our techniques using scaling studies (up to 16k processes) on Fugaku [30] and Polaris [2]. Our algorithms exhibit a performance improvement of 60.60× (*ParLogNa*), 138.59× (*ParLinNa*) over the vendor implementation of `MPI_Alltoallv`.

Experimental methodology and artifact availability: We thoroughly evaluated our algorithms using micro-benchmarks and real applications on two supercomputers: Polaris at Argonne National Laboratory and Fugaku at RIKEN R-CCS. In evaluating our approach, we used two complementary comparison strategies. We benchmarked against the vendor-optimized `MPI_Alltoallv` implementation, which automatically selects algorithms based on runtime parameters such as the size of data-blocks (S) and P. However, recognizing that this automatic selection might not always identify the optimal algorithm, we took an additional step: we individually implemented and tested every available alltoallv algorithm from both the OpenMPI [15] and MPICH [18] libraries. This exhaustive testing, documented in Section 5.3 (Figures 11 and 12), ensured we were comparing our method against the best possible existing MPI implementation rather than relying solely on MPI's automatic algorithm selection.

2 BACKGROUND

In this section, we provide a concise overview of the fundamental base algorithms to establish a clear context for our work.

(a) Bruck [6, 32] is a classic logarithmic *uniform* all-to-all algorithm with radix (base) 2. It comprises three phases: an initial rotation phase, a communication phase with $log_2 P$ rounds, and an inverse rotation phase. It is a store-and-forward algorithm, where

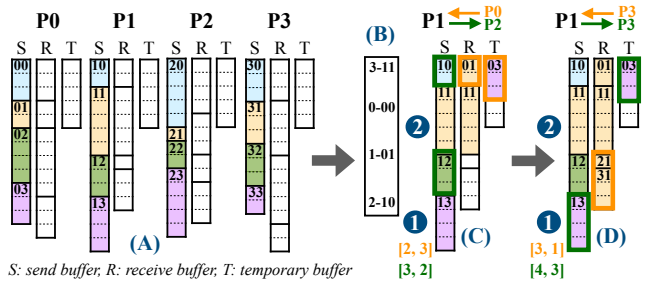

Figure 1: Example of the *ParLogNa* with $P = 4$ and $r = 2$. (A) is the initial state. S is made of 4 data-blocks (of different sizes), shown in different colors. (B) shows the rotated data-block indices and their matching binary representation for $P1$. (C) and (D) illustrate two communication rounds for $P1$. A two-phase communication scheme is employed in each round: ① metadata exchange, and ② actual data exchange.

data-blocks received during one round are forwarded in subsequent rounds for further transmission.

(b) Two-phase non-uniform Bruck [11] is a logarithmic *non-uniform* all-to-all algorithm with radix 2. Owing to the nature of the store-and-forward algorithm, adapting Bruck for *non-uniform* all-to-all requires that each process has: (1) prior knowledge of the data size they will receive during intermediate rounds, and (2) access to a large buffer to store intermediate data. The algorithm employs a coupled communication strategy consisting of metadata and data exchange phases to manage the *non-uniform* workload, while also using a large temporary buffer for intermediate data storage.

(c) Parameterized Uniform Bruck [13] is the generalized version of Bruck for *uniform* workloads, where the radix can be tuned between 2 and P. This gives the ability to tune the number of communication rounds and the amount of data exchanged. The paper [13] offers valuable insight into the selection of the optimal radix, notably by observing that $r = \sqrt{P}$ yields the best overall performance.

(d) Standard non-uniform all-to-all: In both MPICH and OpenMPI, `MPI_Alltoallv` implementations employ variants of the spread-out (linear) [22] algorithm. Spread-out schedules all send and receive requests in a round-robin order, ensuring that each process sends to a unique destination per round to avoid network congestion. The *scattered* algorithm in MPICH further improves this by dividing communication requests into batches, where a tunable parameter, *batch-size*, decides the size of batches. It waits for all the requests in one batch to be completed before moving on to the next, further reducing network congestion. OpenMPI's linear approach deviates from spread-out, as it initiates all communication in ascending rank order instead of round-robin. OpenMPI's other implementation, called *pairwise* algorithm, initiates a single receiving request with the non-blocking `Irecv` and opts for a blocking `Send`. It then awaits the completion of these two requests per communication round.

3 PARAMETERIZED LOGARITHMIC NON-UNIFORM ALL-TO-ALL (PARLOGNA)

In this section, we present the **Par**ameterized **Log**arithmic **N**on-uniform **A**ll-to-all (*ParLogNa*) algorithm, which facilitates *non-uniform* all-to-all data exchanges in a configurable number of communication rounds, parameterized with a tunable radix (r). *ParLogNa*

is built upon three key ideas: (1) a logarithmic-time generalized implementation of the Bruck-style algorithm with varying radices (see Section 3.1), (2) a two-phase data exchange mechanism in each communication round, comprising a metadata exchange followed by actual data transfer (see Section 3.2), and (3) a strategically sized temporary buffer (T) to support intermediate data exchanges during logarithmic communication rounds (see Section 3.3). To improve clarity, we also provide a table summarizing the definitions of notations frequently used in this paper (see Table 1).

3.1 Tunable radix

In non-uniform all-to-all communication, each process exchanges a distinct data-block with every other process, where the size of each individual data-block may vary. An all-to-all implementation accomplishes the entire communication through K rounds of point-to-point communication, during which a total of D data-blocks are exchanged. In Bruck, $K = \log_2 P$, and scattered, $K = P$. Within *ParLogNa*, both K and D are parameterized on the radix r and P as described below.

Every process encodes the indices of their P data-blocks using a r-base representation (see Figure 1 (B)). In this encoding scheme, the maximum number of digits required is denoted by $w = \lceil log_r P \rceil$, and each digit can assume one of r unique values. For example, in Figure 1, $w = log_2 4 = 2$, and each digit is 0 or 1. Consequently, a given data exchange round k ($0 \leq k < K$) can be uniquely identified by two variables: x ($0 \leq x < w$) and z ($1 \leq z < r$). The variable x represents the digit position within the r-base encoding, while z corresponds to the specific value of the digit at that position. This leads to $K \leq w \cdot (r-1)$ communication rounds. During each round, each process sends the data-blocks whose x_{th} digit matches the value z to the process with a rank distance of $z \cdot r^x$. For instance, in Figure 1 (C), *P1* sends the data-blocks whose 0_{th} digit equals 1 to *P2* whose rank distance is $1 \cdot 2^0 = 1$. Each process transmits up to r^{w-1} data-blocks per round, bounding D to $w \cdot (r-1) \cdot r^{w-1}$.

This communication strategy shows that both K and D are functions of r. These two parameters exhibit an inverse correlation, meaning an increase in K corresponds to a decrease in D, and vice versa. K represents the latency-related metric while D is the bandwidth-related metric. As a result, by increasing r, the algorithm can effectively transition from a latency-bound regime (low latency) to a bandwidth-bound regime (high bandwidth). This trade-off between K and D, facilitated by adjusting r, provides a mechanism for tuning the communication performance.

3.2 Two-phase communication

Although parametrization allows us to adjust K and D, it does not directly address the *non-uniform* nature of workloads. The parameterized Bruck [13] is not suitable for *non-uniform* in its current form. This is because the Bruck-style algorithm requires certain data blocks to traverse multiple intermediate steps before reaching their final destination. In addition, the algorithm is a store-and-forward approach, which utilizes send-and-receive buffers as temporary storage during the intermediate communication phases. To accommodate *non-uniform* data distributions, we incorporate two key elements: a two-phase communication scheme and a temporary buffer (T). The scheme facilitates the exchange of intermediate data blocks, while T provides the necessary storage for these blocks during the data exchange phase.

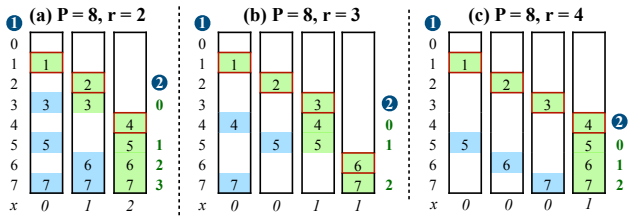

Figure 2: Examples of memory optimization with three configurations, each showing a single process and the data blocks exchanged per communication round. In each round, green blocks reach their destination, while blue blocks are temporarily stored in T for transfer in later rounds. Meanwhile, green blocks with red boxes are sent only once during the entire communication, allowing their space in T to be omitted.

The two-phase scheme is employed in each communication round; the first metadata-exchange phase transfers the size of each sent data-block, followed by the actual transmission of data. For instance, in Figure 1 (C), process *P1* needs to send data-blocks 12 and 10 to process *P2* (highlighted with green boxes). *P1* first sends an array $[3, 2]$ to *P2*, representing the sizes of the two data-blocks. During the data exchange phase, the sizes of the received data blocks may be larger than the sent ones in the send buffer or the corresponding segments in the receive buffer (R). To solve this issue, the algorithm employs a temporary buffer (T) to accommodate for all intermediate received data blocks that will be transferred again in subsequent rounds, while data blocks destined for the current process are stored in R. For instance, in Figure 1 (C), *P1* receives data-blocks 01 in R and 03 in T from *P0*. In the next round (Figure 1 (D)), *P1* sends the data-block 03 from T again to *P3*. Upon completing the communication phase, all processes receive the required data-blocks in R, which lie in ascending order, as illustrated in Figure 1 (D). Utilizing both T and R, *ParLogNa* effectively manages and rearranges the received data-blocks, eliminating the overhead associated with the final rotation phase.

Table 1: Definitions of Notations

P	Total number of processes	≥ 1
r	Base of the logarithmic complexity	$2 \leq r < P$
S	Maximum size of data-blocks	≥ 0 (bytes)
N	Number of nodes	≥ 1
Q	Number of processes per node	≥ 1
K	Number of communication rounds	$log_2 P \leq K < P$
w	Number of digits with base-r encoding	≥ 1
D	Number of exchanged data-blocks	$P \leq D < K \cdot r^{w-1}$

3.3 Estimating temporary buffer size

Previous studies on modifying Bruck-style methods for *non-uniform* workloads, including the *two-phase non-uniform Bruck*, adopted a specific approach to temporary buffer sizing [11, 36]. They designed the temporary buffer (T) to accommodate all data-blocks by setting its size to the product of two factors: the maximum block size (S)

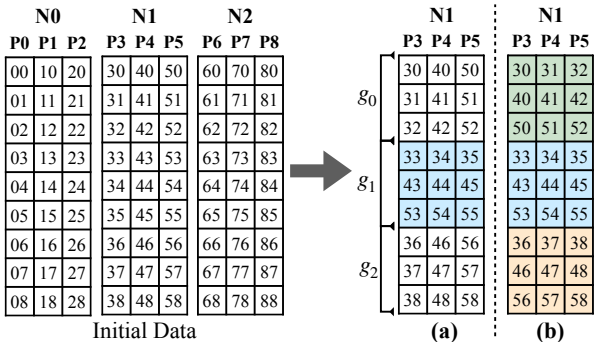

Figure 3: Two intra-node strategies: (a) explicit and (b) implicit (ours). Assuming data blocks on each node are logically divided into $N = 3$ groups, each process within a node has $Q = 3$ data-blocks per group. An explicit strategy performs all-to-all only within the group whose index matches the node's ID. Our approach performs all-to-all within each group.

across all processes and P. While this approach works, it is wasteful in its memory requirements and can lead to memory overflow for large values S or P. We observe that T only needs to store a certain number of intermediate data-blocks, and therefore, a tighter bound on the size of T can be obtained. To this end, we performed a theoretical analysis of the underlying communication pattern.

We observe that for every communication round, a process sends at most r^x data-blocks that reach their destination. In contrast, the remaining blocks are only transferred to some intermediate process that gets sent in the coming communication rounds. Here, x refers to the digit of indices of data-blocks in r-base encoding, ranging from 0 to w (see Section 3.1). For instance, Figure 2 (a) with $P = 8$ processes and $r = 2$ requires three rounds. In each round, the data-blocks needed to be temporarily stored in T are marked in blue, while those reaching their destination are marked in green. For example, there are $2^1 = 2$ green data-blocks in the second communication round. Notably, we observed that in each round, the first data-block is always sent directly to its destination process without further transfers in subsequent rounds, referred to as the *direct* data-block. In Figure 2 (a), the green data-blocks highlighted with red boxes indicate the *direct* data-block. Since *direct* data-blocks do not need to be stored in T, we can put a tighter bound on their size. A process can set T to store $B = (P - (K + 1))$ data-blocks, accounting for one block destined for itself. We note that (B) is a function of both r and P, the value of which decreases as r increases for a given P. For instance, in Figure 2, r takes values of 2, 3, and 4 in subfigures (a), (b), and (c), respectively. The corresponding value of B for the three radices is 4, 3, and 3. This approach significantly reduces the memory footprint for T when using a high radix r. In particular, when r exceeds $(P-2)$, no temporary buffer is required (equivalent to a linear-time scattered algorithm).

While our approach minimizes the size of T, it introduces the challenge of mapping the indices of the data-block (referred to as o) into T. For example, in Figure 2 (a), ① indicates the original indices (o) of data-blocks, ranging from 0 to 7 and ② represents the corresponding mapped indices (t) in T, ranging from 0 to 3. Such as $o = 3$ maps $t = 0$ while $o = 5$ maps $t = 1$. The new position (t) is

Algorithm 1 *ParLogNa* Algorithm

```
1:  Find maximum data-block size S with MPI_Allreduce;
2:  Allocate a temporary buffer T with necessary length;
3:  Allocate rotation array I for each process p;
4:  I[i] = (2 × p − i + P) % P,  i ∈ [0, P];
5:  for x ∈ [0, w] do
6:    for z ∈ [1, r] do
7:      n = 0;
8:      for i ∈ [0, P] whose x^th digit of r-encoding is 1 do
9:        sd[n++] = (p + i) % P        /* sd: the array for n send
          data-block indices */
10:     end for
11:     sendrank = (p − z × r^x + P) % P;
12:     recvrank = (p + z × r^x) % P;
13:     Send metadata to sendrank and receive updated metadata
        from recvrank;
14:     Send sent data-block I[sd[i]] (i ∈ [0, n]) to sendrank;
15:     if (i % r^x == 0) (i ∈ [0, n]) then
16:       Receive data-block i into R from recvrank;
17:     else
18:       Receive data-block i into T from recvrank;
19:     end if
20:   end for
21: end for
```

calculated as $t = o - 1 - dx \cdot (r-1) - dz$; this corresponds to taking the block's original index (o) and subtracting the number of *direct* data-blocks with lower indices than the current block. Variables dx and dz, mirrors x and z explained in Section 3.1. $dx = \lceil log_r o \rceil$ represents the highest digit when encoding index o of a data-block in the r-encoding, and $dz = o/r^d x$ represents the value of that digit.

Algorithm: The pseudocode of *ParLogNa* is shown in Algorithm 1. It shows the three core ideas of *ParLogNa*: (1) a logarithmic number of data exchange phases executed using a tunable number of communication rounds. The number of rounds is parameterized by the variables x and z, as shown in lines 5 and 6. (2) Each data exchange phase comprises two sub-phases: a meta-data exchange and the actual data exchange. This two-phase approach is evident in lines 13 and 14 of the pseudocode. (3) Creation and usage of an optimally sized temporary buffer (T), as shown in lines 2 and 18.

4 PARAMETERIZED LINEAR NON-UNIFORM ALL-TO-ALL (PARLINNA)

Modern HPC systems feature a hierarchical architecture, in which each computing node comprises multiple CPU cores with shared memory access [20]. This facilitates rapid intra-node data exchanges via direct memory transfers, typically significantly faster than inter-node exchanges (over the network). To leverage this hierarchical architecture, we present the **Par**ameterized **Lin**ear **N**on-uniform **A**ll-to-all (*ParLinNa*) algorithm. This algorithm has two salient features. First, it leverages the hierarchical structure of HPC systems by decoupling the communication rounds into two phases: (1) intra-node communication and (2) inter-node communication. The intra-node phase uses shared memory within a node, while the inter-node communication transfers messages over the network.

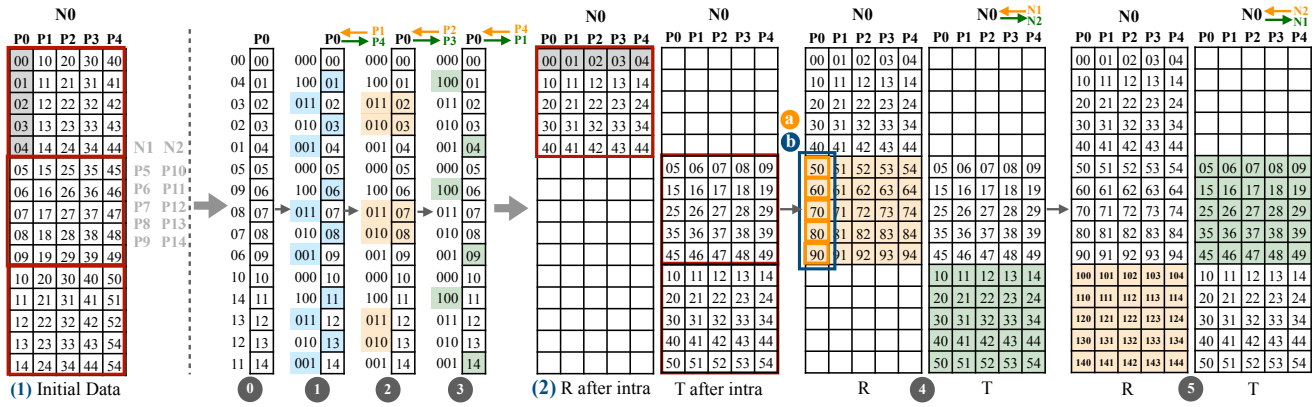

Figure 4: An example of *ParLinNa* when $P = 15$, $N = 3$, $r = 2$ and $Q = 5$. (1) depicts the initial state in send buffer for all processes within node $N0$. Each process logically has Q data-blocks in every N group (separated by red boxes). ⓪ shows the rotation index array for $P0$ based on the group rank ID ($g = p \% Q$). ① ② ③ illustrate the three intra-node communication steps, where the sent data-blocks are colored. (2) presents the data status in receive (R) and temporary (T) buffers after the intra-node communication, where R holds the data-blocks destined the processes within $N0$. ④ ⑤ depict two steps for inter-node communication. ⓐ and ⓑ are two communication patterns (matching Figure 5). Finally, each process receives the required data-blocks in R.

Second, it features two tunable parameters, one for the intra-node communication and the other for inter-node communication.

The two distinct communication phases of *ParLinNa* exhibit unique performance characteristics. In the intra-node phase, data transfers occur within localized process groups. Due to two key factors, the limited number of participating processes and the minimal cost of data exchange at this stage, performance in this phase is primarily determined by latency considerations rather than bandwidth constraints. Therefore, to optimize performance, *ParLinNa* employs *ParLogNa* for intra-node communication. While the *ParLogNa* implementation results in increased data exchange between processes, the cost of these transfers is negligible within a node. This design choice effectively reduces latency overhead, leading to improved overall performance. Conversely, the inter-node phase is characterized by aggregated data transfers occurring at the node level, wherein processes within a single node engage in comprehensive communication with all processes residing on other nodes. This scenario introduces bandwidth limitations due to the shared nature of network resources among co-located processes, leading to bandwidth-constrained communication patterns. Therefore, to keep the number of data exchanges in the network to a minimum, we design *ParLinNa* to use the linear-time scattered algorithm for inter-node communication. It therefore places *ParLinNa* within the category of linear algorithms, as its primary data-intensive communication operations are executed using a linear approach.

ParLinNa algorithm then incorporates dual parameterization mechanisms to optimize communication efficiency. For intra-node communication, it employs a radix parameter, which establishes an equilibrium between latency and bandwidth utilization. Concurrently, for inter-node communication, it implements a *batch_size* parameter. It waits (blocks) for all the requests in one batch to be completed (in a non-blocking way) before moving on to the next batch. This approach regulates the concurrent network communication requests, achieving a balanced state between blocking and non-blocking communication patterns, thus relieving potential network congestion. These configurable parameters collectively facilitate performance optimization. Finally, we also introduce two implementations of *ParLinNa* based on the distinct communication patterns of the inter-node (see Section 4.2): (1) staggered *ParLinNa* and (2) coalesced *ParLinNa*.

It is important to note that while *ParLogNa* can be configured to operate with a linear number of rounds using higher radix values, its fundamental store-and-forward architecture results in blocking operations. Unlike scattered, which delivers data-blocks directly to their destinations using non-blocking operations. As we will see later in Section 5, this blocking characteristic reduces *ParLogNa*'s effectiveness for a wide range of input configurations when compared to *ParLinNa*, where the degree of blocking and non-blocking data exchanges can be tuned by the *batch_size* parameter.

4.1 Hierarchical *ParLinNa* algorithm phases

ParLinNa algorithm is composed of intra-node and inter-node data exchanges, which we now present in detail.

Intra-node communication. With a total of $P = Q \cdot N$ processes, where Q represents the number of processes per node and N represents the total number of nodes, the intra-node data exchange phase consists of N concurrent all-to-all exchanges, each of which involves Q processes. In standard all-to-all, each process (p) must send one data-block (i) to process i and receive one data-block (p) from process i, with the total number of data-blocks equalling the number of processes (P). In our intra-node communication, all P processes are logically grouped into N groups (indexed $0, \ldots, N-1$), each containing Q processes. Subsequently, we group the P data-blocks into N groups, with each group handling Q data-blocks. We then perform all-to-all exchanges concurrently in each group using the *ParLogNa* algorithm. Figure 3 (b) shows an example involving three nodes, where we perform three concurrent $Q \times Q$ all-to-all

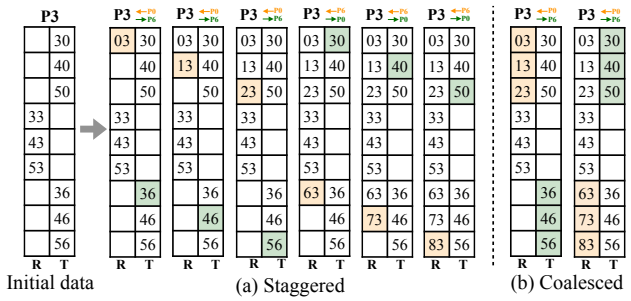

Figure 5: Two inter-node communication patterns: (a) staggered and (b) coalesced. Taking one process in the first node *P3* as an example, each process in (a) sends/receives one data-block to the same destination per round, requiring $(N-1)\cdot Q$ rounds. In (b), each process sends/receives Q data-blocks per round, requiring $(N-1)$ rounds. See Section 4.2 for details.

exchanges for each node. Our approach differs from an explicit approach, which creates a local sub-communicator for every group by splitting the MPI communicator using `MPI_Comm_split`. During the all-to-all exchange, each process is aware only of the size of the data block destined for itself. Therefore, Q local processes perform only a local all-to-all exchange within the group whose index matches the node ID (n) (see Figure 3 (a)). Unlike our approach, which allows for a simultaneous all-to-all exchange within every group. Our implicit approach avoids the overhead of creating new local communicators and also better prepares the data-blocks for the subsequent inter-node data exchange.

We note that to achieve our implicit strategy, a metadata exchange is required to exchange the data-block sizes destined for processes in other nodes. Fortunately, the intra-node communication phase of the *ParLinNa* algorithm employs the *ParLogNa* algorithm, which internally includes a metadata exchange phase, requiring no extra cost. Finally, we note that the intra-node communication phase in *ParLinNa* is implemented through *ParLogNa* and is thus tunable with a radix $r \in [0, \ldots, Q]$.

Inter-node communication. This phase conducts all-to-all communication between nodes. In this phase, all Q processes within a node must communicate with all Q processes in another node. The communication process pairs have the same number of group ID ($g = p \ \% \ Q$). This follows the Q-port model, in which every Q point-to-point data exchange is delivered simultaneously. Each node serves as a communication port, transmitting a $^1/_Q$ message to the corresponding process in another node. For example, in Figure 4 ④, node *N0* needs to send all green data-blocks in T to node *N2*, while receiving all orange data-block in R from node *N1*. In this case, each process sends its own Q orange data-blocks to the matching process. The inter-node communication utilizes the scattered algorithm with an adjustable *batch_size* to manage the communication load. This algorithm divides the communication requests into manageable batches, executed sequentially to mitigate network congestion. The *batch_size* parameter determines the batch size, which can significantly influence the overall performance.

This inter-node communication is analogous to the concept of inter-group communication in MPI, implementable by using the `MPI_Intercomm_create` routine, creating an inter-communicator

Algorithm 2 Coalesced *ParLinNa* Algorithm

1: Allocate a temporary buffer T with necessary length;
2: Compute rank id in each group: $g = p \ \% \ Q$;
3: Compute node id: $n = p \ / \ Q$;
4: Allocate rotation array I for each process p;
5: $I[i * Q + j] = i \times Q + (2 \times g - j + Q) \ \% \ Q$; $i \in [0, N]$, $j \in [0, Q]$;
6: **for** $x \in [0, w]$ **do**
7: **for** $z \in [1, r]$ **do**
8: $n = 0$;
9: **for** $i \in [0, P]$ whose x^{th} digit of *r-base* is 1 **do**
10: $sd[n{+}{+}] = n \times Q + (g + i) \ \% \ Q$ /* sd: the array for n send data-block indices*/
11: **end for**
12: $sendrank = n \times Q + (g - z \times r^x + Q) \ \% \ Q$;
13: $recvrank = n \times Q + (g + z \times r^x) \ \% \ Q$;
14: Send metadata to *sendrank* and receive updated metadata from *recvrank*;
15: Send sent data-block $I[sd[i]]$ ($i \in [0, n]$) to *sendrank*;
16: Receive data-block i into T from *recvrank*;
17: **end for**
18: **end for**
19: Rearrange T to removing empty data-blocks;
20: **for** ($ii = 0; ii < N; ii \mathrel{+}= batch\_size$) **do**
21: **for** $i \in [0, batch\_size]$ **do**
22: $nsrc = (n + i + ii) \ \% \ Q$; $src = nsrc \times Q + g$;
23: Receive data-blocks ranging from *nsrc* to $(nsrc + Q)$ from *src*;
24: **end for**
25: **for** $i \in [0, batch\_size]$ **do**
26: $ndst = (n - i - ii + Q) \ \% \ Q$; $dst = ndst \times Q + g$;
27: Send data-blocks ranging from *ndst* to $(ndst + Q)$ to *ndst*;
28: **end for**
29: Wait for communication completion using `MPI_Waitall`;
30: **end for**

to define the local and remote groups. All processes in the local group exchange data with all processes in the remote group. However, MPI's inter-group approach is limited to only communication between exactly two groups, posing a constraint on scalability and flexibility. In contrast, we implement the inter-node exchange using point-to-point data exchanges, explicitly computing the ranks of all point-to-point exchange pairs, allowing our approach to support concurrent inter-communication among multiple groups.

4.2 Staggered and coalesced *ParLinNa*

To further optimize the *ParLinNa* algorithm across diverse communication scenarios, we develop two variants based on distinct inter-node communication patterns (see Figure 5): (a) staggered *ParLinNa* and (b) coalesced *ParLinNa*. The staggered communication pattern involves sequentially exchanging one data-block per communication round with the target process, completing the communication within two nodes over Q rounds (see Figure 5 (a)). Inter-node communication requires $(N-1)$ exchanges between nodes. This method, therefore, requires a total of $Q \cdot (N-1)$ communication rounds. Conversely, the coalesced one consolidates the transmission, sending all Q data-blocks in a single round to the

Algorithm 3 Staggered *ParLinNa* Algorithm

```
1:  Same intra-node communication with coalesced ParLinNa
2:  for (ii = 0; ii < P; ii += batch_size) do
3:    for i ∈ [0, batch_size] do
4:      gi = (ii + i)/n; gr = (ii + i) % n; nsrc = (g + gi) % N;
        /* g: rank id in each node; n: node id. */
5:      if (nsrc != g) then
6:        d = nsrc * Q + gr; src = nsrc * n + g;
7:        Receive data-blocks d from src using MPI_Irecv;
8:      end if
9:    end for
10:   for i ∈ [0, batch_size] do
11:     gi = (ii + i)/n; gr = (ii + i) % n; ndst = (g − gi + N) % N;
12:     if (ndst != g) then
13:       d = ndst * Q + gr; dst = ndst * n + g;
14:       Send data-blocks d to dst using MPI_Isend;
15:     end if
16:   end for
17:   Wait for communication completion using MPI_Waitall
18: end for
```

target process (see Figure 5 (b)). This approach yields $(N-1)$ communication rounds. Theoretically, the coalesced variant is effective for short-message scenarios, where the number of communication rounds dominates the performance. In contrast, the staggered one is more appropriate for bandwidth-bound long-message scenarios.

Algorithms: Algorithms 3 and 2 provide pseudocode for the staggered and coalesced *ParLinNa*, respectively. Both share the same intra-node communication phase, detailed in lines 5 to 18 of Algorithm 2. The inter-node communication phases are outlined in lines 20 to 30 of Algorithm 2 for the coalesced method and lines 2 to 18 of Algorithm 3 for the staggered method. It is important to note that after the intra-node communication, non-continuous data-blocks are stored in the temporary buffer T. Hence, a local data rearrangement is necessary to eliminate any empty intermediate segments in T, thus streamlining the buffer for efficient coalesced inter-node communication.

5 EVALUATION

We thoroughly evaluated our algorithms using micro-benchmarks on two production supercomputers: Polaris at Argonne National Laboratory and Fugaku at RIKEN R-CCS. Polaris' 560 nodes have 32-core AMD CPUs and 4 Nvidia A100 GPUs each, totaling a peak performance of 44 petaflop/s. A Slingshot-based Dragonfly topology connects the nodes. The 488 Pflop/s Fugaku fields 158,976 compute nodes, each hosting 48 user-accessible A64FX cores. Fugaku's network is a 6D-torus Tofu-D interconnect. Polaris utilizes Cray MPICH version 8.1.16, whereas Fugaku employs Fujitsu MPI version 4.12.0, which is based on OpenMPI.

To demonstrate the efficacy of *ParLogNa* and *ParLinNa*, we evaluate their performance against vendor-optimized, closed-source implementations of `MPI_Alltoallv` in Section 5.1 and Section 5.2. In addition to assessing the performance of standard *non-uniform* all-to-all communication algorithms, we implement the four algorithms in OpenMPI and MPICH (detailed in Section 2). Subsequently, we compare our algorithms, optimized with the best parameter

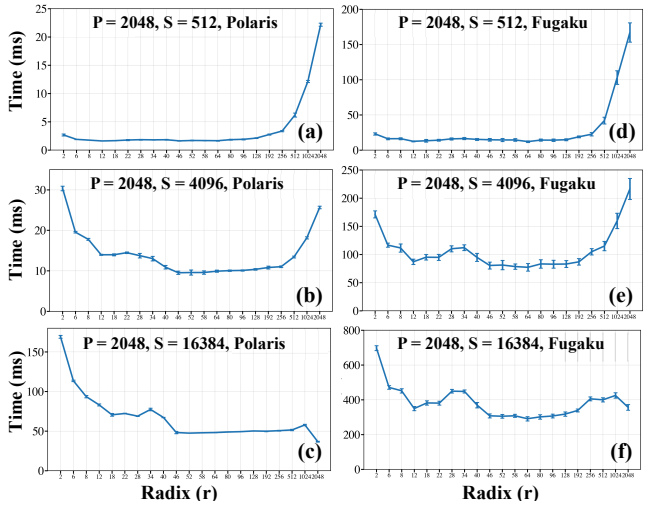

Figure 6: Three trends of *ParLogNa* on Polaris and Fugaku

configurations, against the top-performing `MPI_Alltoallv` benchmark to demonstrate the efficiency of our approach (Section 5.3). Finally, we demonstrate the effectiveness and generalizability of our algorithm using two standard data distributions (Section 5.4).

In all the aforementioned experiments, we vary P, r, and maximum size of data-blocks (S). By studying these parameters, we aim to determine optimal settings and evaluate how well our algorithms scale and adapt to various conditions. All our experiments are performed for at least 20 iterations, and we report the median and the standard deviation (using error bars). For Polaris and Fugaku, we utilized 32 processes per node in our experiments.

5.1 Performance analysis of *ParLogNa*

In our experiments, every process generates data-blocks whose sizes follow the continuous uniform distribution. This distribution ensures that data-block sizes are randomly selected and uniformly sampled between 0 and S, thus yielding an average data block of size $S/2$. To understand the performance of *ParLogNa*, we varied S from 16 bytes to 16 KiB (generated using FP64 vectors), r from 2 to P, and P from 512 to 16,384.

Three performance trends: Based on our experimental results, we identify three distinct performance trends for *ParLogNa* when increasing radix r, which are consistent across all process counts (P): (1) for small S ranging from 2 to 512 bytes, the performance of *ParLogNa* exhibits an increasing trend with increasing radices. (2) For medium S ranging from 512 to 8 KiB, the performance of *ParLogNa* follows a *U-shaped* trend. (3) For large S exceeding 8 KiB on Polaris and 32 KiB on Fugaku, the performance of *ParLogNa* shows a decreasing trend with increasing radix values. We can see these three trends in Figure 6 for $P = 2{,}048$. Our measured ideal r is around 2 for the first trend involving small messages. This is attributed to small-sized message communication being dominated by latency, which requires minimal communication rounds to achieve optimal performance. The U-shaped trend suggests a balance between latency and bandwidth is sought for middle-sized message communication. Prior work [13] has shown that $r \approx \sqrt{P}$ achieves this balance, minimizing the overall communication cost.

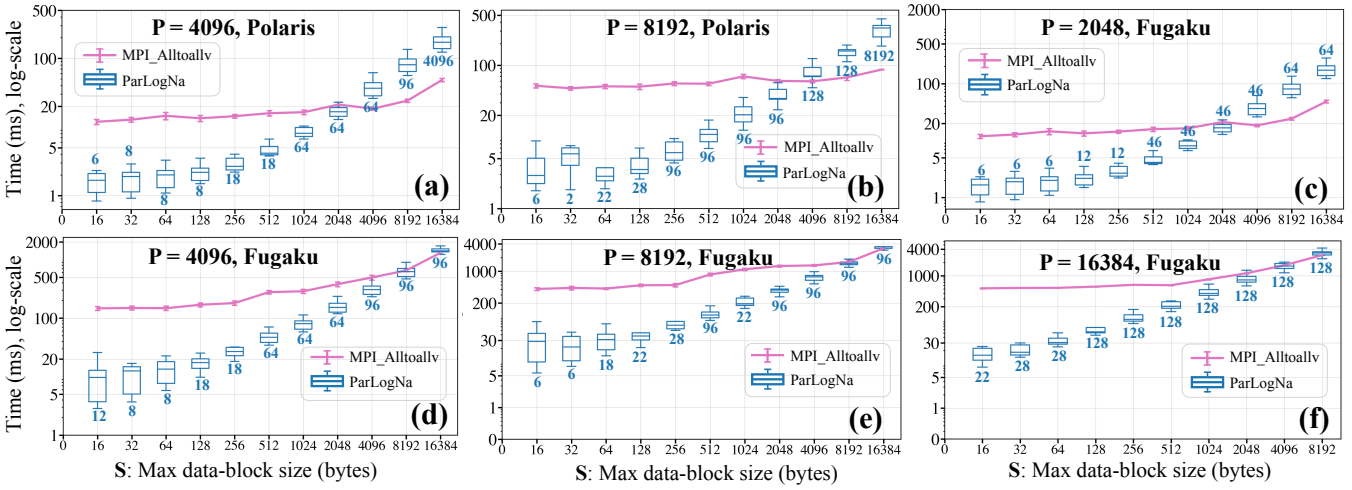

Figure 7: Comparing *ParLogNa* with `MPI_Alltoallv` on Polaris and Fugaku. Detailed analysis provided in Section 5.1.

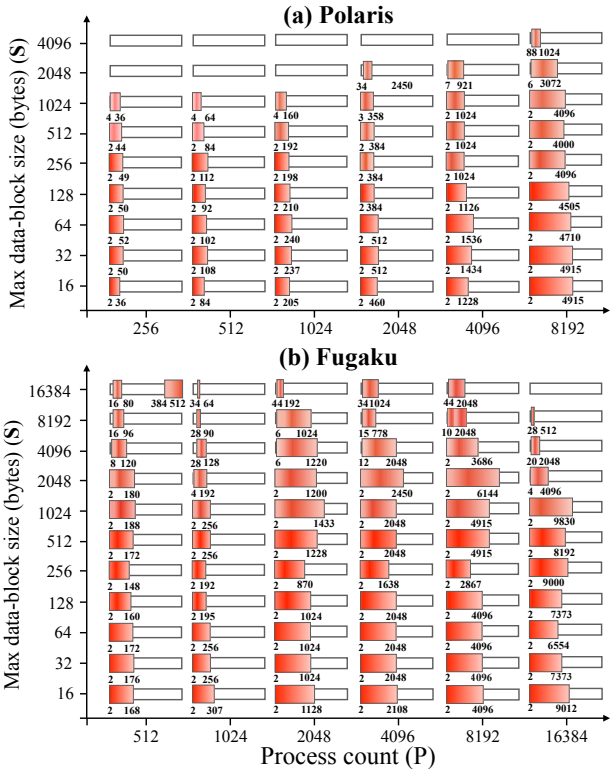

Figure 8: Ranges of radix where *ParLogNa* outperforms `MPI_Alltoallv` on (a) Polaris and (b) Fugaku, visualized through a series of heatmaps. Each heatmap (top box) corresponds to a P and S pair, where the intensity of the red color indicates the degree of performance advantage offered by *ParLogNa*. The bottom boxes indicate the entire radix range (from 2 to P). Refer to Section 5.1 for details.

The bandwidth dominates the performance for large message communication, where the total transferred data size across all rounds becomes the critical factor. Therefore, $r \approx P$ is the sweet spot, as it minimizes the total transmitted message size. Overall, the ideal r increases when S increases, transitioning from a latency-dominated regime to a bandwidth-dominated regime.

Performance comparison: Figure 7 shows our comparisons of *ParLogNa* against `MPI_Alltoallv`. The performance of *ParLogNa* is presented through box plots, each representing the range of performance across various radices $\in [2, \ldots, P]$. Our measured ideal r for each scenario is highlighted beneath its respective box, aligning with the above-observed trends of increasing r. We see that *ParLogNa* outperforms `MPI_Alltoallv` when S is no more than 2 KiB on Polaris and 16 KiB on Fugaku. Particularly, *ParLogNa* demonstrates significant performance advantages when S is less than 512 B on Polaris and 2 KiB on Fugaku. For instance, when P = 8,192 and S = 16 bytes, *ParLogNa* with ideal r is $^{51.8}/_{1.78}$ = 29× and $^{408.33}/_{5.79}$ = 70.48× faster than `MPI_Alltoallv` on Polaris and Fugaku, respectively. *ParLogNa* performs effectively with mid-ranged S. For example, it achieves 5.62× and 7.26× speedup on Polaris and Fugaku when S = 1,024 and P = 8,192. While larger S shows reduced performance on Polaris, *ParLogNa* still manages a speedup of 1.67× on Fugaku when P = 16,384 and S = 8,192.

Radix selection of *ParLogNa*: We finally summarize all our experiments in Figure 8, which is developed to show the optimal radix range of *ParLogNa* that outperforms the vendor-optimized implementation of `MPI_Alltoallv`. In this figure, P is represented on the x-axis and S on the y-axis. A combined shape containing two rectangles is presented for all combinations of P and S. The longer rectangle represents the entire range of radices for *ParLogNa*, ranging from 2 to P. The shorter rectangle, nested within the longer one, represents the specific range of radices for which *ParLogNa* outperforms `MPI_Alltoallv`. Additionally, the shorter rectangle is depicted as a heatmap, where the intensity of the red color indicates the degree of performance improvement achieved by *ParLogNa*. A stronger red color signifies a higher performance gain offered by *ParLogNa*. The previously mentioned trends can be observed from the heatmap, reinforcing the relationship between the optimal radix selection, P, and S. These figures provide a concise and visually intuitive representation of the performance landscape, enabling

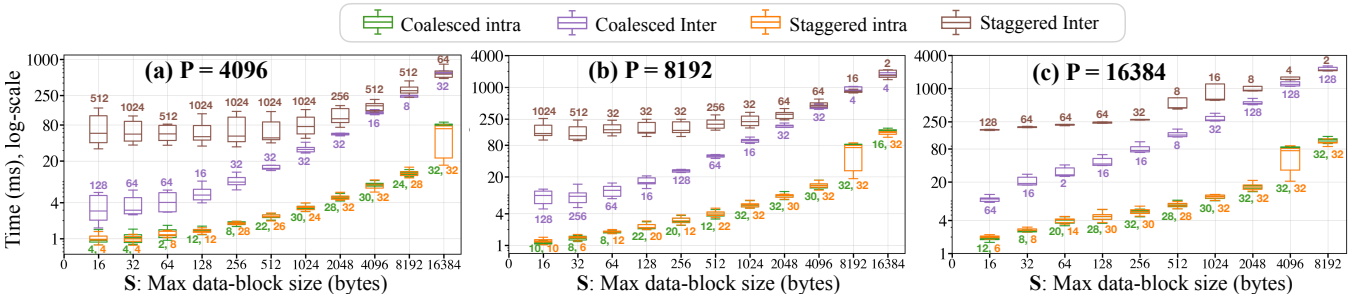

Figure 9: Comparing coalesced and staggered *ParLinNa* algorithms on Fugaku. Intra-node and inter-node data exchanges are plotted separately using box plots for each algorithm.

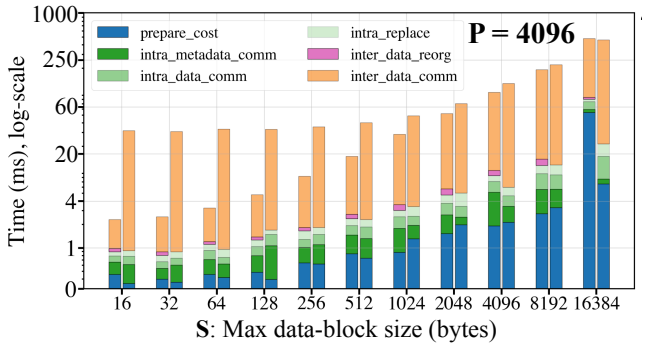

Figure 10: Breakdowns of coalesced (left bar) and staggered (right bar) *ParLinNa* algorithms on Fugaku.

us to make informed decisions when selecting suitable radices of *ParLogNa* for given P and S.

5.2 Performance analysis of coalesced and staggered *ParLinNa*

Both coalesced and staggered *ParLinNa* employ the *ParLogNa* algorithm for intra-node communication and the scattered algorithm for inter-node communication. The *ParLogNa* algorithm includes a configurable parameter, radix (r), which can be adjusted from 2 to Q. Similarly, the scattered algorithm features a tunable parameter, *batch_size*, which varies from 1 to $(N-1)$ for the coalesced variant and from 1 to $((N-1)\cdot Q)$ for the staggered variant. Figure 9 presents the performance of these algorithms on Fugaku using box plots. In the experiments, the number of processors (P) was varied from 4,096 to 16,384, and the message size (S) from 16 to 16 KiB. Each box plot, distinguished by unique colors, represents either the intra-node or inter-node communication performance with its corresponding tunable parameter. Green and purple boxes depict the intra-node and inter-node phases of the staggered variant, while orange and brown boxes illustrate these phases for the coalesced one. The ideal parameters for each configuration are indicated beneath each corresponding box plot with a matching color.

Parameter selection analysis: Figure 9 illustrates the trends in intra-node and inter-node communications under coalesced and staggered approaches using four box plots. For each box, the values of P and S are fixed, while the variations across different *radix* and *batch_size* configurations are summarized. The optimal values for these two parameters are annotated with colored text corresponding to each box, providing an intuitive visualization of the parameter selection trends. The figure explores a range of *radix* values from 2 to P and *batch_size* from 1 to P. From this figure, we observe that the choice of optimal radix for intra-node communication does not follow a strict pattern; however, smaller radices generally perform better for smaller S (under 1 KiB), while larger radices are better for larger S. The observed pattern is reasonable because communication involving small message sizes tends to be more affected by the total number of communication rounds compared to communication with larger message sizes. For inter-node communication, varying the *batch_size* (B) shows a clear trend with increasing S, where larger S typically favors smaller B. For example, at $P = 8{,}192$, the ideal B for staggered *ParLinNa* (indicated by the brown box) is 1,024 and 2 at $S = 16$ and 16 KiB, respectively. Moreover, as P increases, the ideal B for the same S tends to decrease. For example, with $S = 512$, the ideal B for $P = 4{,}096$, 8,192, and 16,384 is 1,024, 256, and 8, respectively. The choice of *batch_size* significantly impacts performance, particularly for the staggered variant, which makes more communication requests than the coalesced configuration.

Performance comparison: We also observe that the coalesced *ParLinNa* significantly outperforms the staggered one, particularly for small message sizes S. The staggered algorithm only exhibits competitive performance when S is at least 8 KiB. For instance, at $P = 4{,}096$ and $S = 16$, the coalesced is 17.06× faster than the staggered. Conversely, at $S = 16{,}384$ bytes, the staggered achieves a 1.23× speedup compared to the coalesced. Figure 10 provides a detailed breakdown of the coalesced (left-bar) and staggered (right-bar) algorithms. The algorithms are divided into six components: (1) prepare-cost, which includes all preparatory steps (see lines 1-7 and 11-13 in Algorithm 2); (2) and (3) correspond to metadata-cost (line 16) and data-cost (line 17), respectively; (4) replace-cost, which covers the cost of inter-data copying in each round (line 18); (5) data-rearrange time (line 21), applicable only to the coalesced algorithm; (6) inter-node comm cost (lines 22-32). The figure illustrates that the inter-node comm cost for the staggered is significantly higher than that for the coalesced.

5.3 Comparison with vendor-optimized MPI

To evaluate the efficiency of our proposed algorithms, we benchmarked the performance of four standard *non-uniform* all-to-all algorithms from MPI libraries (detailed in Section 2). Figure 11 presents a comparison of the default `MPI_Alltoallv` on Fugaku

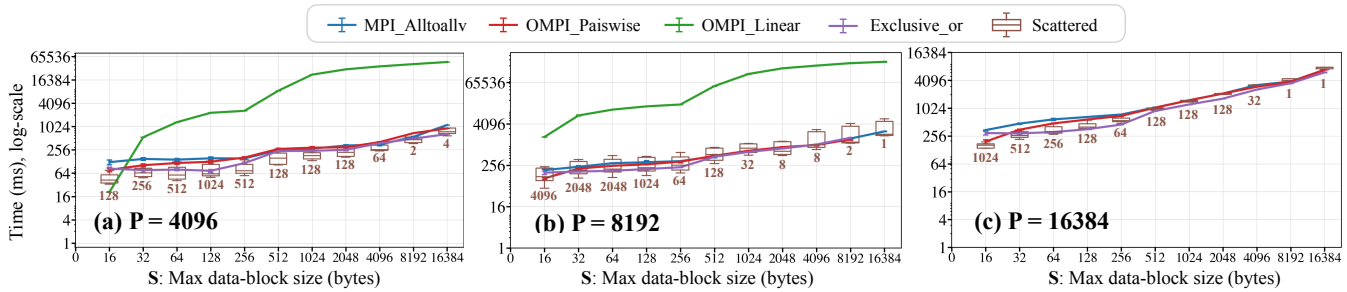

Figure 11: Benchmarking the non-uniform all-to-all implementations in OpenMPI and MPICH on Fugaku.

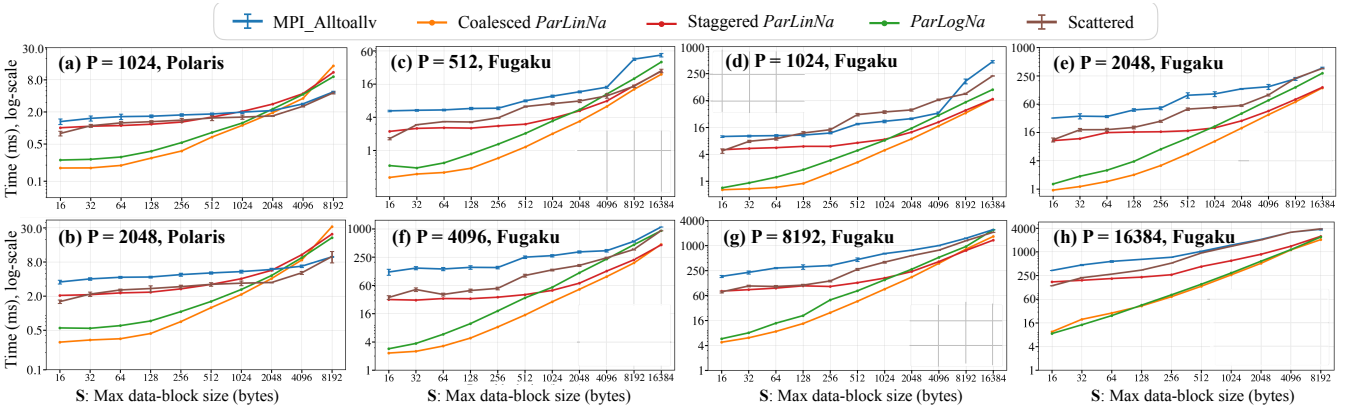

Figure 12: Comparing the proposed algorithms with the top-performing benchmarks. Refer to Section 5.3 for analysis.

and Polaris. Meanwhile, the performance of the scattered algorithm with a tunable *batch_size* (B) is displayed using a box plot. We observe that OpenMPI's linear algorithm, which is a blocking linear algorithm, performs the worst, particularly for larger P. The pairwise, exclusive-or, and default `MPI_Alltoallv` algorithms exhibit similar performance. Notably, when configured with ideal B, the scattered algorithm outperforms the others in most scenarios.

Performance comparison: Consequently, we compared our proposed algorithms, involving *ParLogNa*, coalesced and staggered *ParLinNa*, against both the scattered algorithm and the default `MPI_Alltoallv`. Each algorithm was configured with its ideal parameter. Figure 12 illustrates this comparison, where all proposed algorithms surpass the default `MPI_Alltoallv` across all scenarios and outperform the scattered algorithm in most instances. Specifically, the performance improvements with small message sizes S were notable, achieving maximum 60.60× (*ParLogNa*), 138.59× (coalesced), and 12.29× (staggered) speedups compared to `MPI_Alltoallv` on Fugaku. Among these, the coalesced *ParLinNa* consistently demonstrated the highest performance across all scenarios. For instance, at P = 16,384 on Fugaku, it is 42.08× and 14.61× faster than `MPI_Alltoallv` and the scattered algorithm at S = 16; and it maintained 2.20× and 2.14× improvements at S = 8,192. Additionally, although our proposed algorithms perform suboptimally with large S (larger than 2 KiB), the coalesced one achieves 11.68× and 2.71× speedup at S = 64 and 2 KiB.

5.4 Standard distributions

In addition to the uniform distribution presented previously, we further assess the effectiveness and generalization of our algorithms by validating them against two standard distributions: a power-law (exponential) distribution and a normal (Gaussian) distribution. Figure 13 (a) and (b) depict the communication data for process 0 with P = 4,096 on Fugaku, following the two distributions, with a maximum data block-size of 1,024 bytes.

Normal: Figure 13 (c) presents a weak scaling performance comparison of our algorithms against `MPI_Alltoallv` on Fugaku, with workload size following a normal distribution. The results show that all of our algorithms outperform `MPI_Alltoallv`, with coalesced *ParLinNa* demonstrating the best performance in almost all cases. In contrast, staggered *ParLinNa* performs worse than the other two proposed algorithms. For instance, at P = 4,096, *ParLogNa*, coalesced *ParLinNa*, and staggered *ParLinNa* are 3.21×, 3.63×, and 1.57× faster than `MPI_Alltoallv`, respectively. This observation is consistent with the results obtained under the uniform distribution.

Power-law: Similarly, Figure 13 (d) shows results for the power-law workload distribution, which is characterized by the rarity of large-sized data-blocks and the sparsity of the data distribution. From the figure, we can draw conclusions similar to those made for the uniform and normal distributions. Notably, both *ParLogNa* and coalesced *ParLinNa* significantly outperform `MPI_Alltoallv`, particularly at large scales.

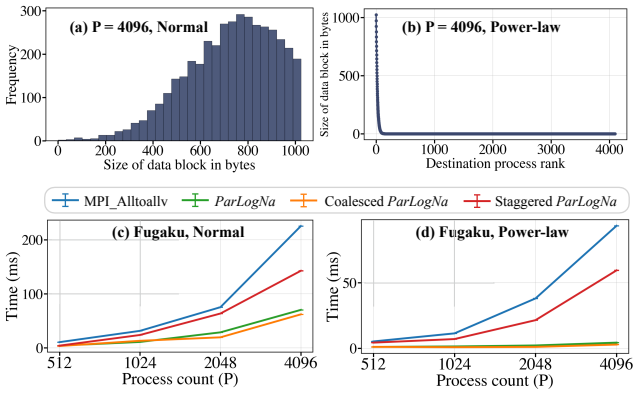

Figure 13: (a) histogram to show normal data distribution used (mean: 1,000, and standard deviation: 240) and (b) shows power-law data distribution (exponent: 0.95); (c) and (d) present a performance comparison for them.

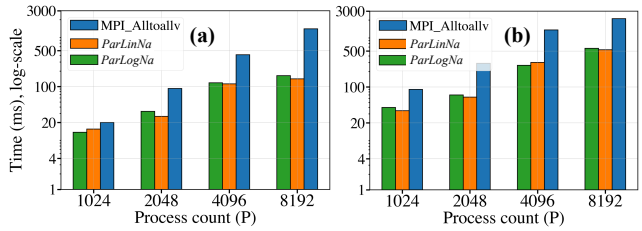

Figure 14: Performance of applying our algorithms to an FFT HPC workload with two different input sizes $\mathcal{N}_1$ and $\mathcal{N}_2$.

6 APPLICATIONS

In this section, we assess the performance of our algorithms using real-world applications, including Fast Fourier Transform (FFT) (see Section 6.1) and path-finding (see Section 6.2).

6.1 Fast Fourier transform

Fast Fourier transform (FFT) computations are crucial for many scientific domains, such as fluid dynamics and astrophysics [9]. Parallel FFT is characterized by performing three matrix transposes using all-to-all exchanges. FFW3, an open-source parallel FFT library, distributes the sub-problems evenly among processes. When the problem size $\mathcal{N}$ is not an integer multiple of P^2, *non-uniform* all-to-all exchanges are employed. The $\mathcal{N}$ data type is `fftw_complex`, comprising two FP64 values representing a complex number's real and imaginary components.

We perform two experiments to test the effectiveness of our algorithms, featuring different *non-uniform* data distributions using two values of $\mathcal{N}$: (1) $\mathcal{N}_1 = \lceil 0.78125 \cdot P \rceil \cdot \lceil P \cdot 0.625 \rceil \cdot 8$. This setup ensures that processes with ranks lower than $\lceil P \cdot 0.625 \rceil$ (referred to as *worker*) are assigned data, while the remaining ranks receive no data. Each *worker* fills the first $\lceil P \cdot 0.78125 \rceil$ data-blocks with 8 FP64 values. (2) $\mathcal{N}_2 = ((P-1) \cdot 32 + 8) \cdot P$. This leads to a near-uniform distribution where each process (except the last) transmits 64 FP64 values, and the last one transmits 16 FP64 values.

Figure 14 presents the comparative results of our algorithms against `MPI_Alltoallv`, using the two configurations mentioned above on Fugaku. We report the application runtime for the three approaches, dominated by all-to-all exchanges. For both experimental setups, all our algorithms outperform `MPI_Alltoallv` for all process counts (P). Consistent with our observations made in Section 5.3, the *ParLinNa* (coalesced) steadily outperforms the other algorithms. Additionally, our proposed algorithms demonstrate better performance for $\mathcal{N}_1$, which involves a smaller problem size. For example, when P = 8,192, the *ParLinNa* (coalesced) is 9.42× and 4.01× faster than `MPI_Alltoallv` for $\mathcal{N}_1$ and $\mathcal{N}_2$, respectively.

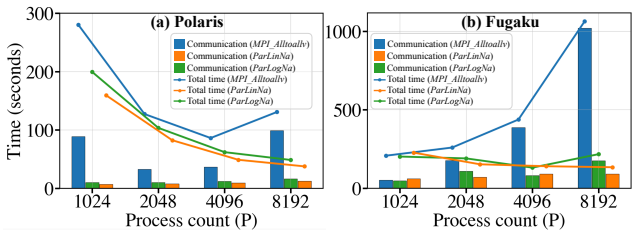

Figure 15: Performance of applying our algorithms to path finding (P = 8,192).

6.2 Graph Mining: path finding

We evaluate our algorithms by applying them to a popular graph mining algorithm that computes all reachable paths in a graph, also known as the transitive closure of a graph [25, 28]. The transitive closure (TC) of a graph can be computed through a classic fixed-point algorithm that repeatedly applies a relational algebra (RA) kernel to a graph G. This operation discovers paths of increasing length within the graph. The process continues until no new paths can be identified, reaching the fixed point. We use the MPI-based open-source library for parallel relational algebra [12, 17, 23, 24], which utilizes `MPI_Alltoallv` in each iteration of the fixed-point loop to shuffle data. Our proposed algorithms maintain the same function signature as `MPI_Alltoallv`, and hence, they can be seamlessly substituted in its place. We use a graph with 1,014,951 edges, sourced from the Suite Sparse Matrix Collection [10]. This graph undergoes repeated all-to-all across more than 5,800 iterations to reach the fixed point.

Figure 15 presents the strong scaling performance comparison of our proposed algorithms, configured ideally as described in Section 5, against the vendor-optimized `MPI_Alltoallv`, using the same graph on Polaris (a) and Fugaku (b). In both subfigures, we depict the communication overhead with bar charts and the total execution time with line charts. These figures highlight the critical role of all-to-all communication in this application. The results demonstrate that our proposed algorithms outperform `MPI_Alltoallv` in most cases. For instance, at P = 8,192, *ParLogNa* achieves speedups of 5.98× on Polaris and 5.80× on Fugaku, while *ParLinNa* (coalesced) delivers even greater improvements of 7.96× and 11.09×, respectively. Additionally, *ParLinNa* (coalesced) generally surpasses *ParLogNa* in performance, with the exceptions being cases that could be further optimized through parameter tuning. These observations align with those discussed in Section 5. It is worth noting that, despite the application's limited scalability, our proposed algorithms still provide significant performance gains over `MPI_Alltoallv` on both machines.

7 RELATED WORK

While substantial research efforts [14, 26, 33, 34] went into optimizing all-to-all for *uniform* messages, the exploration of *non-uniform* data-loads has received comparatively little attention. Most relevant to our work are studies [11, 36], which adapted Bruck's algorithm (with a radix of 2) for *non-uniform* all-to-all.

Bruck variants: Träff et al. [33] presented two improvements over Bruck, termed modified Bruck and zero-copy Bruck. The former omits the final rotation phase by rearranging data-blocks in an initial rotation. The latter aims at reducing internal memory duplications. Cong et al. [36] eliminates the shifting of data-blocks during the initial rotation by employing an index array that stores the desired order of data-blocks, effectively avoiding the actual movement of data. Subsequently, Fan et al. [11] further tuned Bruck by refine [33] and [36] to eliminate the initial and final rotations.

Tunable radix-based collectives: Gainaru et al. [16] studied logarithmic all-to-all algorithms with varying radices, but focused on exploring various memory layout configurations. Taru et al. [31] presented all-to-all with high radices. However, it does not conduct performance analysis of varying radices nor provide a heuristic for selecting radices. Andreas et al. [21] investigated the efficiency of collectives, with emphasis on *allgatherv*, *reduce-scatter*, and *allreduce* with varying radices. Their research indicates that high radices work for shorter messages and low radices work for longer ones.

Hierarchical collectives: Jackson et al. [19] introduced a planned *non-uniform* all-to-all, which transmits data from all processes on the same node to a designated master. Subsequently, only the master participates in a global exchange, reducing network congestion and hence demonstrating gains for concise messages. Similarly, Plummer et al. [29] segmented all processes into non-intersecting groups. Within each group, processes transfer data to the leading process, which then participates in the (sparse) all-to-all, improving scenarios where data load distribution is irregular but remains constant over time. Bienz et al. [5] proposed a locality-aware allgather built upon the Bruck algorithm, which clusters processes into groups of regions exhibiting low communication overhead.

8 CONCLUSION

We tackle the complex problem of optimizing the performance of *non-uniform* all-to-all exchanges by proposing two novel algorithms, *ParLogNa* and *ParLinNa*. *ParLogNa* is a parameterizable algorithm that outperforms vendor-optimized MPI implementations on two production supercomputers. Building upon *ParLogNa*, *ParLinNa* adds a hierarchical design to leverage the fast memory buffers of modern systems. Splitting the communication into node-local and global components further improves efficiency. Our experiments with up to 16k MPI ranks on up to 512 compute nodes show that *ParLinNa* outperforms both *ParLogNa* and the vendor-provided `MPI_alltoallv`. To showcase the effectiveness of our algorithms, we apply them to real-world applications, gaining nearly 10× speedup. In summary, our techniques can improve the performance of a wide range of applications relying on *non-uniform* all-to-all. Applications and vendors can easily adopt our open-source implementations, offering an interface equivalent to `MPI_Alltoallv` paired with tunable parameters for optimal performance.

9 ACKNOWLEDGEMENT

This work was funded in part by NSF PPoSS large grant CCF-2316157 and NSF SHF Small grant CCF-2221811. We are thankful to the ALCF's Director's Discretionary (DD) program for providing us with compute hours to run our experiments on the Polaris supercomputer located at the Argonne National Laboratory. We also extend our thanks to RIKEN for granting access to computing time on the Fugaku supercomputer at the RIKEN Center for Computational Science.

REFERENCES

[1] MPICH Home Page. https://www.mpich.org.
[2] Argonne National Laboratory. 2024. Polaris | Argonne Leadership Computing Facility. https://www.alcf.anl.gov/polaris.
[3] David E Bernholdt, Swen Boehm, George Bosilca, Manjunath Gorentla Venkata, Ryan E Grant, Thomas Naughton, Howard P Pritchard, Martin Schulz, and Geoffroy R Vallee. 2020. A survey of MPI usage in the US exascale computing project. *Concurrency and Computation: Practice and Experience* 32, 3 (2020), e4851.
[4] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '17)*. Association for Computing Machinery, New York, NY, USA, 93–104. https://doi.org/10.1145/3078597.3078616
[5] Amanda Bienz, Shreeman Gautam, and Amun Kharel. 2022. A locality-aware bruck allgather. In *Proceedings of the 29th European MPI Users' Group Meeting*. 18–26.
[6] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, Eli Upfal, and Derrick Weathersby. 1997. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on parallel and distributed systems* 8, 11 (1997), 1143–1156.
[7] Chen-Chun Chen, Kawthar Shafie Khorassani, Quentin G. Anthony, Aamir Shafi, Hari Subramoni, and Dhabaleswar K. Panda. 2022. Highly Efficient Alltoall and Alltoallv Communication Algorithms for GPU Systems. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 24–33. https://doi.org/10.1109/IPDPSW55747.2022.00014
[8] Gerald Collom, Rui Peng Li, and Amanda Bienz. 2023. Optimizing Irregular Communication with Neighborhood Collectives and Locality-Aware Parallelism. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W '23)*. Association for Computing Machinery, New York, NY, USA, 427–437.
[9] JW Cooley and JW Tukey. 1965. An algorithm for the machine computation of the complex fourier series, in mathematics of computation. *April* (1965).
[10] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages.
[11] Ke Fan, Thomas Gilray, Valerio Pascucci, Xuan Huang, Kristopher Micinski, and Sidharth Kumar. 2022. Optimizing the Bruck Algorithm for Non-uniform All-to-all Communication. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*. 172–184.
[12] Ke Fan, Kristopher Micinski, Thomas Gilray, and Sidharth Kumar. 2021. Exploring MPI Collective I/O and File-per-process I/O for Checkpointing a Logical Inference Task. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 965–972.
[13] Ke Fan, Steve Petruzza, Thomas Gilray, and Sidharth Kumar. 2024. Configurable Algorithms for All-to-All Collectives. In *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*. 1–12. https://doi.org/10.23919/ISC.2024.10528936
[14] Ahmad Faraj and Xin Yuan. 2005. Automatic generation and tuning of MPI collective communication routines. In *Proceedings of the 19th annual international conference on Supercomputing*. 393–402.
[15] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11*. Springer, 97–104.
[16] Ana Gainaru, Richard L. Graham, Artem Polyakov, and Gilad Shainer. 2016. Using infiniband hardware gather-scatter capabilities to optimize mpi all-to-all. In *Proceedings of the 23rd European MPI Users' Meeting*.
[17] Thomas Gilray, Sidharth Kumar, and Kristopher Micinski. 2021. Compiling data-parallel Datalog. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction* (Virtual, Republic of Korea) *(CC 2021)*. Association for Computing Machinery, New York, NY, USA, 23–35. https://doi.org/10.1145/

3446804.3446855

[18] William Gropp and Ewing Lusk. 1996. User's Guide for mpich, a Portable Implementation of MPI.

[19] Adrian Jackson and Stephen Booth. 2004. Planned AlltoAllv a Cluster Approach. (2004).

[20] Andreas Jocksch, Matthias Kraushaar, and David Daverio. 2019. Optimized all-to-all communication on multicore architectures applied to FFTs with pencil decomposition. *Concurrency and Computation: Practice and Experience* 31, 16 (2019), e4964.

[21] Andreas Jocksch, Noe Ohana, Emmanuel Lanti, Vasileios Karakasis, and Laurent Villard. 2020. Optimised allgatherv, reduce_scatter and allreduce communication in message-passing systems. *arXiv preprint arXiv:2006.13112* (2020).

[22] Qiao Kang, Robert Ross, Robert Latham, Sunwoo Lee, Ankit Agrawal, Alok Choudhary, and Wei-keng Liao. 2020. Improving all-to-many personalized communication in two-phase i/o. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–13.

[23] Sidharth Kumar and Thomas Gilray. 2019. Distributed Relational Algebra at Scale. In *International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE.

[24] Sidharth Kumar and Thomas Gilray. 2020. Load-Balancing Parallel Relational Algebra. In *High Performance Computing: 35th International Conference, ISC High Performance 2020, Frankfurt/Main, Germany, June 22–25, 2020, Proceedings* (Frankfurt am Main, Germany). Springer-Verlag, Berlin, Heidelberg, 288–308. https://doi.org/10.1007/978-3-030-50743-5_15

[25] Walaa Eldin Moustafa, Vicky Papavasileiou, Ken Yocum, and Alin Deutsch. 2016. Datalography: Scaling datalog graph analytics on graph processing systems. In *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 56–65.

[26] Naeris Netterville, Ke Fan, Sidharth Kumar, and Thomas Gilray. 2022. A Visual Guide to MPI All-to-all. In *2022 IEEE 29th International Conference on High Performance Computing, Data and Analytics Workshop (HiPCW)*. IEEE, 20–27.

[27] NVIDIA Corporation. 2022. Multinode Multi-GPU: Using NVIDIA cuFFTMp FFTs at Scale. https://developer.nvidia.com/blog/multinode-multi-gpu-using-nvidia-cufftmp-ffts-at-scale/.

[28] Sarthak Patel, Bhrugu Dave, Smit Kumbhani, Mihir Desai, Sidharth Kumar, and Bhaskar Chaudhury. 2021. Scalable parallel algorithm for fast computation of Transitive Closure of Graphs on Shared Memory Architectures. In *2021 IEEE/ACM 6th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*. IEEE, 1–9.

[29] Martin Plummer and Keith Refson. 2004. An lpar-customized mpi alltoallv for the materials science code castep. *Technical Report, EPCC (Edinburgh Parallel Computing Centre)* (2004).

[30] Mitsuhisa Sato, Yutaka Ishikawa, Hirofumi Tomita, Yuetsu Kodama, Tetsuya Odajima, Miwako Tsuji, Hisashi Yashiro, Masaki Aoki, Naoyuki Shida, Ikuo Miyoshi, Kouichi Hirai, Atsushi Furuya, Akira Asato, Kuniki Morita, and Toshiyuki Shimizu. 2020. Co-Design for A64FX Manycore Processor and "Fugaku". In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '20)*. IEEE Press, Atlanta, GA, USA, 1–15.

[31] Doodi Taru, Nusrat Islam, Gengbin Zheng, Rubasri Kalidas, Akhil Langer, and Maria Garzaran. 2021. High Radix Collective Algorithms. *Proceedings of EuroMPI 2021* (2021).

[32] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications* 19, 1 (2005), 49–66.

[33] Jesper Larsson Träff, Antoine Rougier, and Sascha Hunold. 2014. Implementing a classic: Zero-copy all-to-all communication with MPI datatypes. In *Proceedings of the 28th ACM international conference on Supercomputing*.

[34] Manjunath Gorentla Venkata, Richard L Graham, Joshua Ladd, and Pavel Shamis. 2012. Exploring the all-to-all collective optimization space with connectx corredirect. In *2012 41st International Conference on Parallel Processing*. IEEE.

[35] Dennis Willsch, Madita Willsch, Fengping Jin, Hans De Raedt, and Kristel Michielsen. 2023. Large-Scale Simulation of Shor's Quantum Factoring Algorithm. *Mathematics* 11, 19 (2023).

[36] Cong Xu, Manjunath Gorentla Venkata, Richard L Graham, Yandong Wang, Zhuo Liu, and Weikuan Yu. 2013. Sloavx: Scalable logarithmic alltoallv algorithm for hierarchical multicore systems. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE, 369–376.

[37] Q. Zhou, B. Ramesh, A. Shafi, M. Abduljabbar, H. Subramoni, and D. Panda. 2024. Accelerating MPI AllReduce Communication with Efficient GPU-Based Compression Schemes on Modern GPU Clusters. In *ISC HIGH PERFORMANCE 2024*.