Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Scalable Algorithms for MPI Intergroup Allgather and Allgatherv☆,☆☆

Qiao Kang[a],*, Jesper Larsson Träff[b], Reda Al-Bahrani[a], Ankit Agrawal[a], Alok Choudhary[a], Wei-keng Liao[a]

[a] Department of Electrical and Computer Engineering, Northwestern University, USA
[b] Faculty of Informatics, Vienna University of Technology, Austria

A R T I C L E   I N F O

Article history:
Received 20 October 2018
Revised 6 February 2019
Accepted 29 April 2019
Available online 30 April 2019

Keywords:
Intergroup collective communication
All-to-all broadcast
Allgather
Allgatherv

A B S T R A C T

MPI intergroup collective communication defines message transfer patterns between two disjoint groups of MPI processes. Such patterns occur in coupled applications, and in modern scientific application workflows, mostly with large data sizes. However, current implementations in MPI production libraries adopt the "root gathering algorithm", which does not achieve optimal communication transfer time. In this paper, we propose algorithms for the intergroup Allgather and Allgatherv communication operations under single-port communication constraints. We implement the new algorithms using MPI point-to-point and standard intra-communicator collective communication functions. We evaluate their performance on the Cori supercomputer at NERSC. Using message sizes per compute node ranging from 64KBytes to 8MBytes, our experiments show significant performance improvements of up to 23.67 times on 256 compute nodes compared with the implementations of production MPI libraries.

1. Introduction

MPI [2] intergroup collective communication defines message transfer patterns between two disjoint groups of MPI processes, with communication taking place between groups but not inside groups. In contrast to the intragroup collective communication counterparts, where all processes in an Allgather communication receive the same messages from all other processes, intergroup collectives can have expressive and performance advantages for coupled scientific applications and workflows systems.

Intergroup collective communication patterns occur frequently in modern parallel frameworks for scientific applications. As discussed in [1], a scientific application workflow system consists of different groups of processes, where the results computed by one process is sent to another via intergroup All-to-All broadcast (Allgather) and/or All-to-All personalized communication. In [3], the authors proposed a parallel data transfer framework for computationally intensive weather prediction system, SCALE-LETKF [4]. Communications among workflow components are achieved using MPI intergroup Allgather. An advantage of using intergroup communication is achieving a higher degree of fault tolerance. Examples discussed in [5] include applications such as DNA sequencing, graphics rendering and searching for extraterrestrial intelligence that relies on a manager/worker model can benefit from the encapsulation provided by inter-communicators. Researchers have designed sophisticated approaches that reduce the data size communicated among workflow components and in turn improving the communication performance. For example, Zhang et al. [6] have proposed a distributed framework for maximizing on-chip data exchange in order to reduce the amount of communication among workflow components. Docan et al. [7] have designed an Active Spacing strategy that reduces the size of message transfers among workflow components by moving programs to staging areas. Existing literature that focuses on reducing the communication message size and frequency have successfully improved the communication performance between workflow components. Nevertheless, intergroup communications with large data size seem unavoidable, and applications can benefit from optimal MPI implementations of the intergroup collective communication.

In this paper, we focus on intergroup All-to-All broadcast communication, which in MPI corresponds to the `MPI_Allgather` and `MPI_Allgatherv` functions when using MPI inter-communicators. With an inter-communicator as an input argument, `MPI_Allgather` and `MPI_Allgatherv` exchange data between the two disjoint process groups, such that every process in one group receives messages from all processes in the other group. MPICH [8], MVAPICH [9], and OpenMPI [10], the three

☆ **Conflict of interest.** None.
☆☆ The paper is a revised, extended version of the paper "Full-Duplex Inter-Group All-to-All Broadcast Algorithms with Optimal Bandwidth" presented at the 25th European MPI Users' Group Meeting (EuroMPI), Barcelona, September 2018 [1].
* Corresponding author.
E-mail address: qiao.kang@eecs.northwestern.edu (Q. Kang).

https://doi.org/10.1016/j.parco.2019.04.015
0167-8191/© 2019 Elsevier B.V. All rights reserved.

most widely-used MPI implementations in the parallel processing community, implement these intergroup collectives (as well as most other intergroup collectives) based on the "root gathering" algorithm paradigm. The underlying principle of the root gathering algorithm is single-process accumulation per group followed by a pair-wise exchange of the gathered messages followed by one-to-all broadcasts within the two groups. However, this approach does not achieve optimal communication time under the single-port communication constraint, because the complete data to be exchanged are sent multiple times.

In our previous work [1], we presented an algorithm for `MPI_Allgather` that outperforms the root gathering algorithm by a large factor and achieves optimal communication transfer time under reasonable system (and message size) assumptions. The algorithm divides messages into segments in order to fully utilize the available communication links between the two groups (as well as within each of the groups), achieving optimal transfer time.

In this paper, we further improve the formulation of the algorithm proposed for `MPI_Allgather` in [1] by combining the two-stage intragroup Allgather communication into a single stage with the same communication time. In addition, the implementation strategy for handling the case of indivisible process numbers is described in detail. Moreover, we extend the message segmentation concept to design an algorithm for intergroup `MPI_Allgatherv`. The extension generalizes the solution for balancing the message size exchanged between two groups for intergroup `MPI_Allgather`. We show that the proposed algorithm for intergroup `MPI_Allgatherv` significantly improves the end-to-end execution time compared with the root gathering algorithm for larger message sizes.

We implement the proposed algorithms for intergroup `MPI_Allgather` and `MPI_Allgatherv` using MPI point-to-point and collective communication functions, namely intergroup send/receive, intragroup `MPI_Allgather`, and intragroup `MPI_Allreduce`. We conduct experiments on Cori, a Cray XC40 supercomputer at the National Energy Research Scientific Computing Center (NERSC). Direct comparisons with the MPI native library function `MPI_Allgather` that implements the root gathering algorithm are presented. Although the communication network topology on Cori is pseudo-fully connected (dragon-fly), we show that our ideas are applicable to hierarchically organized processes by evaluating and comparing our algorithms against the MPI library installed on Cori with different process settings. With message sizes per compute node ranging from 64KBytes to 8MBytes per node, the experiments show a significant performance improvement achieved by our algorithms, which are up to 23.67 times on 256 compute nodes faster than the production MPI libraries.

2. Background

For design and analysis, we employ the single-port communication model as defined in [11]. A set of *ranked processes* $\{0, 1, \ldots, p-1\}$ can communicate as specified by a communication graph representing the communication network. Processes are nodes in the graph, and edges represent communication links, such that pair-wise processes connected by an edge can communicate directly. The basic communication operations are sends and receives initiated by the processes. When a send or receive operation is called by a process, the function is blocked until the communication has been completed. Therefore, a process cannot send to multiple processes simultaneously. Likewise, a process cannot receive from multiple processes at the same time.

Communication cost is determined by a startup time term t_s and a per-byte transfer time term t_w. If there is a link between a sender x and a receiver y, sending a message of size k bytes from x to y has a communication transmission time of $t_s + kt_w$ time units. We refer to the t_s term as the *startup time* and kt_w as the *transfer time*. In this model, the objective in designing collective communication algorithms is to minimize communication transfer time. Modern networks are typically bidirectional, and a process can be involved in both a (non-blocking) receive and send operation at a time. Our algorithms exploit this when exchanging messages between processes in the two groups. Modern networks may be able to sustain more communication operations at a time. Our algorithms can also exploit such capabilities, but we leave this optimization as an implementation detail and use MPI non-blocking send and receive operations in our concrete implementations.

We assume that there are communication links between all processes in the two groups. This assumption is strong, and may not hold for all applications of the MPI intergroup collectives. For instance, coupled applications running on different parts of a heterogeneous system. There may be only weakly connected with only a few communication edges between the two process groups.

2.1. Problem definition

Let $A = \{a_0, \ldots, a_{p-1}\}$ and $B = \{b_0, .., b_{q-1}\}$ be two disjoint groups of processes. Group A has size p and group B has size q. Initially, every $a_i \in A$ has a unique message $m_{A,i}$ of size k_A bytes and every $b_j \in B$ has a unique message $m_{B,j}$ of size k_B bytes. We term $m_A = \{m_{A,i} | \forall 0 \le i < p\}$ and $m_B = \{m_{B,i} | \forall 0 \le i < q\}$ the *full messages* of the two groups. The goal of the All-to-All broadcast (Allgather) operation is to let every $b_j \forall 0 \le j < q$ receive $m_A = m_{A,i}$ $\forall 0 \le i \le p-1$ and every $a_j \forall 0 \le j < p$ receive $m_B = m_{B,i}$ $\forall 0 \le i \le q-1$, that is, to exchange the full messages between all processes in the two groups.

We refer to k_A and k_B as the *block sizes*, and $pk_A + qk_B$ as the (total) *problem size*. In the following, we will assume without loss of generality that $q \le p$.

2.2. Optimal transfer time

Consider a single process $a \in A$, which has to receive all messages from group B in the end. Thus, process a has to receive qk_B bytes. By the single-port constraint, it takes at least qk_Bt_w time units for a to receive all messages. Hence the lower bound on the transfer time is qk_Bt_w. The same argument can be applied to each process $b \in B$, giving a lower bound on the message transfer time of pk_At_w time units. By the argument, it follows that

$$\mathrm{M} = \max(qk_B, pk_A)t_w$$

is a lower bound on the transfer time for intergroup Allgather. We prove that this lower bound is the largest asymptotic lower bound by proposing an algorithm with transfer time converging to it given large p and q.

3. Related work

In MPI, the collective communication operations come in two flavors: Intragroup and intergroup operations [2, Chapter 5]. MPI distinguishes between the two cases by the communicator argument, which can be either an intra-communicator or an inter-communicator. The actual interfaces are the same in the two cases, but the underlying communication patterns are different. Intercommunicators in MPI can be viewed as objects that contain two intra-communicators representing the two disjoint process groups. In our implementations, we can reuse the efficient MPI intragroup collectives for communication inside the two process groups.

Intragroup collective communication algorithms have been thoroughly studied in existing literature [12]. Johnson et al. [13] have implemented the Allgather function under single-port communication constraint on hypercube topology. The algorithm of Bruck et al. [14] addresses the case of non-power of two number of processes for Allgather and All-to-All. Thakur et al. [15] optimized intragroup Allgather for MPI using recursive doubling and the Bruck algorithm [14] for non-power number of processes. Träff [16] has proposed an algorithm for intragroup Allgatherv using a ring topology that is efficient for larger data sets. Like the root gathering algorithm, our new algorithms benefit from efficient implementations of these algorithms for intragroup collectives, in particular, implementations that adapt to the hierarchical structure of modern, parallel systems.

Intergroup collective communication, on the other hand, is still in need of further research. For intergroup all-to-all broadcast (`MPI_Allgather`), MPICH [8], MVAPICH [9] and OpenMPI [10], the most widely-used MPI implementations in the HPC community, adopt the "root gathering algorithm" which does, by a significant factor, not achieve optimal transfer time. Also closed source, vendor libraries like Intel MPI seem to use some variant of the root gathering paradigm.

3.1. The root gathering algorithm

The *root gathering algorithm*, adopted by popular MPI production libraries such as MPICH, MVAPICH, and OpenMPI, is discussed in [17]. To the best of our knowledge, this algorithm is the state-of-art implementation of MPI inter-group Allgather. For each of the two groups, a root process gathers all messages from other processes in the local group and broadcasts the gathered message to all processes in the remote group. The root gathering algorithm can be readily implemented using existing MPI functionality without creating any intermediate communicators. However, it does not meet the lower bound on communication transfer time under the single-port communication constraint by a large factor.

The best version of the root gathering algorithm has three stages. Let a_0 and b_0 be chosen root processes in groups A and B. In the first stage, root process a_0 gathers the full message m_A from the other $p-1$ processes in group A. Root process b_0 gathers the full message m_B from the other $q-1$ processes in group B. The overall communication cost of this stage is

$$\max\left(\log(p)t_s + (p-1)k_A t_w, \log(q)t_s + (q-1)k_B t_w\right)$$

using standard, optimal, binomial-tree algorithms, see, e.g. [12]. In the second stage, the two roots exchange their aggregated messages, which takes

$$\max\left(t_s + pk_A t_w, t_s + qk_B t_w\right)$$

time units. In the third stage, root process a_0 broadcasts m_B of size qk_B to the other $p-1$ processes in its group. Root process b_0 broadcasts m_A of size pk_A to the other $q-1$ processes within its group. This stage takes time

$$\max\left(\log(p)t_s + qk_B t_w, \log(q)t_s + pk_A t_w\right)$$

(plus lower order terms) using an optimal algorithm for the broadcast operations [18,19].

Let T be total the communication cost of the three stages. A lower bound for the communication cost of the root gathering algorithm can be bounded as expressed by Eq. (1), with $\alpha = (1 + 2\log(p))$.

$$\begin{aligned}
\mathrm{T} &\le \alpha t_s + (\max((p-1)k_A, (q-1)k_B))t_w + 2\mathrm{M} \\
&= \alpha t_s + \max((3p-1)k_A, (3q-1)k_B)t_w \\
&< 2\log(p)t_s + 3\mathrm{M} \qquad \text{as } p, q \to \infty
\end{aligned} \tag{1}$$

The time of this version of the root gathering algorithm is about a factor three larger than the lower bound.

MPI libraries like MPICH, MVAPICH, and OpenMPI do not implement optimal algorithms for broadcast, but algorithms that are a factor of two off in the transfer time term [15]. Furthermore, in their implementations, the intergroup exchange of gathered messages at Stage 2 and the one-to-all broadcasts at Stage 3 are all performed sequentially. Thus, the implementation of the root gathering algorithm in MPI production libraries has communication cost T roughly as shown in Eq. (2), where $\beta = (2 + 2\log(p) + \log(q))$ This cost is a baseline for evaluating our new algorithm.

$$\begin{aligned}
\mathrm{T} &\le \beta t_s + \max((p-1)k_A, (q-1)k_B) + 3pk_A + 3qk_B)t_w \\
&\le 3\log(p)t_s + \mathrm{M} + 3(pk_A + qk_B)t_w \qquad \text{as } p, q \to \infty
\end{aligned} \tag{2}$$

4. The Allgather Algorithm

We present our new algorithm for the intergroup Allgather operation that achieves the transfer time lower bound asymptotically. In [1], we have presented an algorithm for intergroup Allgather with optimal transfer time for large numbers of processes. The algorithm for intergroup Allgather proposed in this paper simplifies the previous work by reducing the two-stage intragroup Allgather operations to one stage. We show that the transfer time of the proposed algorithm is also optimal for large numbers of processes.

Overall, the idea of the proposed algorithm is to exploit as many communication links between the two groups concurrently as possible in order to exchange the full messages m_A and m_B between the two groups. After that, concurrent intragroup Allgather operations in each of the groups collect the received messages to achieve the objective of intergroup Allgather. Our algorithm keeps q pairs of processes active throughout the exchange (recall that we assumed that $q \le p$), and the exchange is done over $\lceil \frac{p}{q} \rceil$ communication rounds. We divide the messages from the smaller group B into segments such that the total number of segments in group B is p. From that, the correctness of the algorithm is clear by construction: Each process in group A will receive a segment from a process in group B, and each process in group B will receive one or more blocks from processes in group A. All blocks from group A will have been transferred to processes in group B, and all segments from group B will likewise have been transferred to processes group A. The concurrent Allgather operations in the two groups collect the blocks and the segments together, such that each process will have the full message from the processes of the other group. The partner processes in the exchange are chosen such that blocks and segments are received in rank order over the processes, and an Allgather operation without any block or segment reordering therefore suffices. The same idea can also be applied to the `MPI_Allgatherv` operation as described in Section 4.1.

Algorithm 1: Full-duplex Intergroup All-to-All Broadcast with Message segmentation.

```
1  for $i \in \left[0, .., \frac{p}{q} - 1\right]$ do
2      # Concurrent loop $j$
3      for $j \in [0, .., q-1]$ do
4          $a_{\frac{p}{q}j+i}$ send $m_{A,\frac{p}{q}j+i}$ to $b_j$
5          $b_j$ send $m_{B,j,i}$ to $a_{\frac{p}{q}j+i}$
6      end
7  end
8  # Group $A$ Allgather received messages from group $B$.
9  Intra-Allgather $a_j, m_{B,j,\lfloor \frac{j}{q} \rfloor} \forall\{j : 0 \le j < p\}$
10 # Group $B$ Allgather received messages from group $A$
11 Intra-Allgather $b_j, m_{A,qi+j} \forall(0 \le j < q, 0 \le i < \frac{p}{q})$
```

Algorithm 1 formally shows the proposed intergroup Allgather operation. It consists of a message exchange step between the two

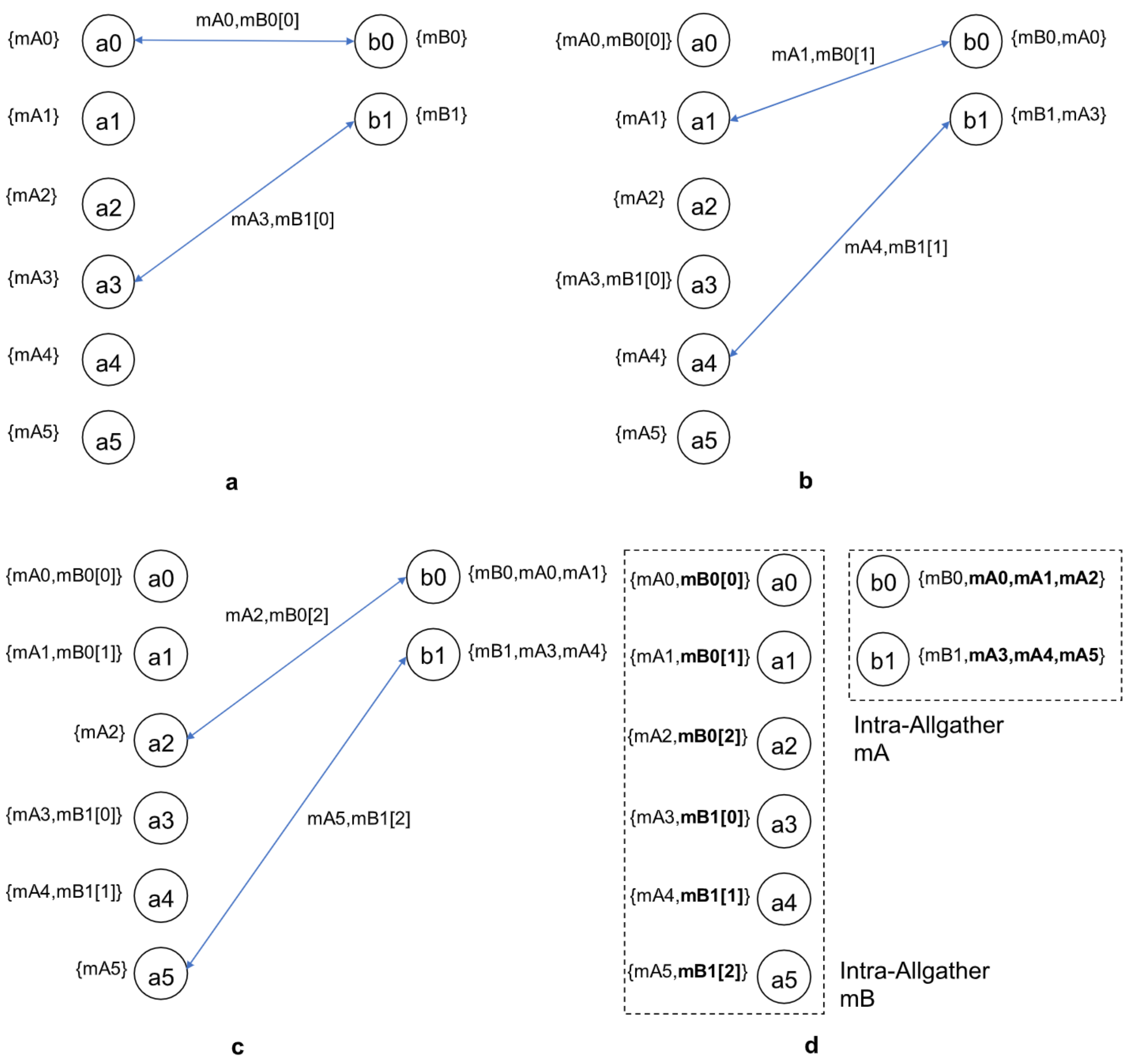


Fig. 1. Example execution of Algorithm 1 for $p = 6$ and $q = 2$.

groups of processes, followed by concurrent intragroup Allgather operations within each of the groups. We first assume that q divides p. Each process i in group A sends its message $m_{A,i}$ to process $j = \lfloor i/\frac{p}{q} \rfloor$, as shown in Line 4. The messages in group B are segmented and sent across the groups in $\frac{p}{q}$ communication rounds, as shown in Line 5. For now, we also assume that k_B is divisible by $\frac{p}{q}$. Each process j in group B sends a part of its message $m_{B,j}$ of size $k_B/\frac{p}{q}$ to process $j\frac{p}{q} + i$ for $i = 0, \dots, \frac{p}{q} - 1$. After this exchange in $\frac{p}{q}$ communication rounds, it is obvious that all messages from group A have arrived at processes in group B, and that all segments of the messages in group B are likewise present at processes in group A. To complete the All-to-All broadcast operation, each of the two groups performs intragroup Allgather operations over messages received from the remote group. These two Allgather operations can take place concurrently. The intragroup Allgather operations of group A for messages received from group B is in Line 9. The intragroup Allgather operations of group B for messages received from group A is in Line 11. Fig. 1 shows an execution of the algorithm with $p = 6$ and $q = 2$.

To handle the case where $\frac{p}{q}$ is not an integer, we divide processes in group A into q disjoint subgroups $\{s_0, .., s_{q-1}\}$ such that all ranks in s_i are smaller than ranks in s_j if $i < j$. We assign the size of the subgroups in the following way. Let $r = p \bmod q$. For $i < r$, s_i has size $\lceil \frac{p}{q} \rceil$. Otherwise, s_i has size $\lfloor \frac{p}{q} \rfloor$. Since $p = \lceil \frac{p}{q} \rceil r + \lfloor \frac{p}{q} \rfloor (q - r)$, the union of the subgroups is A.

We modify Algorithm 1 in the following way. Every $b_i \in B$ exchanges messages with all processes in the subgroup s_i. Then, both group A and group B perform intragroup Allgather operations for messages received from their remote groups. Since every subgroup of group A has almost the same size, the segmented message received by the processes in group A also has almost the same size. Likewise, every process in group B also has similar size of messages received from group A. With the balanced send sizes, the final intragroup Allgather operations can be performed using the largest send size of any process without having significant completion time overhead, or by an Allgatherv operation. Fig. 2 gives an example with $p = 8$ and $q = 3$. We have $s_0 = \{0, 1, 2\}$, $s_1 = \{3, 4, 5\}$, and $s_2 = \{6, 7\}$. The final intragroup Allgather for group B has a send size of $3k_A$. The final intragroup Allgather for group B has a send size of $\frac{k_B}{2}$.

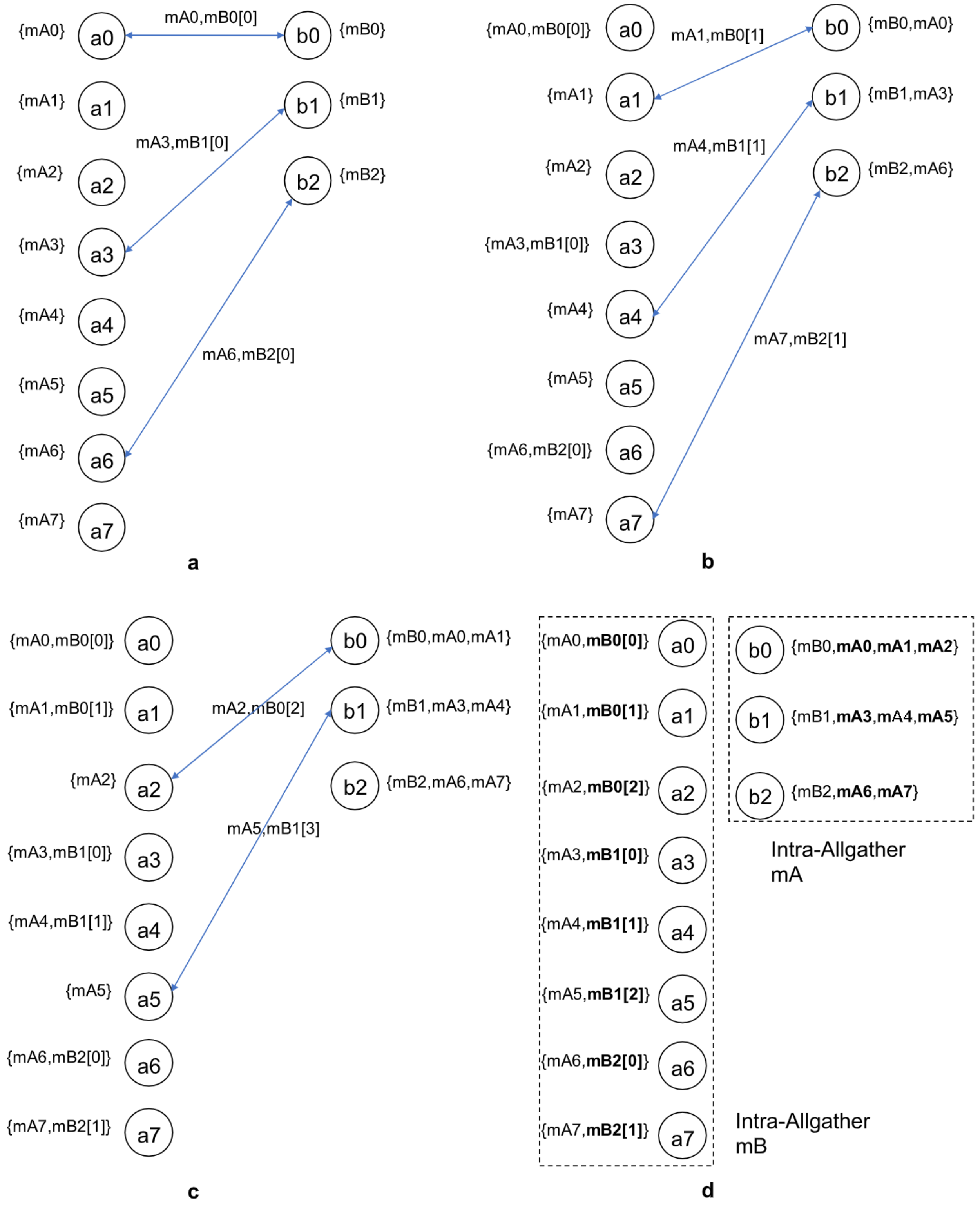


Fig. 2. Example execution Algorithm 1 for $p = 8$ and $q = 3$.

If the message size k_B of group B is not divisible by $\frac{p}{q}$, some processes from B will send a smaller, potentially empty message to some processes in group A. The algorithm is still correct in the sense that all segmented messages from group B are present in group A after the exchange. Again, the Allgather step is done either by using the larger send size, or by an Allgatherv operation.

The segmentation for messages transferred from group B to group A is necessary, especially when $\frac{p}{q}$ is large. In [1], we analyzed the transfer time of an algorithm similar to Algorithm 1, but without message segmentation. We showed that the transfer time of this algorithm is not bounded by any multiple of the optimal transfer time lower bound when $pk_A < qk_B$ as $\frac{p}{q} \to \infty$.

The message segmentation algorithm proposed in [1] differs from Algorithm 1 in the way that processes in group A gather all messages from group B via two steps of intragroup Allgather operations. With the homogeneous communication model cost assumed in this paper, the two-step intergroup Allgather operation is unnecessary, since the one-step version shown in Algorithm 1 achieves the task with the same transfer time. However, the two-step implementation of [1] could have advantages on hierarchical systems with non-homogeneous costs in case the `MPI_Allgather` operation is not properly aware of the processor hierarchy. For instance, one of the Allgather operations can operate entirely inside processor nodes. We prefer the formulation of Algorithm 1 and assume that the `MPI_Allgather` operation is properly implemented to be efficient over any assignment of MPI processes to processors in the compute nodes.

In Algorithm 1, Lines 1–7 initiate $\frac{p}{q}$ operations, so the startup time is $\frac{p}{q}t_s$. For Line 9, the Allgather can be finished in $\log(p)$ communication rounds using the Bruck algorithm [14]. For Line 11, the number of communication rounds is $\log(q)$. Therefore, the overall startup time for Lines 9–11 is $\log(p)t_s$ since $q \le p$ by assumption.

Theorem 1. *The transfer time term of Algorithm 1 is bounded by* $(\max(pk_A, qk_B) + k_B)t_w = \mathrm{M} + k_B t_w$.

Proof. The total transfer time of Lines 1–7 is

$$\max\left(k_A\frac{p}{q}, \frac{k_B}{\frac{p}{q}}\frac{p}{q}\right)t_w = \max\left(k_A\frac{p}{q}, k_B\right)t_w.$$

Line 9 has a total transfer time of $\left(q - \frac{q}{p}\right)k_B t_w$, since it is an intragroup Allgather operation of $\frac{qk_B}{p}$ message size over p processes. Line 11 has transfer time $\left(p - \frac{p}{q}\right)k_A t_w$. The intragroup Allgather at Line 11 runs parallel to Line 9. It follows that the overall transfer time can be bounded using Eq. (3).

$$\begin{aligned}
&\left(\max\left(k_A\frac{p}{q}, k_B\right) + \max\left(k_A\left(p - \frac{p}{q}\right), k_B\left(q - \frac{q}{p}\right)\right)\right)t_w \\
&\le \left(\max\left(k_A\frac{p}{q}, k_B\right) + \max\left(k_A\left(p - \frac{p}{q}\right), k_B q\right)\right)t_w \\
&\le \left(\max\left(k_A\frac{p}{q}, k_B\right) + \max\left(k_A\frac{p}{q}, k_B\right)(q-1) + k_B\right)t_w \qquad (3)\\
&= (\max(pk_A, qk_B) + k_B)t_w \\
&= \mathrm{M} + k_B t_w \quad \square
\end{aligned}$$

Theorem 1 gives a lower bound for the transfer time of Algorithm 1. This lower bound justifies the optimality of transfer time for Algorithm 1 asymptotically with respect to p and q.

We summarize the correctness, namely that the full messages m_A and m_B are gathered at all processes in groups B and A as required in Theorem 2.

Theorem 2. *Algorithm 1 correctly implements the intergroup Allgather operation.*

Theorem 2 follows from the description of Algorithm 1. In Fig. 1, it is clear that messages transferred from group A to group B do not overlap and have a total size of pk_A. Likewise, messages transferred from group B to group A are also disjoint and have a total size of qk_B. Since all messages received from the other group are contributed in each of the Allgather operations, Algorithm 1 correctly implements the intergroup Allgather.

Algorithm 1 does not utilize all links possible between the $p+q$ processes. We note that optimal transfer time requires much fewer links than the $(p+q)(p+q-1)$ links of a fully connected system. The intragroup Allgather only requires a ring topology to reach optimal completion time [12,16]. For Algorithm 1, Lines 1–7 require p links, since every process in group B sends messages to $\frac{p}{q}$ processes in group A and every process in group A sends messages to only one process in group B. Line 9 requires p processes connected in a ring to perform the Allgather. Likewise, Line 11 requires q processes connected in a ring to perform the Allgather. Therefore, the minimum number of links to maintain optimal transfer time is less than $p + q + \frac{p}{q}$ and linear in the number of processes.

4.1. The Allgatherv Algorithm

The `MPI_Allgatherv` operation generalizes the `MPI_Allgather` collective. Instead of every process in each group having the same block size, `MPI_Allgatherv` allows every process to contribute a block of different size. Therefore, a process receives messages of different size from every process in its remote group.

Let $k_{A,i}$ be the message size of $m_{A,i}$ sent by process a_i and $k_{B,j}$ be the message size of $m_{B,j}$ sent by process b_j. The goal of intergroup MPI_Allgatherv is to let every $b_j\, \forall 0 \le j < q$ receive $m_A = m_{A,i}\ \forall 0 \le i \le p-1$ and every $a_j\, \forall 0 \le j < p$ receive $m_B = m_{B,i}\ \forall 0 \le i \le q-1$. By the same argument as for Allgather in Section 2.2, it follows that a lower bound for the optimal transfer time is

$$\mathrm{M} = \max\left(\sum_{i=0}^{p-1} k_{A,i}, \sum_{i=0}^{q-1} k_{B,i}\right)t_w.$$

The algorithm for the `MPI_Allgatherv` follows the same principle as for `MPI_Allgather`. The processes in each group compute the total message sizes $K_A = \sum_{i=0}^{p-1} k_{A,i}$ and $K_B = \sum_{i=0}^{q-1} k_{B,i}$, respectively, which can be done by an Allreduce operation concurrently within the two groups. The messages to be sent from group A to group B are then segmented into blocks of size roughly $\frac{K_A}{q}$, and the blocks from group B to group A into blocks of size roughly $\frac{K_B}{p}$. Thus, processes having large blocks will send these over a number of send operations, while small blocks will be sent in a single operation. The overall time for the exchange step will thus be determined by the largest blocks, that is $\max(k_{A,i}, k_{B,j})$. This exchange step ensures that all segments from group B will be present at processes in group A in segments of roughly the same size, and that all segments from processes in group A will likewise be present in group B, such that final Allgather operations can collect the full messages at all processes in the two groups, as required. For process a_i in group A, to compute which processes in group B it has to send its segments, it only has to compute the size of blocks at the processes $a_0, a_1, \ldots, a_{i-1}$ with a lower rank than a_i. Call this prefix sum which can be computed by an Exscan operation for $R_{A,i}$. The first process in group B to receive a segment from a_i is then $R_{A,i}/\frac{K_A}{q}$. The same applies to process b_j in B. This ensures that all segments will be present in rank order in each of the groups, such that an Allgather operation can indeed complete the algorithm without any need for reordering. The cases where block size is not exactly divisible can be handled as described for Algorithm 1. It is important to notice that the total sizes of segments received by a process is almost the same for all processes in the group, such that an Allgather operation with same block sizes can be used.

Algorithm 2 describes our algorithm for intergroup Allgatherv more formally. To make the segmentation easier to describe, we assign a unique, global index for every byte of the messages in each of the groups, ordered based on the rank of processes. Formally, the global index of byte y in a message $m_{A,x}$ is denoted by $m_{A,x,y}$ and defined as $m_{A,x,y}.index = \sum_{i=0}^{x-1} k_{A,i} + y$. We define the set of indices associated with message $m_{A,x}$ as $m_{A,x}.index = \{\sum_{i=0}^{x-1} k_{A,i} + y \forall 0 \le y < k_{A,x}\}$. For example, if $m_{A,0}$ has a size of x bytes and $m_{A,1}$ has a size of y bytes, $m_{A,0,i}.index = i$ and $m_{A,1,j}.index = x + j$ for $0 \le i < x$ and $0 \le j < y$. $m_{A,0}.index = [0, x)$ and $m_{A,1}.index = [x, x+y)$. The same definition applies to m_B.

Algorithm 2: Full-duplex Intergroup Allgatherv.

```
1  $k'_A \leftarrow$ Intra-group Allreduce (Sum) $a_j \forall (0 \le j < p)$ over $k_{A,j}$
2  $k'_B \leftarrow$ Intra-group Allreduce (Sum) $b_j \forall (0 \le j < q)$ over $k_{B,j}$
3  $r_A \leftarrow k_A \bmod q$
4  $r_B \leftarrow k_B \bmod p$
5  for $i \in [0, q-1]$ do
6      if $i < r_A$ then
7          $s_{A,i} \leftarrow \{j : j \in [\lceil \frac{k'_A}{q} \rceil, (i+1)\lceil \frac{k'_A}{q} \rceil)\}$
8      else
9          $s_{A,i} \leftarrow \{j : j \in$
           $[r_A \lceil \frac{k'_A}{q} \rceil + (i - r_A)\lfloor \frac{k'_A}{q} \rfloor, r_A \lceil \frac{k'_A}{q} \rceil + (i+1-r_A)\lfloor \frac{k'_A}{q} \rfloor)\}$
10     end
11 end
12 for $i \in [0, p-1]$ do
13     $a_i$ send $m_{A,i,j}$ to $b_k \forall j$ such that $m_{A,i,j}.index \in s_{A,k} \forall 0 \le k <$
       $q \wedge m_{A,i}.index \cap s_{A,k} \neq \phi$
14 end
15 for $i \in [0, p-1]$ do
16     if $i < r_B$ then
17         $s_{B,i} \leftarrow \{j : j \in [i\lceil \frac{k'_B}{p} \rceil, (i+1)\lceil \frac{k'_B}{p} \rceil)\}$
18     else
19         $s_{B,i} \leftarrow \{j : j \in$
           $[r_B \lceil \frac{k'_B}{p} \rceil + (i - r_B)\lfloor \frac{k'_B}{p} \rfloor, r_B \lceil \frac{k'_B}{p} \rceil + (i+1-r_B)\lfloor \frac{k'_B}{p} \rfloor)\}$
20     end
21 end
22 for $i \in [0, q-1]$ do
23     $b_i$ send $m_{B,i,j}$ to $b_k \forall j$ such that $m_{B,i,j}.index \in s_{B,k} \forall 0 \le k <$
       $p \wedge m_{B,i}.index \cap s_{B,k} \neq \phi$
24 end
25 Intra-Allgather $a_j \forall (0 \le j < p)$ over $m_B$
26 Intra-Allgather $b_j \forall (0 \le j < q)$ over $m_A$
```

Initially, each of the groups sums up the total message size of their local groups in Lines 1–2. In Lines 5–24, both groups exchange their messages. Lines 5–14 and Lines 15–24 are similar. The first one is from group A to group B and the other is from group B to group A. The definition of $s_{A,i}$ at Lines 7 and 9 ensure that the message received by every process in group A at Line 13 is unique and has size either $\lfloor \frac{k'_A}{q} \rfloor$ or $\lceil \frac{k'_A}{q} \rceil$. Similarly, the definition of $s_{A,i}$ at Lines 17 and 19 ensure that every process in group B receives unique messages of size either $\lfloor \frac{k'_B}{p} \rfloor$ or $\lceil \frac{k'_B}{p} \rceil$ at Line 23. Fig. 3 gives an example of this stage. The balancing of the message sizes received by receivers at this stage is necessary for the `MPI_Allgather` operations in the next stage. Lines 25 and 26 perform intragroup Allgather for messages received from the remote group of each group.

At Lines 13 and 23 of Algorithm 2, the send operations are run in parallel. However, by the single-port communication constraint, a sender can send to one receiver and a receiver can receive from one sender at a time. Therefore, there may be delays of send/receive operations that can compromise the transfer time of the algorithm at Line 13 and 23. For example, in Fig. 3a, a_3 sends messages to b_1 and b_2. b_2 receives messages from a_3 and a_4. To achieve the best possible transfer time with the algorithm, it is crucial that sending and receiving is done in such a way that sending processes can be busy throughout.

We order the send/receive operations in the following way. If a sender sends messages to multiple receivers, the sender always sends its messages in the order of its receivers' ranks. We call the senders that send to multiple receivers multi-target senders. A receiver decides the order of senders it receives messages from and senders should wait for the receiving channel of the receiver to be idle in order to initiate the send operation. It is obvious that one receiver has at most two multi-target senders by the design of Algorithm 2. The receivers receive messages from the multi-target sender with higher rank, followed by senders with ranks between the two multi-target senders, followed by the multi-target sender with a lower rank. For example, in Fig. 3, both a_2 and a_3 try to send messages to b_1. In such case, the receiver receives messages from a_3 with higher priority, since a_3 is a multi-sender with the highest rank among all its senders. Another example is b_2, both a_3 and a_4 send messages to b_2. Moreover, both a_3 and a_4 are multi-target senders. Since a_4 has a higher rank, b_2 receives messages from a_4 with higher priority. Theorem 3 presents an upper bound for the transfer time of Algorithm 2.

Definition 1. A receiver receives messages from a set of senders compactly if the total transfer time is equal to the senders' total messages size times t_w.

Theorem 3. *The transfer time term of Algorithm 1 is bounded by*

$$\max\left(\sum_{i=0}^{p-1} k_{A,i} + \max(k_A), \sum_{i=0}^{q-1} k_{B,i} + \max(k_B)\right) t_w$$
$$+ \log(pq) t_w$$
$$\le \mathrm{M} + (\max(\max(k_A), \max(k_B)) + \log(pq)) t_w \quad (4)$$

Proof. The Allreduce operations at Lines 1–2 have a transfer time $\log(p) t_w + log(q) t_w$.

Consider a multi-target sender a_j, let $b_i, ., b_{i+y}$ for $y > 1$ be its receivers. Process b_i receives the message from a_j at the beginning, since a_j is the multi-target sender of b_{i+y} with highest rank by the design of $S_{A,i+y}$. The processes $b_{i+z} \forall 0 < z < y$ receive messages exclusively from a_j, so the send operations are also not delayed. Now we consider process b_{i+y}. Let $a_j, .., a_{j+c}$ be its senders. If a_{j+c} is a multi-target sender, b_{i+y} receives messages in order of $\{a_{j+c}, a_{j+1}, ., a_{j+c-1}, a_j\}$. The processes $\{a_{j+1}, .., a_{j+c}\}$ are senders that only send to b_{i+y} and b_{i+y} is the lowest rank among all receivers of process a_{j+c}. If process a_{j+c} is not a multi-target sender, b_{i+y} receives messages in order of $\{a_{j+1}, ., a_{j+c}, a_j\}$. Processes $\{a_{j+1}, ., a_{j+c}\}$ are senders that send messages to b_{i+y} exclusively. Therefore, b_{i+y} has been compactly receiving messages when it starts to receive messages from a_j in either of the cases. If the send operation from process a_j to process b_{i+y} is not delayed, the total transfer time of the send operations for a_j is $k_{A,\,j} t_w \le \max(k_A) t_w$. If the send operation from a_j to b_{i+y} is delayed, the total transfer time of the send operations for a_j is less than $\lceil \frac{\sum_{i=0}^{p-1} k_{A,i}}{q} \rceil t_w$, since a_j is the last sender of the receiver b_{i+y} and b_{i+y} can receive at most $\lceil \frac{\sum_{i=0}^{p-1} k_{A,i}}{q} \rceil \le \max(k_A)$ messages by design of $S_{A,i+y}$.

If a sender a_j is not a multi-target sender, it is obvious (from previous argument for b_{i+y}) that its only receiver b_i receives messages compactly until receiving messages from a_j. Since b_i can receive at most $\lceil \frac{\sum_{i=0}^{p-1} k_{A,i}}{q} \rceil \le \max(k_A)$ messages by design of $S_{A,i}$, the transfer time of the send operation for a_j is less than $\max(k_A) t_w$.

Line 25 is an Allgather operation of messages received from Lines 12–14 for processes in group B. The send size is $size_A = \lceil \frac{\sum_{i=0}^{p-1} k_{A,i}}{p} \rceil$. Therefore, the transfer time is $p(size_A - 1) t_w \le \sum_{i=0}^{p-1} k_{A,i} t_w$. Consequently, $(\max(k_A) + \sum_{i=0}^{p-1} k_{A,i}) t_w$ is the total transfer time of Lines 12–14 and 25.

We can apply the same argument to derive that $(\max(k_B) + \sum_{i=0}^{q-1} k_{B,i}) t_w$ is the total transfer time of 22–24 and 26. Lines 12–14 and 25 run in parallel with Lines 22–24 and 26. Thus, the optimal transfer time bound in the theorem is proven. □

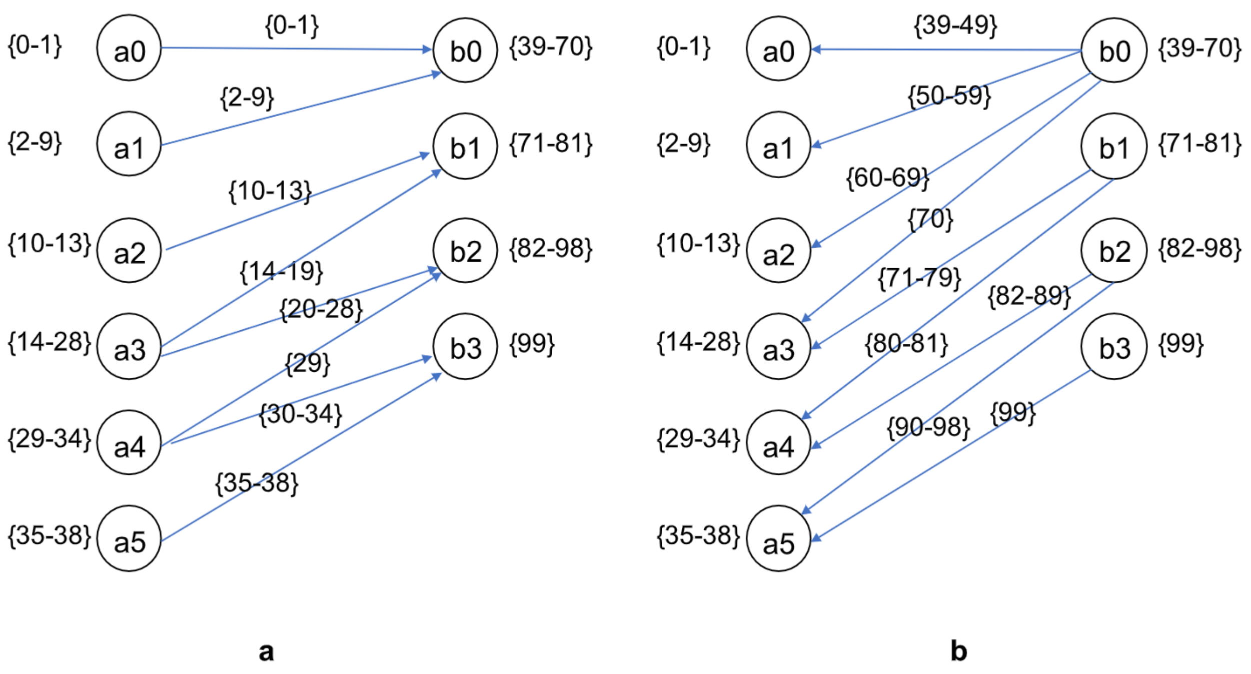

{0-1} a0 {0-1} b0 {39-70}
{2-9} a1 {2-9} b1 {71-81}
{10-13} a2 {10-13} b2 {82-98}
{14-28} a3 {14-19} {20-28} b3 {99}
{29-34} a4 {29} {30-34}
{35-38} a5 {35-38}
a
{0-1} a0 {39-49} b0 {39-70}
{2-9} a1 {50-59} b1 {71-81}
{10-13} a2 {60-69} b2 {82-98}
{14-28} a3 {70} {71-79} b3 {99}
{29-34} a4 {80-81} {82-89}
{35-38} a5 {90-98} {99}
b

Fig. 3. An illustration for the intergroup message exchange in Algorithm 2 for $p = 6$ and $q = 2$. (a) The send operations in Lines 12–14. (b) The send operations in line 22–24.

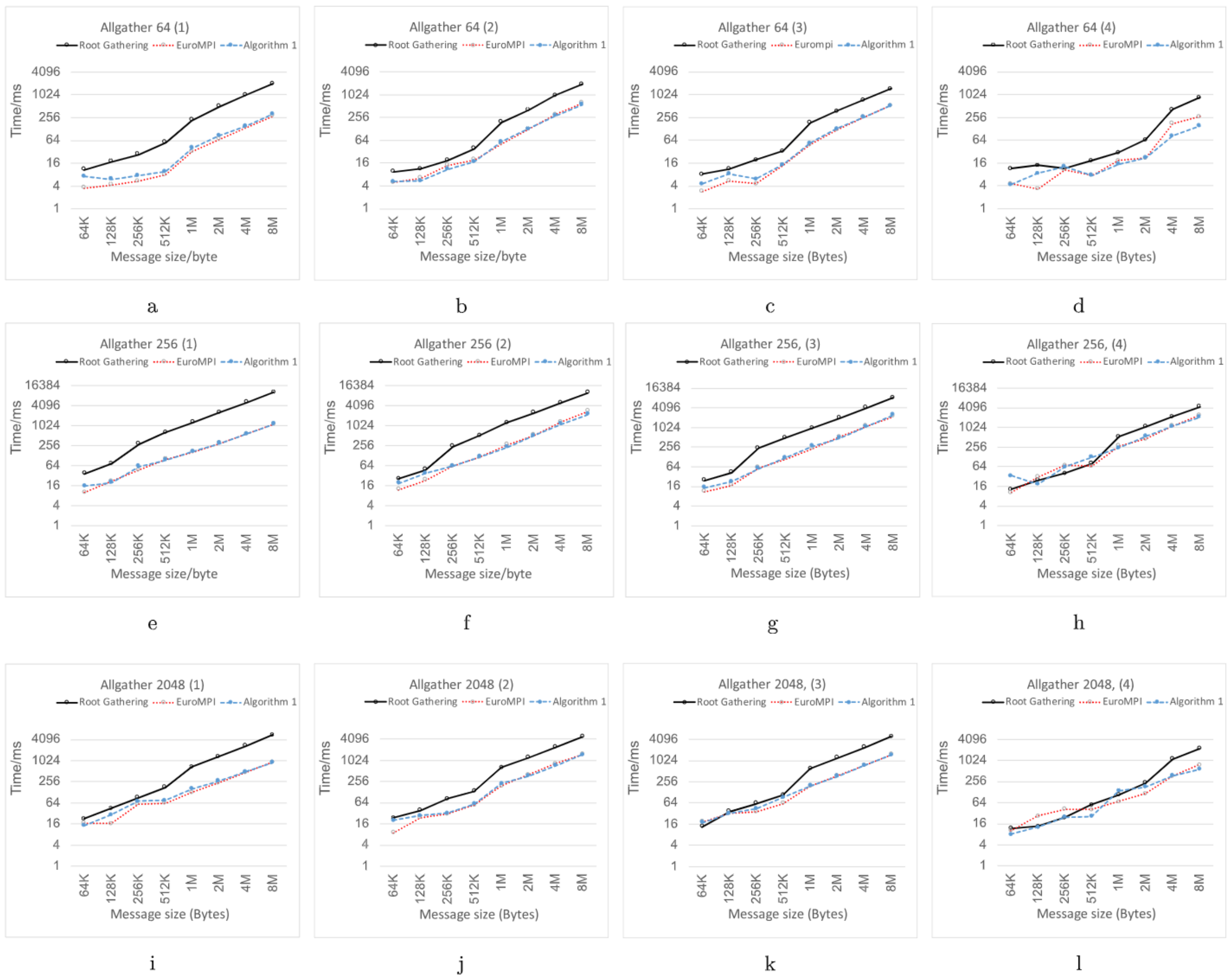


Fig. 4. Allgather results for settings 1–4. Legend “Algorithm1” refers to Algorithm 1 of this paper, legend “EuroMPI” to Algorithm 2 in [1], and legend “Root Gathering” to the MPI library native root gathering algorithm. The plots are labeled with titles of the form “Number of processes (setting number)”. Settings for p, q, k_A, and k_B have been defined in the text. The x-axis gives the maximum message size per node in Bytes, and the y-axis gives the completion time in milliseconds. The data type used is `MPI_INT`.

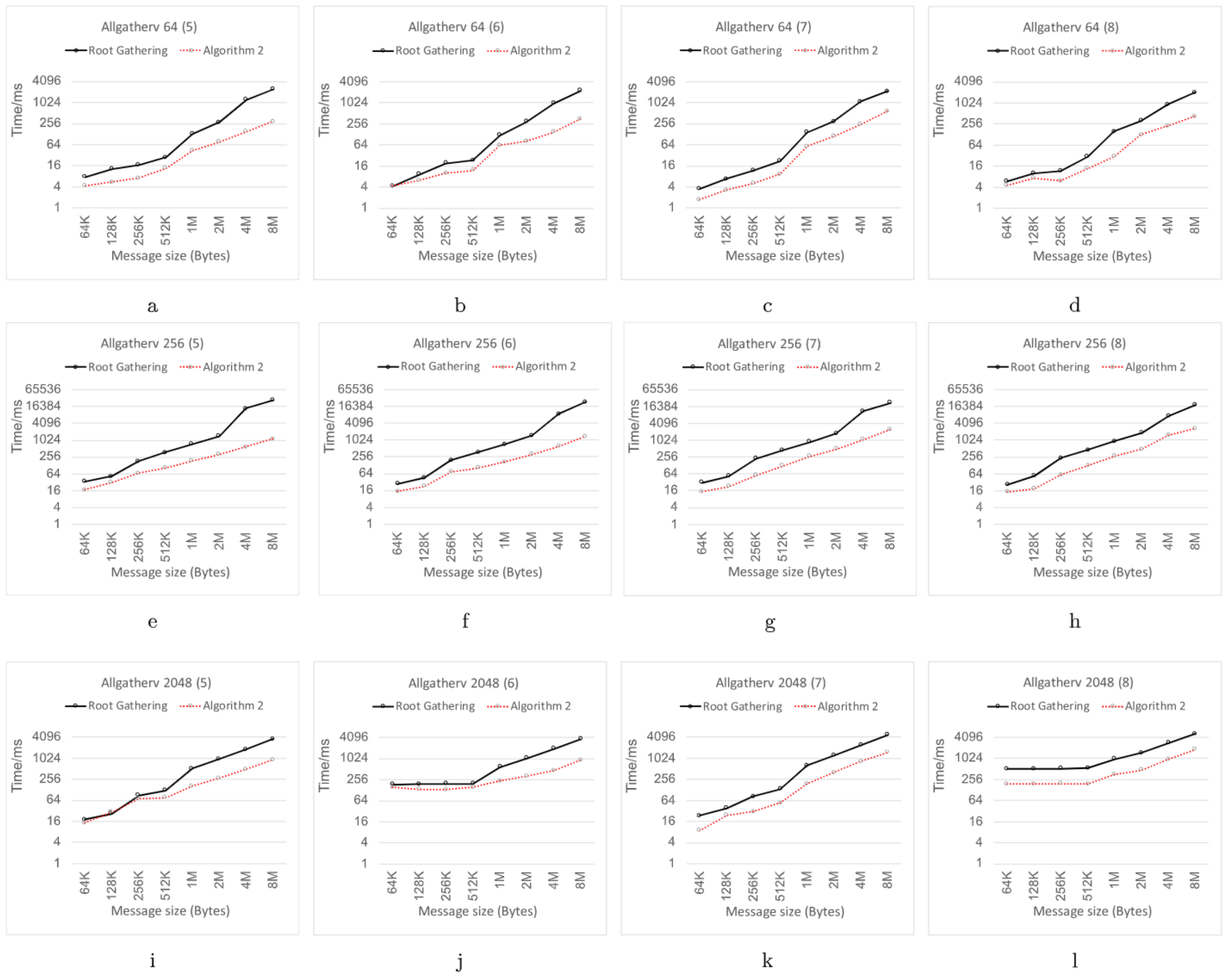


Fig. 5. Algorithm 1 versus the MPI library native root gathering algorithm for settings 5–8. The plots are labeled with titles of the form "Number of processes (setting number)". Settings for p, q, k_A, and k_B have been defined in the text. The x-axis gives the maximum message size per node in Bytes, and the y-axis gives the completion time in milliseconds. The data type used is `MPI_INT`.

Theorem 3 shows that the message transfer ordering achieves optimal transfer time asymptotically. The term $\log(pq)t_w$ is negligible compared with M for large message sizes and the term $\max(\max(k_A), \max(k_B))t_w$ is negligible for large p and q. Therefore, Algorithm 2, along with the ordering proposed for the send operations, converges to the lower bound of the optimal transfer time M.

Theorem 4. *Algorithm 2 correctly implements the intergroup Allgatherv operation.*

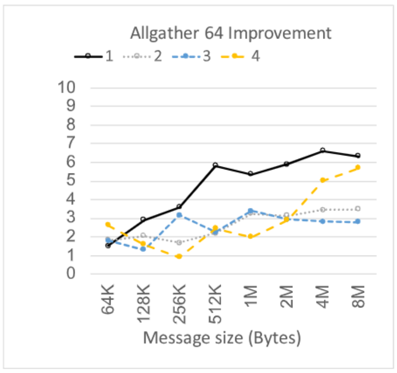
Theorem 4 recapitulates the correctness of Algorithm 2, and follows from the explanation given above.

5. Experimental results

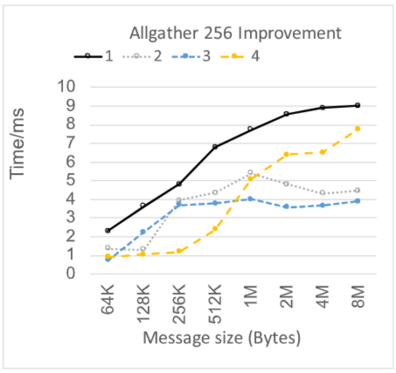
To study the actual improvements achieved over the intergroup collective implementations in standard MPI libraries, a number of experiments have been performed on Cori, a Cray XC40 supercomputer at the National Energy Research Scientific Computing Center (NERSC). We have used the Cray MPI compiler(Cray-mpich/7.6.0), which is based on MPICH [8]. We have implemented our algorithms as described in the previous sections, using intra-communicator `MPI_Allgather` and `MPI_Allgatherv` as building blocks. Creation of new communicators is not necessary since an inter-communicator gives access to the intra-communicator that a given process belongs to. The `MPI_Allgather` and `MPI_Allgatherv` interfaces give the information on block sizes as required by the algorithms to perform the message segmentation.

For our experiments, we fill all message buffers with random numbers. We measure the `MPI_Allgather` 8 times and report the average completion time. As completion time for each individual measurement, we pick the maximum time among all processes. We implement Algorithms 1 and 2 using MPI functions and make direct comparisons with the root gathering algorithm by calling the MPI library native `MPI_Allgather` and `MPI_Allgatherv` operations on the proper inter-communicators.

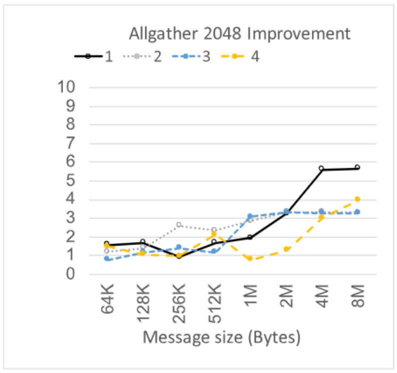
Algorithm 1 is equivalent to Algorithm 2 presented in [1] in terms of transfer time. Therefore, we expect the same performance. In this section, we present additional experimental results for cases where p and q are pairwise indivisible. We present four parameter settings, numbered from 1 to 4. Setting 1 is when $p = q$ and all processes are sending exactly the same data size. Setting 2 is when $\frac{p}{q} = \frac{25}{7}$ and all processes are sending exactly the same data size. This setting evaluates an imbalanced number of processes in the two groups. Moreover, the sizes of the two groups are pairwise indivisible. Setting 3 is when $\frac{p}{q} = \frac{25}{7}$ and group A sends larger messages than group B. Setting 4 is when $\frac{p}{q} = \frac{25}{7}$ and group B sends larger messages than group A. Settings 3 and 4 evaluate imbalanced message sizes for the two groups.



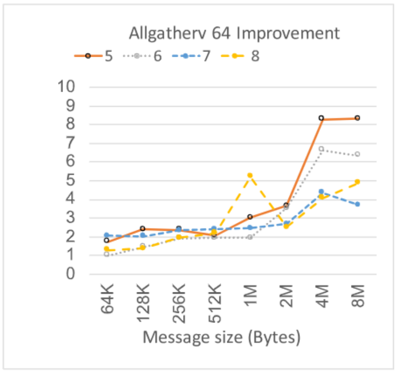
a



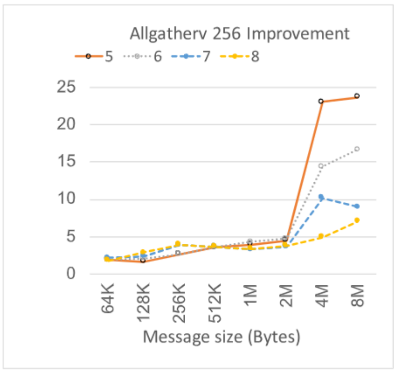
b



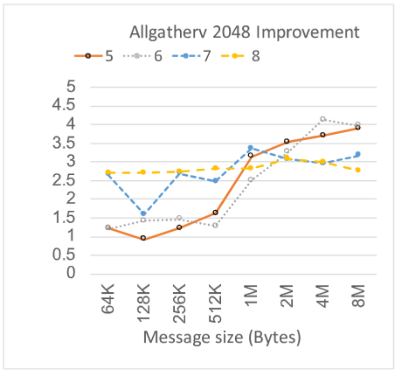
c



d



e



f

Fig. 6. Improvement of the proposed algorithms (Algorithm 1 for Allgather and Algorithm 2 for Allgatherv) (T_{new}) compared with the root gathering algorithm adopted by the MPI library (T_{old}). Improvements are computed as $\frac{T_{\text{old}}}{T_{\text{new}}}$. Each curve represents the improvement of one of the settings with different message sizes. The number in titles refers to the total number of processes.

We use four parameter settings, numbered from 5 to 8, on 64 nodes and 256 nodes to evaluate Algorithm 2 for `MPI_Allgatherv`. Setting 5 is when $p = q$ and all processes are sending exactly the same data size. This setting has the same end result as the first setting used to evaluate Algorithm 1. Setting 6 is when $p = q$ and processes in each of the groups are sending messages with block sizes forming an arithmetic sequence, e.g., rank i sends a message of size ki for some a chosen, fixed k. This setting evaluates extreme cases of imbalanced send sizes. Setting 7 is when $\frac{p}{q} = \frac{25}{7}$ and all processes are sending equal-sized messages. This setting has the same end result as the fourth setting used for evaluating Algorithm 1. Finally, setting 8 is when $\frac{p}{q} = \frac{25}{7}$ and the send sizes form an arithmetic sequence. This setting evaluates the case of imbalanced process numbers and irregular message sizes.

We experiment with the different parameter settings described on three MPI process configurations. For the first two configurations, one process is assigned to every node of 64 nodes and 256 nodes. Hence the total number of processes is 64 and 256 respectively. These two setups emulate a homogeneous communication cost model where a data transfer between any two processes has the same communication cost. For hybrid MPI and OpenMP applications that use shared memory at the same node and process communication across nodes, this setup is reasonable. For the last setup, we assign 32 processes to every node of the 64 nodes. Thus, the total number of processes is 2048. This setup emulates applications that fully utilize cores with MPI. We perform experiments with maximum message sizes from 64KBytes to 8MBytes per node. Fig. 4 shows the run time comparison between Algorithm 1 and the MPI library native root gathering algorithm. Fig. 5 shows the run time comparison between Algorithm 2 and the MPI library native root gathering algorithm.

As explained in Section 3.1, the version of the root gathering algorithm implemented by MPICH has transfer time $(M + 3pk_A + 3qk_B)t_w$ by Eq. (2) for $M = \max(pk_A, qk_B)$. The transfer time of Algorithm 1 is $\max(pk_A,\ qk_B)t_w$. The expected improvement is therefore

$$\frac{M + 3pk_A + 3qk_B}{M}. \tag{5}$$

Fig. 4 illustrates the end to end execution time of the root gathering algorithm, Algorithm 2 in [1], and Algorithm 1 from message size 64KBytes to 8MBytes per process with settings 1–4 and three process configurations mentioned previously. Since the theoretical transfer time of Algorithm 1 is equivalent to that of Algorithm 2 presented in [1], we expect a similar performance of these two algorithms for all settings when message size is large. The results in Fig. 4 matches this expectation. Setting 2, 3, and 4 provide additional results for settings that are not presented in [1], where p and q are indivisible of each other.

Fig. 5 illustrates the end to end execution time of the root gathering algorithm and Algorithm 2 from message size 64KBytes to 8MBytes per node with settings 5–6 and three process configurations mentioned previously. The proposed algorithm shows much better performance than expected with 256 processes running on 256 nodes. This observation can be explained by the MPICH implementation that implements the intergroup `MPI_Allgatherv` slightly different from intergroup `MPI_Allgather`. The gathering operations of messages to roots are implemented using a ring algorithm. If the communication cost between nodes is the same, the transfer time of the ring-based intragroup Gather algorithm achieves the lower bound of intragroup Gather. However, as mentioned before, Cori is pseudo-fully connected, so every step of the ring algorithm is dominated by the largest communication cost between nodes with adjacent ranks, which can lead to poor performance as the number of nodes increases. As a result, the overall performance of the root gathering algorithm is worse than its theoretical performance on 256 nodes.

Fig. 6 illustrates the improvements in Algorithms 1 and 2 over the root gathering algorithm for all eight settings and three process configurations. When message sizes are small, the performance of the proposed algorithms is not necessarily better than the performance of the root gathering algorithm, because the t_w term is relatively small compared with t_s which represents the communication cost independent of the message size. Discussion for small message sizes is beyond the scope of this paper since our objective is to optimize the t_w term. For large message sizes, we can observe that the performance of the proposed algorithms is many times faster than the performance of the root gathering algorithm. According to Eq. (5), improvement curves for setting 1, 2, 3, 4, 5, 6, 7, and 8 should converge to 7.0, 4.84, 4.21, 6.68, 7.0, 7.0, 4.84, and 4.84 respectively as the message size increases. Since Cori is not strictly fully connected, the homogeneous communication cost assumption is not necessarily valid. Thus, the improvements may not converge to the exact ratios, since different pairs of arbitrarily selected processes can have different communication cost. Nonetheless, the significant improvements with large message size experiments demonstrate the robustness of the proposed algorithms under a weak assumption for communication cost for different process configurations and communication patterns.

6. Conclusion

In this paper, we proposed scalable algorithms for the full-duplex intergroup All-to-All broadcast collective operations that can readily be used to implement the `MPI_Allgather` and `MPI_Allgatherv` collectives of MPI. Under single-port communication assumptions, the proposed algorithms achieve the lower bound for optimal transfer time asymptotically. Theoretical analysis shows that the proposed algorithms can achieve much lower transfer times than the root gathering algorithm that is adopted by many modern MPI production libraries. We have implemented the proposed algorithms. Experimental results are in line with our theoretical analysis under a broad selection of configurations (numbers of processes, relative block sizes, process placement), and the practical gains correspond to the model predictions. In addition, these results show that the relative improvements are robust also for hierarchical systems where our homogeneous cost model does not apply. Consequently, we suggest using these implementations in MPI libraries instead of the now commonly used root gathering algorithm.

Intergroup collective operations embody communication patterns that cannot easily be expressed with the ordinary, intragroup collectives, namely bipartite patterns between processes in two disjoint groups where all communication is between groups and never within groups.

Although it is easily possible to improve over the root gathering algorithm also for the intergroup `MPI_Bcast`, `MPI_Gather`, `MPI_Gatherv`, `MPI_Scatter` and `MPI_Scatterv` operations, partly by the ideas presented in this paper, partly by reusing ideas from algorithms for the corresponding intragroup collectives, the full-duplex All-to-All operations still pose interesting challenges. While MPI can only express bipartite patterns over two disjoint process groups, it might be worthwhile to study multi-partite patterns both for use in concrete application and to gain insights into more general collective communication operations.

Acknowledgment

This work is supported in part by the DOE office of Advanced Scientific Computing Research, awards DE-SC0014330 and DE-SC0019358. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S Department of Energy under Contract NO. DE-AC02-05CH11231.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.parco.2019.04.015. Source code for experiments is on http://github.com/QiaoK/Intergroup_Allgather_Evaluation.

References

[1] Q. Kang, J.L. Träff, R. Al-Bahrani, A. Agrawal, A. Choudhary, W.-k. Liao, Full-duplex inter-group all-to-all broadcast algorithms with optimal bandwidth, in: Proceedings of the 25th European MPI Users' Group Meeting, ACM, 2018, p. 1.
[2] MPI Forum, MPI: amessage-passing interface standard. Version 3.1, 2015. http://www.mpi-forum.org.
[3] J. Liao, B. Gerofi, G.-Y. Lien, T. Miyoshi, S. Nishizawa, H. Tomita, W.-K. Liao, A. Choudhary, Y. Ishikawa, A flexible I/O arbitration framework for netCDF-based big data processing workflows on high-end supercomputers, Concurrency Comput. 29 (15) (2017) e4161.
[4] J. Hacker, W. Angevine, Ensemble data assimilation to characterize surface-layer errors in numerical weather prediction models, Mon. Weather Rev. 141 (6) (2013) 1804–1821.
[5] W. Gropp, E. Lusk, Fault tolerance in message passing interface programs, Int. J. High Perform. Comput. Appl. 18 (3) (2004) 363–372.
[6] F. Zhang, C. Docan, M. Parashar, S. Klasky, N. Podhorszki, H. Abbasi, Enabling in-situ execution of coupled scientific workflow on multi-core platform, in: 26th International Parallel & Distributed Processing Symposium (IPDPS), IEEE, 2012, pp. 1352–1363.
[7] C. Docan, F. Zhang, T. Jin, H. Bui, Q. Sun, J. Cummings, N. Podhorszki, S. Klasky, M. Parashar, Activespaces: exploring dynamic code deployment for extreme scale data processing, Concurrency Comput. 27 (14) (2015) 3724–3745.
[8] Mpich3, 2017, (http://www.mpich.org/downloads/).
[9] D.P.A. Mamidala, A. Vishnu, Efficient shared memory and rdma based design for mpi_allgather over infiniband, 2006.
[10] E. Gabriel, G.E. Fagg, G. Bosilca, T. Angskun, J.J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R.H. Castain, D.J. Daniel, R.L. Graham, T.S. Woodall, 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, in: Lecture Notes in Computer Science, 3241, Springer, 2004, pp. 97–104.
[11] V. Kumar, A. Grama, A. Gupta, G. Karypis, Introduction to parallel computing: design and analysis of algorithms, 400, Benjamin/Cummings, 1994.
[12] E. Chan, M. Heimlich, A. Purkayastha, R. Van De Geijn, Collective communication: theory, practice, and experience, Concurrency Comput. 19 (13) (2007) 1749–1783.
[13] S.L. Johnsson, C.-T. Ho, Optimum broadcasting and personalized communication in hypercubes, IEEE Trans. Comput. 38 (9) (1989) 1249–1268.
[14] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, D. Weathersby, Efficient algorithms for all-to-all communications in multiport message-passing systems, IEEE Trans. Parallel Distrib. Syst. 8 (11) (1997) 1143–1156.
[15] R. Thakur, R. Rabenseifner, W. Gropp, Optimization of collective communication operations in MPICH, Int. J. High Perform. Comput. Appl. 19 (1) (2005) 49–66.
[16] J.L. Träff, A. Ripke, C. Siebert, P. Balaji, R. Thakur, W. Gropp, A pipelined algorithm for large, irregular all-gather problems, Int. J. High Perform. Comput. Appl. 24 (1) (2010) 58–68.
[17] P. Silva, J.G. Silva, Implementing MPI-2 extended collective operations, in: Recent Advances in Parallel Virtual Machine and Message Passing Interface. 6th European PVM/MPI Users' Group Meeting, Vol. 1697 of Lecture Notes in Computer Science, Springer, 1999, pp. 125–132.
[18] P. Sanders, J. Speck, J.L. Träff, Two-tree algorithms for full bandwidth broadcast, reduction and scan, Parallel Comput. 35 (12) (2009) 581–594.
[19] J.L. Träff, A. Ripke, Optimal broadcast for fully connected processor-node networks, J. Parallel Distrib. Comput. 68 (7) (2008) 887–901.