Uniform Algorithms for Reduce-scatter and (most) other Collectives for MPI

Jesper Larsson Träff, Sascha Hunold, Ioannis Vardas, and Nikolaus Manes Funk
TU Wien, Faculty of Informatics, Research Group Parallel Computing, Vienna, Austria
Email: {traff,hunold,vardas}@par.tuwien.ac.at

***Abstract*—We explore the use of a regular, circulant graph communication pattern for the implementation of the reduction-to-all (`MPI_Allreduce`), by specialization the reduction-to-root (`MPI_Reduce`), the reduce-scatter (`MPI_Reduce_scatter_block`), the all-to-all-broadcast (`MPI_Allgather`) and the rooted gather and scatter (`MPI_Gather` and `MPI_Scatter`) collective operations, all as found in MPI (the Message-Passing Interface), for commutative operators and for any number of processes. The reduction-to-all algorithm reconstructs the little known algorithm by Bar-Noy, Kipnis and Schieber (1993), which the paper considerably extends.**

We experiment with extensions and combinations of the algorithms for these operations, and examine their performance from the perspective of performance guidelines, and in direct comparison to the implementations in common MPI libraries. On a small cluster with 36×32 cores and two larger HPC production systems, we show that we can especially for `MPI_Reduce_scatter_block` achieve considerably better performance than standard MPI library implementations. Our algorithms can perform consistently, which the implementations in standard MPI libraries sometimes do not.

In a homogeneous, one-ported communication system with linear transmission costs, reduction-to-all, reduce-scatter and all-to-all-broadcast can all be implemented in $O(\log p + m)$ time steps for problems of size m with small constants which we analyze and discuss.

I. Introduction

The reduction collectives of MPI, the Message-Passing Interface [1], are among its most important operations, and find use in almost all large scale, parallel applications [2], [3], [4]. Recently, much work has been devoted to improving the reduction-to-all operation (`MPI_Allreduce`) in machine learning contexts [5]. The exotic reduce-scatter operation (`MPI_Reduce_scatter_block`) that performs a reduction on vectors and distributes the result vector blockwise across the calling processes is likewise important, e.g., in classical, distributed linear algebra algorithms [6], and as a building block for reduction-to-all algorithms that can be implemented as a reduce-scatter followed by an all-to-all-broadcast (`MPI_Allgather`) operation. These collective reduction operations are intimately linked, functionally and performance wise, which we explore and exploit in this paper.

We assume a system with p processes that can communicate with each other by sending and receiving messages in a specified communication pattern. In a communication operation, a process can send data to a neighboring process and at the same time receive data from another (one-ported, fully bidirectional communication). Neighboring processes are defined by a communication pattern (graph) that specifies which processes are allowed to communicate with which other processes. We let m denote the problem size, which is the (maximum) amount of data elements to be either sent or received by a process. A given commutative operator $\oplus$ can operate on two vectors of m elements at cost γm. A local copy of an m-element vector likewise takes γm time units. We assume homogeneous, linear communication costs with constant latency α and cost per item of β, $\alpha + \beta m = O(m)$.

We first derive a simple algorithm for the reduction-to-all operation (`MPI_Allreduce`) that works for any number of processes p when the binary operator used to combine two vectors is commutative (as are all built-in operators of MPI, e.g., `MPI_SUM`, `MPI_MAX`, `MPI_BOR`, ...). This algorithm is based on a circulant graph communication pattern, and requires the optimal $\lceil \log_2 p \rceil$ number of communication rounds [7], for any number of processes. The pattern turns out to be identical to the pattern of the perhaps little-known census function algorithm of Bar-Noy, Kipnis and Schieber [8], and our exposition can be seen as an intuitive reconstruction of this algorithm. By simply not performing certain communications, the algorithm can be specialized to the reduction-to-root (`MPI_Reduce`) collective as well. As our main result, we then show how to modify these two algorithms to the reduce-scatter operation (`MPI_Reduce_scatter`). A similar algorithm was given in [9], [10], but our derivation and resulting algorithms are simpler, as we show with concrete code snippets. The communication pattern can also be used for the all-to-all-broadcast operation (`MPI_Allgather`), and gives rise to a different round and volume optimal algorithm than the well-known algorithm of Bruck et al. [7] with only half the local copy cost. By specialization, the rooted gather (`MPI_Gather`) and scatter (`MPI_Scatter`) collectives can be covered as well. We finally explore combinations of these algorithms for the implementation of `MPI_Allreduce`, and also discuss adaptation to irregular collectives like `MPI_Allgatherv` and `MPI_Reduce_scatter`.

All algorithms of Section III have been implemented. In Section IV, we conduct experiments on a smaller 36 node cluster and on two larger HPC systems. Our `MPI_Reduce_scatter_block` (and `MPI_Reduce_scatter`) implementations significantly outperform standard MPI libraries. Our implementations perform consistently with respect to each other according to expectations formalized as performance guidelines. Standard MPI libraries often do not.

TABLE I: Worst case running times of our algorithms compared to standard butterfly/hypercube algorithms in terms of p and m in a one-ported, linear transmission cost model α, β with computation and local copy cost γ. We assume that p is not a power of two; otherwise the correction terms for the butterfly algorithms disappear.

Operation	Our result	Standard	
Reduction-to-all	$\lceil \log_2 p \rceil (\alpha + \beta m + 2\gamma m)$	$\lceil \log_2 p \rceil (\alpha + \beta m + \gamma m) + \alpha + \beta m$	(butterfly)
Reduction-to-root	$\lceil \log_2 p \rceil (\alpha + \beta m + \gamma m)$	$\lceil \log_2 p \rceil (\alpha + \beta m + \gamma m)$	(butterfly)
Reduce-scatter	$\lceil \log_2 p \rceil \alpha + (\beta + \gamma) \frac{2^{\lceil \log_2 p \rceil} - 1}{p} m$	$\alpha + \beta m/2 + \gamma m/2 + \lfloor \log_2 p \rfloor \alpha + (\beta + \gamma)(2^{\lfloor \log_2 p \rfloor} - 1) 2m/p + \alpha + \beta m/p$	(butterfly)
Allgather	$\lceil \log_2 p \rceil \alpha + \frac{p-1}{p} \beta m + \gamma \lceil \frac{m}{2} \rceil$	$\alpha + \beta m/p + \lfloor \log_2 p \rfloor \alpha + \beta (2^{\lfloor \log_2 p \rfloor} - 1) 2m/p + \alpha + \beta m + \gamma m$	(butterfly)
		$\lceil \log_2 p \rceil \alpha + \frac{p-1}{p} \beta m + \gamma m$	(Bruck)

II. Related Work

Circulant graph communication patterns have often been used for the various collective operations found in MPI, notably [7], [8], [11]. The pattern derived in here is different from the doubling pattern in [7], but turns out to be the same as that used by Bar-Noy, Kipnis and Schieber [8]. It was also used in [10], and indeed in many papers from the mid- to late nineties.

Reduction-to-all (`MPI_Allreduce`), reduction-to-root (`MPI_Reduce`), and reduce-scatter (`MPI_Reduce_scatter_block`, `MPI_Reduce_scatter`) operations are sometimes implemented by butterfly (or, equivalently: hypercube) algorithms, which assume that p is a power of two and do not easily generalize. On the other hand, butterfly algorithms can be made to work also for non-commutative operators [12], [13], [14], [15], and give theoretically optimal performance [16] when p is a power of two. For the extension beyond hypercubes presented for instance in [12], [13], the penalty is at least one more communication round than the optimal $\lceil \log_2 p \rceil$ rounds and $m/2$ extra data. The reduce-scatter algorithm here achieves a better bound.

The algorithms presented here work for any number of processes p and any problem size m. The analysis assumes a system with homogeneous communication costs (same costs between any communicating processes, proportional to m), which is a gross simplification for real-world clusters. However, the algorithms can be used as building blocks for hierarchical and multi-lane clusters to give better implementations as shown in [17], [18]. We do not go into details in this paper.

In Table I, we compare our algorithmic results to common, best-known butterfly-like algorithms.

III. Algorithms and Implementations

We now develop our algorithms for the reduction-like operations, all based on the same, uniform, circulant graph communication pattern, and indicate how the algorithms extend nicely to (almost all) other classical MPI collectives. We present the algorithms with C code snippets and MPI send and receive communication calls, so that they can be immediately implemented. Full code is available from the authors.

A. Circulant Graph Communication Pattern

In order to arrive at the optimal, logarithmic number of communication rounds, our algorithms use a "roughly-doubling" communication pattern. Each process will in each communication round receive data from a process that is a certain number of "skips" away and send data to another process the same number of "skips" away. More precisely, with p processes each with a rank $r \in \{0, 1, \ldots, p-1\}$, we use a *circulant graph* communication pattern with *to-* and *from-edges* (r, t_r^k) and (f_r^k, r) to be used in the kth send-receive communication round. The kth to- and from-processes t_r^k and f_r^k are defined as

$$\begin{aligned} t_r^k &= (r - \texttt{skips}[k] + \epsilon_k + p) \bmod p \\ f_r^k &= (r + \texttt{skips}[k] - \epsilon_k) \bmod p \end{aligned}$$

where $\texttt{skips}[k], k = 0, 1, \ldots$ are computed by repeated halving of p as shown in Listing 1; for convenience, we take $\texttt{skips}[q] = p$, see Table II. The halving is repeated $q = \lceil \log_2 p \rceil$ times, and all our algorithms will go through q communication rounds. The correction term $\epsilon_k \in \{0, 1\}$ for each round will be explained in the next section. In each round, each process will send on its to-edge and receive on its from-edge for that round. This pattern is clearly deadlock-free, since for each round the to-process of the from-process of any one process is indeed that process itself.

Listing 1: The process local computation of the q skips for p processes (equal to the size of the MPI communicator).

```
p = size; q = 0;
while (p>1) { q++; p -= p/2; }

int skips[q+1], e; // skips and correction term
skips[q] = size;
for (i=q-1; i>=0; i--) skips[i] = skips[i+1]-skips[i+1]/2;
```

TABLE II: The $\texttt{skips}[k]$ for $k = 0, 1, \ldots, \lceil \log_2 p \rceil$ for $p = 31, 32, 33$. For $p = 33$, $\texttt{skips}[6] = 33$ is not shown.

p	skips[0]	skips[1]	skips[2]	skips[3]	skips[4]	skips[5]
31	1	2	4	8	16	31
32	1	2	4	8	16	32
33	1	2	3	5	9	17

Lemma 1: For any $k, 0 \leq k < q$ it holds that $\texttt{skips}[k+1] \leq \texttt{skips}[k] + \texttt{skips}[k] \leq \texttt{skips}[k+1] + 1$. It trivially holds (since $\texttt{skips}[q] = p$ and $\texttt{skips}[0] = 1$) that $\sum_{k=0}^{q-1} (\texttt{skips}[k+1] - \texttt{skips}[k]) = p - 1$. For all $p > 1$, it holds that $\texttt{skips}[1] = 2$.

The intuition for the different reduction and all-to-all-broadcast algorithms is the following: In communication round k for $k = 0, 1, \ldots, q-1$, process r receives on its from-edge (f_r^k, r) reduced (or concatenated) data from the

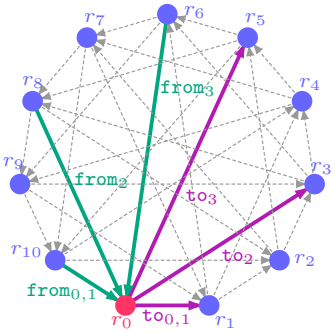


Fig. 1: The full circulant graph pattern with $p = 11$ processes. The to- and from-edges of rank r_0 are highlighted with the iteration as subscript.

processes in the range $[r + \texttt{skips}[k], r + \texttt{skips}[k+1] - 1]$ (all modulo p). We will maintain as an invariant in all of the algorithms that the reduction of data from processes in the range $[r+1, r+\texttt{skips}[k+1]-1]$ has been computed after round k. By Lemma 1, after the last communication round $q-1$, each process will then have received the reduced (or concatenated) data from all $p-1$ other processes. Maintaining the invariant will require the correct choice of the ϵ_k correction terms as will be seen in the next subsection.

B. Using the Pattern for Reduction-to-all

Let $\oplus$ be the commutative binary operator on the input vectors, and let V_r be the input vector for process r. All input vectors have size m (elements). The reduction-to-all operation (`MPI_Allreduce`) computes for each process the sum

$$W = \bigoplus_{i=0}^{p-1} V_i$$

of all input vectors.

The reduction-to-all algorithm will, more precisely, maintain as invariant for process r: After round $k, k = 0, 1, \ldots, q-1$, the algorithm has computed the sum

$$S_r^k = \bigoplus_{i=r+1}^{r+\texttt{skips}[k+1]-1} V_{i \bmod p}$$

of the input vectors of the $\texttt{skips}[k+1]-1$ next higher ranked processes after process r (modulo the number of processes p), *excluding* the input of the process itself.

Since always $\texttt{skips}[1] = 2$ (regardless of p), the invariant can be established if process r receives in the first round $k = 0$ the input vector of its from-process $(r+1) \bmod p$; consequently, process r then has to send its input vector V_r to its to-process $(r-1+p) \bmod p$. For the first round, we therefore take the correction term to be $\epsilon_0 = 0$.

To maintain the invariant after round $k, k > 0$, the first property of Lemma 1 tells that two cases have to be considered. Before round k, by assumption, each process r has computed the sum S_r^{k-1}.

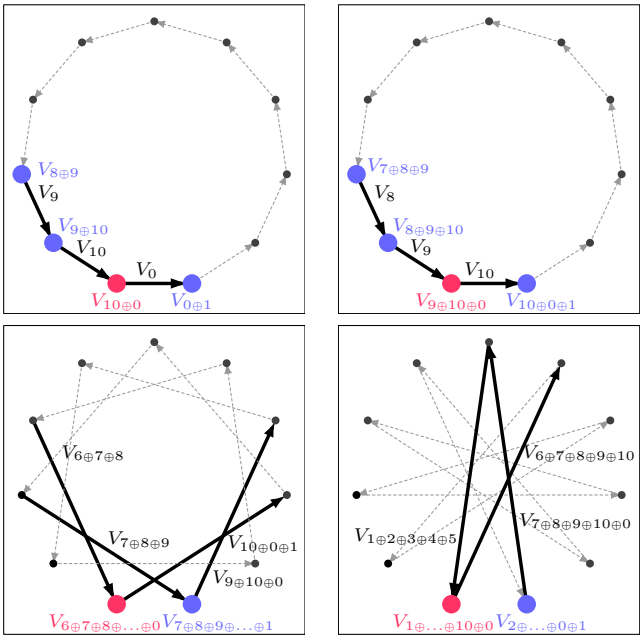


Fig. 2: The four communication rounds of the reduction-to-all algorithm with $p = 11$ processes.

Assume first that $\texttt{skips}[k] + \texttt{skips}[k] = \texttt{skips}[k+1]$, which is the case if and only if $\texttt{skips}[k+1]$ is even. Process r can compute $S_r^k = S_r^{k-1} \oplus (V_f \oplus S_f^{k-1})$ if it receives from its kth from-process $f = f_r^k = (r + \texttt{skips}[k]) \bmod p$ the sum $(V_f \oplus S_f^{k-1})$. The first term is the input vector of from-process f and the second term is the invariant sum S_f^{k-1} previously computed by process f. Also here, we take the correction term to be $\epsilon_k = 0$.

If on the other hand $\texttt{skips}[k]+\texttt{skips}[k] = \texttt{skips}[k+1]+1$ (which is the case iff $\texttt{skips}[k+1]$ is odd), then the sum $S_r^{k-1} \oplus V_f \oplus S_f^{k-1}$ with $f = (r + \texttt{skips}[k]) \bmod p$ would have one last term, namely $V_{(r+\texttt{skips}[k+1]) \bmod p}$ in excess, and we would thus not be able to maintain the invariant. However, the invariant can be maintained by using instead the invariant sum $S^{k-1}_{(r+\texttt{skips}[k]-1) \bmod p}$ of the process just before process f. Process r therefore receives instead from process $f_r^k = (r + \texttt{skips}[k] - 1) \bmod p$ the sum $S_{f_r^k}^{k-1}$ and correctly computes $S_r^k = S_r^{k-1} \oplus S_{f_r^k}^{k-1}$. By symmetry, process r sends S_r^{k-1} to process $t_r^k = (r - (\texttt{skips}[k] - 1) + p) \bmod p$ instead of t_r^k. The correction term in this case is $\epsilon_k = 1$.

At the end of the last round $q-1$, the result for process r is $V_r \oplus S_r^{q-1} = \bigoplus_{i=r+1}^{r+p-1} V_{i \bmod p}$, and all processes will have computed the same result (for commutative operators $\oplus$).

These observations give the same communication pattern and essentially the same algorithm as the census algorithm of Bar-Noy, Kipnis and Schieber [8]. The circulant graph for $p = 11$ processes is shown in Fig. 1, and the ensuing four communication and computation rounds in Fig. 2 .

A concrete MPI implementation is shown in Listing 2. The correction term $\texttt{e} = \epsilon_k$ is determined according to the two cases discussed above ($\texttt{skips}[k+1]$ even or odd), and `ciri` and

Listing 2: The implemented reduction-to-all algorithm.

```
inbuf  = partbuf;
for (i=0; i<q; i++) {
  if ((skips[i+1]&0x1)==0x0) { // skips[i]+skips[i]==skips[i+1]
    e = 0;
    outbuf = recvbuf;
  } else { // skips[i]+skips[i]>skips[i+1]
    e = 1;
    outbuf = partbuf;
  }
  ciri = (rank+skips[i]-e)%size;
  ciro = (rank-skips[i]+e+size)%size;

  MPI_Sendrecv(outbuf,count,datatype,ciro,ALLREDUCE,
               inbuf, count,datatype,ciri,ALLREDUCE,
               comm,MPI_STATUS_IGNORE);

  MPI_Reduce_local(inbuf,recvbuf,count,datatype,op);
  if (inbuf==tempbuf) {
    MPI_Reduce_local(tempbuf,partbuf,count,datatype,op);
  } else inbuf = tempbuf; // nothing to do in first round
}
```

`ciro` (for "circulant in, circulant out") are the corresponding from- and to-processes of the circulant graph.

The inflexibility of the two-argument `MPI_Reduce_local` operation unfortunately makes it necessary to maintain two partial sums, namely the sum of $\texttt{skips}[k+1]-1$ input vectors excluding the process' own input (in `partbuf`) and the sum of $\texttt{skips}[k+1]$ input vectors including the process' own vector (in `recvbuf`), see the discussion in [19]. Two reductions are done in each round (except for round $k=0$; the reductions in `partbuf` can furthermore be saved for the last round $k=q-1$, and the rounds before as long as $\texttt{skips}[k+1]$ is even).

We summarize the algorithm in the following proposition.

Proposition 1: The reduction-to-all problem for p processes with input vectors of m elements can be solved in $\lceil \log_2 p \rceil$ balanced communication rounds with m elements sent and received per process per round for a total communication volume of $\lceil \log_2 p \rceil m$ elements per process, and total cost of $\lceil \log_2 p \rceil (\alpha + \beta m + 2\gamma m) = O(m \log p)$ under homogeneous, linear communication and linear computation costs.

This direct algorithm for commutative reduction-to-all can be practically useful for small input vectors where m is below some (system dependent) threshold. Beyond that, different algorithms closer to the optimal cost of $O(\log p + \frac{(p-1)}{p} m)$ must be used, as discussed in Section III-G.

C. Using the Pattern for Reduction-to-root

The reduction-to-root operation (`MPI_Reduce`) likewise computes the sum of all input vectors

$$W = \bigoplus_{i=0}^{p-1} V_i$$

but stores the result only at one, specific, user-selected root process r. Hence, the algorithm for reduction-to-all can be specialized to reduction-to-root by only performing communication for the processes in the rounds where there is a dependency to the root process r. All other communication can simply be disregarded, which in our MPI implementation is done by sending and receiving from `MPI_PROC_NULL`.

The specialized implementation is shown in Listing 3. Each process i receives partial sums $S^k_{f^k_i}$ from from-processes f^k_i

Listing 3: The implemented reduction-to-root algorithm.

```
virt = (rank-root+size)%size;
i0   = pathhead(virt,q,skips);

if (virt==0||skips[i0+1]%2==0) {
  inbuf = tempbuf;

  MPICPY(recvbuf,sendbuf,count*extent);
} else {
  inbuf = recvbuf;
}
outbuf = sendbuf;

for (i=0; i<q; i++) {
  // skips[i]+skips[i]==skips[i+1]?
  e = ((skips[i+1]&0x1)==0x0) ? 0 : 1;
  ciri = (rank+skips[i]-e)%size;
  ciro = (rank-skips[i]+e+size)%size;

  if (i==0&&i<i0&&i0<q) {
    if (!(q>1&&skips[i0+1]%2==1)) ciro = MPI_PROC_NULL;
  } else if (i<i0) ciro = MPI_PROC_NULL;
  else ciri = MPI_PROC_NULL;
  MPI_Sendrecv(outbuf,count,datatype,ciro,ALLREDTAG,
               inbuf, count,datatype,ciri,ALLREDTAG,
               comm,MPI_STATUS_IGNORE);

  if (ciri!=MPI_PROC_NULL) {
    if (inbuf==tempbuf) {
      MPI_Reduce_local(inbuf,recvbuf,count,datatype,op);
    } else inbuf = tempbuf; // nothing to do in first round
  }
  outbuf = recvbuf;

  if (i==i0) break;
}
```

Listing 4: Finding the first round for process `virt` from which there is a path of increasing skips to virtual root process 0.

```
int pathhead(int virt, int q, int skips[]) {
  int i, jump, offs;
  if (virt==0||virt==skips[q]) return q;

  offs = 0; i = q;
  do {
    i--;
    jump = ((skips[i+1]&0x1)==0x1) ? skips[i]-1 : skips[i];
    if (offs+jump==virt) break;
    if (offs+jump<virt) offs += jump;
  } while (i>0);
  return i;
}
```

determined as explained for the reduction-to-all algorithm for communication rounds $k=0,1,\ldots$ until there is a sequence of $\texttt{skips}[k'] - \epsilon_{k'}$ skips for some $k' > k$ such that $r = i - \sum_{k'>k}(\texttt{skips}[k'] - \epsilon_{k'})$, meaning that the root process r can now be reached from process i in the remaining rounds. For each process i, this will be the case for some k, and in this round process i sends to the to-process t^i_k and can then terminate. The function `pathhead` shown in Listing 4 computes in $O(q) = O(\log p)$ steps for a process i (under the assumption that $r=0$) the round k at which process i sends its partial result. Since the partial sum S^k_i is only sent once (for the non-root processes), only this one sum needs to be maintained. This cuts the time spent in the $\oplus$ computations in half compared to the reduction-to-all algorithm.

It is interesting that both communication pattern and computations of our reduction-to-root algorithm are equivalent to those of our irregular reduce-scatter algorithm when all data are in one block only.

D. Using the Pattern for Reduce-scatter

The communication pattern and basic invariant from the reduction-to-all algorithm can be used for the reduce-scatter operation (`MPI_Reduce_scatter_block`) also. The input to the reduce-scatter operation is for each process r an m-

element vector V_r of p subvectors each with (roughly) m/p elements. The ith subvector is denoted by $V_r[i]$. As before, $\oplus$ is the given, commutative and associative operator on (sub)vectors. The reduce-scatter operation computes for each process r the sum

$$W_r = \bigoplus_{i=0}^{p-1} V_i[r]$$

of all rth input subvectors.

We use $S_r^k[i]$ for expressing the invariant on subvector i, which is defined as

$$S_r^k[i] = \bigoplus_{i=r+1}^{r+\texttt{skips}[k+1]-1} V_{i \bmod p}[i \bmod o] \quad .$$

By the end of the last communication round $q-1$, process r can compute the result for subvector r as $V_r[r] \oplus S_r^{q-1}$.

The invariant that process r will have computed $S_r^k[i]$ after round $k, k = 0, 1, \ldots, q-1$ will be maintained for certain subvectors i, using the same observations and communication pattern as described for the reduction-to-all algorithm. To see which subvectors, we let, for any process r, B_r^k denote the set of indices of the subvectors that will have to be sent by process r in communication round k. In round k, each process r sends subvectors to its to-process t_r^k as described for the reduction-to-all algorithm. At least the subvector for the to-process t_r^k itself has to be sent, and since the to-process will have to send some of the subvectors further in the rounds after round k, namely at least the subvector for the to-process of process $t_{t_r^k}^{k+1}$, we have that

$$B_r^k = \{t_r^k\} \cup \bigcup_{i=k+1}^{q-1} B_{t_r^k}^i \quad . \tag{1}$$

By induction downwards from round $q-1$, it easily follows that the number of subvectors that have to be sent (and received) in round k is $|B_r^k| = 2^{q-1-k}$ ($|B_r^{q-1}| = 1$ since in the last round only the subvector t_r^{q-1} has to be sent; the induction step follows immediately from the definition). Thus, the total number of subvectors sent over all rounds is $2^q - 1$. The crucial property on the subvectors to be sent in round k is that $B_r^{k+1} \subset B_{f_r^k}^k$: the subvectors that process r has to send in round $k+1$ is a subset of those that were received in round k (exactly half of them). If the B_r^k sets are enumerated the same way for each k, all reductions on subvectors can thus be performed in place without any reordering or copying of subvectors necessary. The enumeration (1) is implemented as shown in Listing 5, which computes the relative ranks of all reachable processes from round k onwards. The time of the algorithm is proportional to the size of the computed enumeration of B_r^k returned in the `block` array.

As an example, we consider reduce-scatter on $p = 9$ processes, and compute B_r^k for process $r = 8$. The algorithm in Listing 1 gives $\texttt{skips}[0] = 1, \texttt{skips}[1] = 2, \texttt{skips}[2] = 3, \texttt{skips}[3] = 5$ and $\texttt{skips}[4] = 9$, which for the actual communication pattern is adjusted to $1, 1, 2, 4$ for rounds $k = 0, 1, 2, 3$. Thus, process $r = 8$ sends to processes $7, 7, 6, 4$, and receives from processes $0, 0, 1, 3$. Using the definition of B_r^k we get:

$$\begin{aligned} B_8^0 &= [7, 6, 4, 0, 2, 5, 1, 3] \\ B_8^1 &= [7, 5, 1, 3] \\ B_8^2 &= [6, 2] \\ B_8^3 &= [4] \quad . \end{aligned}$$

The corresponding sets of received subvectors for process $r = 8$ are

$$\begin{aligned} B_0^0 &= [8, 7, 5, 1, 3, 6, 2, 4] \\ B_0^1 &= [8, 6, 2, 4] \\ B_1^2 &= [8, 4] \\ B_3^3 &= [8] \end{aligned}$$

and indeed $B_8^1 \subset B_0^0, B_8^2 \subset B_0^1, B_8^3 \subset B_1^2$ as claimed above.

We can therefore, once and for all, put the subvectors of each input vector V_r into the order in which the sets B_r^k are enumerated. This process local permutation is similar to the permutation trick that was used in [13] for reduce-scatter in a butterfly (hypercube), and saves copying of non-consecutive subvectors of the result vector back and forth between in each communication round.

The (almost) full implementation of the reduce-scatter operation is shown in Listing 6. We summarize the properties of the algorithm as follows.

Proposition 2: The reduce-scatter problem for p processes with input vectors of m elements can be solved in $\lceil \log_2 p \rceil$ (almost) balanced communication rounds, and a total communication and computation volume of $\frac{(p'-1)}{p} m$ vector elements per process, where p' is the smallest power of two greater than or equal to p.

Since $p' < 2p$, the total time cost under homogeneous, linear communication costs is therefore $O(\log p + \frac{(p-1)}{p} m)$, see Table I. The algorithm sends and receives $|B_r^k|$ subvectors of size (roughly) m/p elements in communication round k, for a total of $p' - 1 = 2^q - 1$ subvectors. If the input subvectors differ (considerably) in size, then up to m elements can be sent by some process in each communication round, which will lead to a dependent sequence of communication operations with a volume of up to $\lceil \log_2 p \rceil m$ vector elements. Thus, the same algorithmic pattern can be used for

Listing 5: Computing the relative block offsets.

```
int relative_blocks(const int off, const int k, const int q,
                    int skips[], int b, int block[])
{
  int i;
  for (i=k; i<q; i++) {
    block[b] = off+skips[i];
    if ((skips[i+1]&0x1)==0x1) block[b] -= 1;
    b = relative_blocks(block[b],i+1,q,skips,b+1,block);
  }
  return b;
}
```

Listing 6: The implemented reduce-scatter algorithm.

```
b = relative_blocks(0,0,q,skips,0,block);
b = (b+1)>>1;

// explicit round 0: permute-pack, selective reduction
takebuf = sendbuf;
for (j=0; j<b; j++) {
  MPICPY((char*)partbuf+j*count*extent,
         (char*)takebuf+((rank-block[j]+size)%size)*count*extent,
         count*extent);
}
ciri = (rank+skips[0])%size;
ciro = (rank-skips[0]+size)%size;

MPI_Sendrecv(partbuf,b*count,datatype,ciro,REDSCAT,
             tempbuf,b*count,datatype,ciri,REDSCAT,
             comm,MPI_STATUS_IGNORE);

MPICPY(recvbuf,(char*)takebuf+rank*count*extent,count*extent);
MPI_Reduce_local(tempbuf,recvbuf,count,datatype,op);

boff = 0; bb = b;
for (i=1; i<q; i++) { // remaining q-1 rounds
  bb >>= 1;
  if ((skips[i+1]&0x1)==0x0) { // skips[i]+skips[i]==skips[i+1]
    for (j=boff; j<boff+bb; j++) {
      MPICPY((char*)partbuf+j*count*extent,
         (char*)takebuf+((rank-block[b+j]+size)%size)*count*extent,
         count*extent);
    }
    MPI_Reduce_local((char*)tempbuf+(boff+1)*count*extent,
                     (char*)partbuf+boff*count*extent,
                     bb*count,datatype,op);
  } else { // skips[i]+skips[i]>skips[i+1]
    MPICPY((char*)partbuf+boff*count*extent,
           (char*)tempbuf+(boff+1)*count*extent,bb*count*extent);
  }
  boff += bb;
}
b >>= 1; // now done with round 0

for (i=1; i<q; i++,b>>=1) {
  e = ((skips[i+1]&0x1)==0x0) ? 0 : 1;
  ciri = (rank+skips[i]-e)%size;
  ciro = (rank-skips[i]+e+size)%size;

  MPI_Sendrecv(partbuf,b*count,datatype,ciro,REDSCAT,
               tempbuf,b*count,datatype,ciri,REDSCAT,
               comm,MPI_STATUS_IGNORE);

  MPI_Reduce_local(tempbuf,recvbuf,count,datatype,op);
  partbuf = (char*)partbuf+b*count*extent;
  MPI_Reduce_local((char*)tempbuf+count*extent,
                   partbuf,(b-1)*count,datatype,op);
}
```

Listing 7: The implemented all-to-all-broadcast algorithm.

```
if (rank+last<=size) {
  lowrbuf = (char*)recvbuf+rank*recvcount*extent;
  if (rank==0) upprbuf = (char*)recvbuf+last*recvcount*extent;
  else upprbuf = (void*)malloc(space);
  if (sendbuf!=MPI_IN_PLACE) MPICPY(lowrbuf,sendbuf,sendcount*extent);
} else {
  lowrbuf = (void*)malloc(space);
  upprbuf = (char*)recvbuf+(rank+last-size)*recvcount*extent;
  takebuf = (sendbuf!=MPI_IN_PLACE) ?
    sendbuf : (char*)recvbuf+rank*recvcount*extent;
  MPICPY(lowrbuf,takebuf,recvcount*extent);
}

for (k=0; k<q; k++) {
  b = skips[k+1]-skips[k];

  if ((skips[i+1]&0x1)==0x0) { // skips[i]+skips[i]==skips[i+1]
    e = 0;
    takebuf = lowrbuf;
  } else {
    e = 1;
    takebuf = (char*)lowrbuf+recvcount*extent;
  }
  ciri = (rank+skips[i]-e)%size;
  ciro = (rank-skips[i]+e+size)%size;
  tempbuf =
    (k<q-1) ? (char*)lowrbuf+skips[k]*recvcount*extent : upprbuf;

  MPI_Sendrecv(takebuf,b*recvcount,recvtype,ciro,ALLGATHER,
               tempbuf,b*recvcount,recvtype,ciri,ALLGATHER,
               comm,MPI_STATUS_IGNORE);
}

if (rank+last>size) {
  MPICPY((char*)recvbuf+rank*recvcount*extent,
         lowrbuf,(size-rank)*recvcount*extent);
  MPICPY(recvbuf,(char*)lowrbuf+(size-rank)*recvcount*extent,
         (rank+last-size)*recvcount*extent);
  free(lowrbuf);
} else if (rank>0) {
  MPICPY((char*)recvbuf+(rank+last)*recvcount*extent,
         upprbuf,(size-rank-last)*recvcount*extent);
  MPICPY(recvbuf,(char*)upprbuf+(size-rank-last)*recvcount*extent,
         rank*recvcount*extent);
  free(upprbuf);
}
```

both the regular `MPI_Reduce_scatter_block` and the irregular `MPI_Reduce_scatter` operation, with graceful performance degradation from the best case of roughly equal sized blocks to the extreme case where one process will store the full result vector.

E. Using the Pattern for All-to-all-broadcast

The same communication pattern can also be used for the all-to-all-broadcast operation (`MPI_Allgather`). Let the binary operator $\oplus$ be cyclic concatenation. Each process has a block of roughly m/p data elements, and by the end of the algorithm, each process shall have collected (the concatenation of) the data blocks from all processes in rank order for a total of m elements. The specification of the `MPI_Allgather` operation is that all processes collect the blocks in the same order from 0 to $p-1$. In the reduction-to-all algorithm, partial results are collected in ranges from $[f_r^k + \texttt{skips}[k], f_r^k + \texttt{skips}[k+1] - 1]$ (all modulo p) for the from-processes f^k, and these ranges will in some rounds not be consecutive. In order to avoid having to deal with broken, non-consecutive ranges of blocks, a straightforward adaptation of the reduction-to-all algorithm would use an intermediate buffer for each process r to store blocks in consecutive order, and after the last communication round copy the result from this buffer into the output buffer in rank order as required by the MPI specification. This entails a copy overhead of m data elements. As now shown in Listing 7, this copy overhead can be reduced by a factor of two.

The observation is the following. In communication round k, each process r receives and sends a sequence of $\texttt{skips}[k+1] - \texttt{skips}[k]$ data blocks, which for $k = q-1$ is $\lfloor p/2 \rfloor$. We introduce two communication buffers, such that the first $\texttt{skips}[q-1]$ blocks are in the `lowrbuf` buffer, and the remaining $p - \texttt{skips}[q-1]$ blocks are in `upprbuf`. At most one of these ranges is non-consecutive (broken into two), depending on the process rank r, and only for this range an intermediate buffer must be used. The copying effort after communication is therefore reduced to at most $\lceil p/2 \rceil$ blocks (for process $r = 0$, no copying is needed). The shorthand `MPICPY` denotes "type-correct" copying in MPI, respecting the data type and layout as can be specified with MPI derived datatypes, see again the discussion in [19].

Proposition 3: The all-to-all-broadcast problem for p processes with input blocks of roughly m/p elements can be solved in $\lceil \log_2 p \rceil$ balanced communication rounds, and a total communication volume of $\frac{(p-1)}{p} m$ elements sent and received per process. The total time cost under homogeneous, linear communication costs is $\lceil \log_2 p \rceil \alpha + \beta \frac{(p-1)}{p} m + \gamma m/2 = O(\log p + \frac{(p-1)}{p} m)$.
This communication volume is optimal [7]. The $p-1$ is the number of input blocks (of size m/p elements) that are sent and received in total per process over the $\lceil \log_2 p \rceil$ communication rounds.

The algorithm has the same overall properties as the well-

known, straight doubling algorithm of Bruck et al. [7]. Here, the number of blocks per round is a power-of-two (straight doubling), except for the last round where the number of blocks may be smaller and not a power of two. For this reason, it is not possible to maintain the invariant for the reduction-to-all algorithm, which relied on the fact that $\texttt{skips}[k]+\texttt{skips}[k]$ is at most one greater than $\texttt{skips}[k+1]$, which is the reason why the straight doubling pattern cannot be used for the reduction-to-all and reduce-scatter operations. Also, for the straight doubling algorithm, the trick with the two `lowrbuf` and `upprbuf` communication buffers each with at most $m/2$ elements, only one of which has to be copied back, will not work: some processes will have to copy all m elements (another way of dealing with the problem is to send broken buffers in two parts).

The algorithm can in a straightforward fashion be extended to cover also the irregular `MPI_Allgatherv` operation in which each process r contributes a block of some m_r elements for total number of $m = \sum_{r=0}^{p-1} m_r$ elements to be gathered at each process. We just have to keep track of the total number of elements for round k where $\texttt{skips}[k+1] - \texttt{skips}[k]$ blocks are sent and received. The resulting algorithm has similar properties to the regular `MPI_Allgather` algorithm, but in the worst case where one process has all the m data, with a communication volume of $\lceil \log_2 p \rceil m$ per process.

Proposition 4: The irregular all-to-all-broadcast problem for p processes with input blocks of roughly m_r elements can be solved in $\lceil \log_2 p \rceil$ communication rounds, and a total communication volume of at most $\lceil \log_2 p \rceil m$ elements sent and received per process where $m = \sum_{r=0}^{p-1} m_r$.

This is not optimal [20], but may be acceptable for not too irregular problems where the m_r inputs for the processes do not differ too much. The algorithm in a sense degrades gracefully from the best case where $m_r = m/p$ to the worst case where $m_r = m$ for one process only.

F. Using the Pattern for Gather- and Scatter-to-root

Similarly to the observation in Section III-C, the rooted gather operations (`MPI_Gather`, `MPI_Gatherv`) are specializations of the all-to-all-broadcast operations (`MPI_Allgather`, `MPI_Allgatherv`). Hence, algorithms for the rooted operations can be obtained from the all-to-all-broadcast algorithms by disregarding all communication for which there is no dependency from the root. By reversing the direction of the communication, also the rooted scatter operations (`MPI_Scatter`, `MPI_Scatterv`) can be implemented with the circulant graph communication pattern.

G. Combined Algorithm for Reduction-to-all

For large vectors m, the reduction-to-all algorithm with a communication volume of $\lceil \log_2 p \rceil m$ vector elements per process is unacceptable. It is a standard observation that reduction-to-all can be performed by a reduce-scatter followed by an all-to-all-broadcast operation, as also described in Section IV-C. We have implemented `MPI_Allreduce` by this observation, building on our `MPI_Reduce_scatter` and `MPI_Allgatherv` implementations, and divide the input vector of m elements into p almost equal sized blocks of roughly m/p elements.

IV. Experimental Evaluation

We now compare our new circulant graph implementations of `MPI_Reduce_scatter_block`, `MPI_Reduce_scatter`, `MPI_Allreduce`, and `MPI_Reduce`, first against the performance of native MPI library implementations, and second against themselves in order to explore whether they perform consistently according to expectation. The former comparison is found in Section IV-B, the latter in Section IV-C with a discussion of reasonable and desirable expectations. The `MPI_Allgatherv` implementation is benchmarked only implicitly as part of the combined `MPI_Allreduce` implementation in terms of `MPI_Reduce_scatter` and `MPI_Allgatherv`.

A. Experimental Methodology and Hardware/Software Setup

We have performed experiments on three different parallel systems, listed in Table III. Two machines, *Irene* and *Theta*, are large-scale production systems, where experimental results are subject to noise due to external workloads [21]. To reduce the chance of reporting workload artifacts, we repeated each batch of experiments on different days.

We obtain the running times of the collective communication operations using ReproMPI [22]. The ReproMPI benchmark collects timings for standard MPI functions, such as `MPI_Allreduce` or `MPI_Reduce_scatter_block`. To measure the running times of our circulant graph implementations, we leverage the PMPI interface to redirect MPI calls to our own implementations.[1] To that end, we have implemented all circulant algorithms as part of the PGMPITuneLib [23].

For all experimental results, we report the median running time across all repetitions, as the mean would be impacted strongly by straggler processes, which are usually caused by external workloads rather than by the collective algorithms themselves. ReproMPI has the unique feature to record measurements not only by defining a number of repetitions but also by a maximum experiment time. On all three machines, we record the running time of each collective for either 5000 repetitions or for a maximum of $3\,\mathrm{s}$, whatever occurs first. We use `MPI_BYTE` as base datatype and `MPI_BOR` as reduction operator on the buffers.

It is important to understand which buffer sizes are reported. For `MPI_Reduce_scatter_block`, we report the running time as a function of the size of the individual, per process buffers received in the scatter part. For example, if we report the runtime of `MPI_Reduce_scatter_block` for $1\,\mathrm{Byte}$ messages and p processes, the buffer used in the reduce part contains p Bytes, but each process eventually receives $1\,\mathrm{Byte}$ in the scatter part. For the other collectives, e.g., `MPI_Allreduce` and `MPI_Reduce`, the runtimes are reported for the entire buffer size.

[1] https://github.com/parlab-tuwien/mpi-circulant-collectives

TABLE III: Overview of hardware and MPI libraries used in experimental evaluation.

machine	compute nodes	interconnect	compiler	MPI libraries
Hydra	36 compute nodes (Dell) $2 \times$ 16-core Intel Skylake@2.1GHz	Dual-rail Intel Omni-Path 100	gcc 10.2.1	Open MPI 4.1.5, MVAPICH2 2.3.7 MPICH 4.1.1, Intel MPI 2021.7
Irene	1656 compute nodes (Atos) $2 \times$ 24-core Intel Skylake@2.7GHz	Infiniband EDR	gcc 8.3.0	Open MPI 4.1.4
Theta	4392 compute nodes (Cray XC40) 64-core, Intel Xeon Phi 7230@1.3GHz	Aries interconnect	icc 19.1.0.166	Cray MPICH 7.7.14

B. Experimental Results

We present the performance of our new `MPI_Reduce_scatter_block` implementation on the two production systems, *Irene* and *Theta*. In Fig. 3, we run with 100×48 (a non-power of two) and 256×1 (a power of two) processes, the latter configuration using only network communication between MPI processes. In both cases, our implementation outperforms Open MPI 4.1.4 by a factor on the order of 1.5. Figure 4 shows similar results for the *Theta* system, but the differences are much more pronounced, often showing an order of magnitude (factor 10) improvement. For the power-of-two configuration with 128×64 MPI processes, this is particularly revealing: Our implementation is (currently) not hierarchical and designed using a flat performance model, whereas the MPI library can be expected to take the node architecture (Xeon Phi) into account in some way. Our implementation is better by a very large margin.

Finally, we compare the performance of the other circulant graph collective implementations against the native MPI library implementations. Results for the *Theta* system with Cray MPICH 7.7.14 are shown in Fig. 5 for two process configurations. In the 128×64 configuration, the library and circulant graph `MPI_Reduce` operations are in the same ballpark for the smaller counts of up to 10 000 Bytes. It is surprising that the library implementation that could be expected to be tailored to the strongly hierarchal nature of the *Theta* system does not perform better. The `MPI_Allreduce` implemented by `MPI_Reduce_scatter` followed by `MPI_Allgatherv` performs $2\lceil \log_2 p \rceil$ communication rounds, and can therefore be expected to be a factor of two slower than the direct, circulant graph `MPI_Allreduce` algorithm for smaller counts. In the 128×64 configuration, performance is worse than this expectation. In the 150×1 configuration, for problems up to 10 000 Bytes, the library `MPI_Reduce` performs slightly better than `MPI_Allreduce`, and the same holds for the circulant graph implementations, which are in the same ballpark. For larger counts, the library `MPI_Reduce` performance degrades steeply, and becomes worse than that of `MPI_Allreduce`. The circulant graph implementations are better, and the combined, circulant graph `MPI_Allreduce` can slightly outperform the library native `MPI_Allreduce`.

C. Consistent Performance

We now explore how the different reduction operations and implementations can be expected to perform relative to each other. The reduce-scatter operation with p processes performs a reduction on input vectors of m elements, one for each process, and distributes the resulting vector in smaller, disjoint subvectors over the processes. The reduction-to-all operation, in contrast, stores the full resulting vector at each of the participating processes. The reduction-to-all operation is therefore more general than the reduce-scatter operation which can be implemented in terms of the reduction-to-all operation by simply ignoring the blocks of the result vector that are not relevant for the process. It is therefore natural to expect a concrete implementation of the reduce-scatter operation to perform better, or at least not worse, than an implementation of the reduction-to-all operation: If this is not the case, the slower reduce-scatter could be replaced by a reduction-to-all implementation which delivers only the relevant result block to the process. Such arguments and expectations on the actual performance of implemented collective operations are sometimes termed *performance guidelines* [24], [25]. For a rich interface like MPI with many different collective operations, a large number of such guidelines can be formulated and verified by appropriate benchmarking. A violated performance guideline exhibits problematic, not-to-be-expected behavior of an MPI library, and can be harmful to applications.

We examine more closely the following five guidelines that are expressed semi-formally below. They should hold for any high-quality MPI library implementation for any given number of processes on any communicator and for any given problem size.

$$\texttt{MPI\_Reduce\_scatter}(m) \preceq \texttt{MPI\_Allreduce}(m) \quad (2)$$

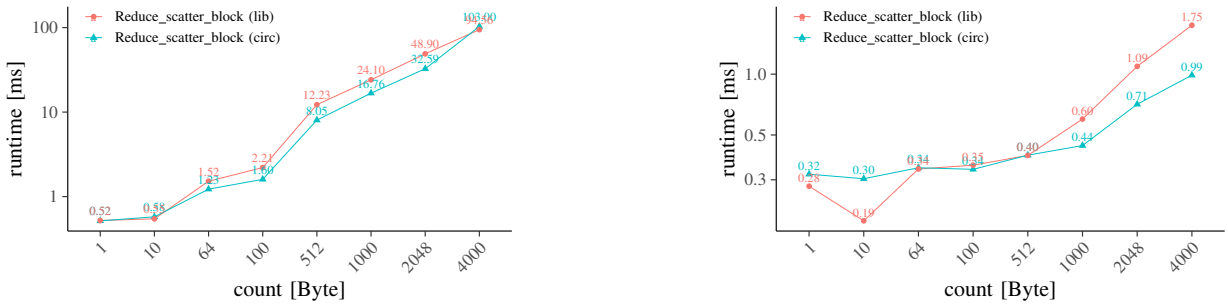
$$\texttt{MPI\_Reduce\_scatter}(m) \preceq \texttt{MPI\_Reduce}(m) + \texttt{MPI\_Scatterv}(m) \quad (3)$$

$$\texttt{MPI\_Allreduce}(m) \preceq \texttt{MPI\_Reduce\_scatter}(m) + \texttt{MPI\_Allgatherv}(m) \quad (4)$$

$$\texttt{MPI\_Reduce}(m) \preceq \texttt{MPI\_Allreduce}(m) \quad (5)$$

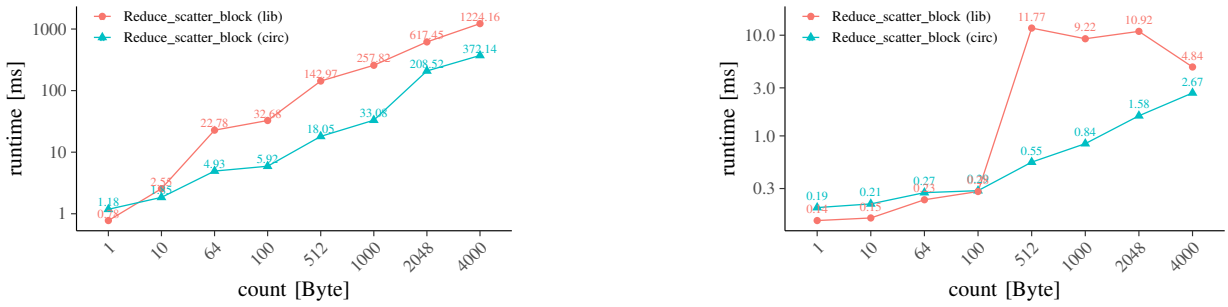
$$\texttt{MPI\_Reduce}(m, r) \approx \texttt{MPI\_Reduce\_scatter}(m, r) \quad (6)$$

Again, m is the size of the problem in number of input vector elements, and r a given root process. Guideline (2) semi-formalizes the discussion above; it should hold regardless of how the m element input vectors are divided into blocks, and thus also constrains the performance of `MPI_Reduce_scatter_block` which can be used when p divides m. Guideline (3) can be taken as the definition of `MPI_Reduce_scatter`, and can be formulated similarly for the regular `MPI_Reduce_scatter_block` operation by replacing `MPI_Scatterv` with `MPI_Scatter`. Guideline (4) formalizes the observation that the effect of `MPI_Allreduce`



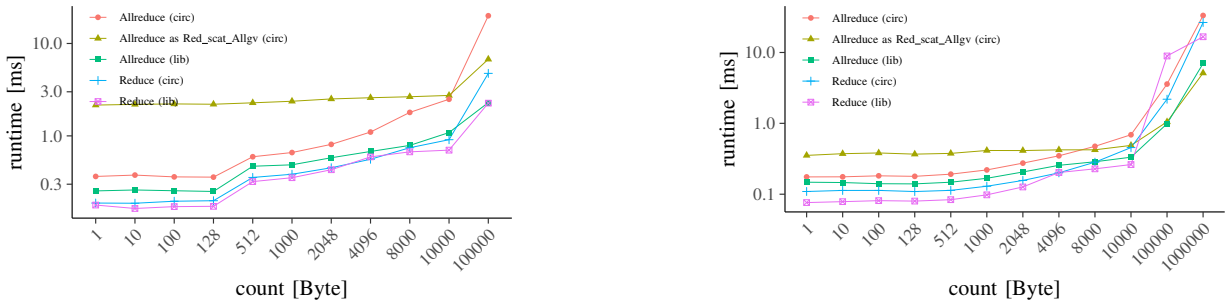
(a) 100×48 processes (b) 256×1 processes

Fig. 3: Runtime comparison of default MPI ("lib" is Open MPI 4.1.4) and circulant graph implementations ("circ") of `MPI_Reduce_scatter_block` on *Irene*.



(a) 128×64 processes (b) 150×1 processes

Fig. 4: Runtime comparison of default MPI ("lib" is Cray MPICH 7.7.14) and circulant graph implementations ("circ") of `MPI_Reduce_scatter_block` on *Theta*.



(a) 128×64 processes (b) 150×1 processes

Fig. 5: Direct comparison of default MPI ("lib" is Cray MPICH 7.7.14) and circulant graph implementations ("circ") of `MPI_Allreduce` and `MPI_Reduce` on *Theta*.

can be achieved by an `MPI_Reduce_scatter` followed by an `MPI_Allgatherv` operation. When p divides m, the guideline should also hold when replacing the right-hand side operations with `MPI_Reduce_scatter_block` and `MPI_Allgather`. Guideline (5) states that the specialized `MPI_Reduce` should perform no worse than `MPI_Allreduce`. Guideline (6) is to be read as follows. The `MPI_Reduce` operation with root r can be implemented as an `MPI_Reduce_scatter` with block size m for process r and block size 0 for all other processes, and we can expect `MPI_Reduce` to perform better (if not, it could be replaced by the degenerated use case of `MPI_Reduce_scatter`). The other way around, if an `MPI_Reduce_scatter` is called with only one non-empty block, it could be implemented by an `MPI_Reduce` call. It might be reasonable to expect that an MPI library exploits this observation.

We now discuss a series of performance guideline experiments both with our new implementations from Section III and with the implementations from standard MPI libraries. Quite significant violations of Guideline (2) were previously

found in [25]. In order to have a baseline for the relative, normalized performance figures given next, we first show absolute running times for `MPI_Reduce_scatter_block` for both library and circulant graph implementations for the four available MPI libraries on the *Hydra* system. Results for the 36×32 configuration are given in Fig. 6. Here, the circulant graph implementation consistently outperforms the library implementations, in the mid-range problem sizes by a large margin (the plots are doubly logarithmic). One library has a particular performance anomaly in this range.

We examine the relative performance of `MPI_Allreduce`, `MPI_Reduce`, and `MPI_Reduce_scatter_block` to guideline implementations (4) and (6). The measured running times have been normalized to the `MPI_Allreduce` operation implemented as in Guideline (4) by `MPI_Reduce_scatter` followed by `MPI_Allgatherv`. Results on the *Hydra* system with Open MPI 4.1.5 in configurations with 32×1, a power of two, and 36×32, a non-power of two, processes are shown in Fig. 7 with the circulant graph implementations on the left and the default MPI library implementations on the right.

The circulant graph implementations perform smoothly relative to the baseline. By construction, `MPI_Reduce` should perform better than `MPI_Allreduce` (Guideline (5) and (6)), and better than the combined implementation of `MPI_Allreduce` up to a certain number of input elements (Guideline (4)). The `MPI_Reduce_scatter` implementation is about a factor of two better than the baseline, and for small inputs all direct implementations perform in the same ballpark.

For the default library implementations, the picture is less clear, and severe violations of guidelines can be seen. We would for instance expect the `MPI_Reduce`, `MPI_Allreduce` and `MPI_Reduce_scatter_block` operations to perform better (not worse) than `MPI_Allreduce` implemented as `MPI_Reduce_scatter` followed by `MPI_Allgatherv`. For increasing input sizes, `MPI_Reduce` which can be seen as a special case of `MPI_Allreduce` is in many situations considerably worse than the baseline. Even more surprisingly, there are cases where `MPI_Reduce_scatter_block` is worse than the baseline, especially for the 32×1 configuration. Even the use of `MPI_Reduce_scatter` to implement `MPI_Reduce` (Guideline (6)) can sometimes be better than the native `MPI_Reduce` operation. As expected, the native `MPI_Allreduce` mostly performs better than the Guideline (4) implementation.

V. Summary

We showed that the same circulant graph communication pattern, with roughly halving skips that are not always powers-of-two, can be used to give easy-to-implement algorithms for the reduction-to-all, reduction-to-root, all-to-all-broadcast, and reduce-scatter collective operations, in both regular (same block sizes for all processes) and irregular (possibly different block sizes) variants, with expected performance similar to or better than currently used, "state-of-the-art" implementations. Our implementations follow

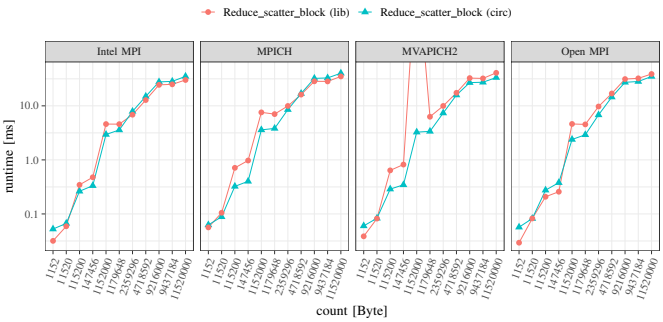


Fig. 6: Runtime comparison of different MPI ("lib") and circulant graph implementations ("circ") of `MPI_Reduce_scatter_block` on *Hydra* with 36×32 processes.

the MPI specification and can readily be used for `MPI_Allreduce`, `MPI_Reduce`, `MPI_Reduce_scatter_block`, `MPI_Reduce_scatter`, `MPI_Allgather`, and `MPI_Allgatherv` (as well as `MPI_Gather`, `MPI_Gatherv` and `MPI_Scatter`, `MPI_Scatterv`). The performance of in particular our `MPI_Reduce_scatter` implementation is in many cases of process and vector sizes significantly better (by a large factor) than that of standard MPI libraries. Our implementations do not take the hierarchical communication systems of clusters into account, but can be improved further by doing so with standard techniques [18].

Our reduction algorithms work for commutative operators. Even under this restriction, it is an interesting, to our knowledge, open question whether there is a reduce-scatter algorithm that for any number of processes p sends and receives exactly $\frac{p-1}{p}m$ data elements per process. For idempotent operators (`MPI_MAX`, `MPI_BAND`,...), we do know such an algorithm, which is an interesting observation.

Acknowledgments

We acknowledge PRACE for awarding us access to Joliot-Curie SKL hosted by GENCI@CEA, France. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. In particular, we acknowledge the MPICH project for providing the compute time at ALCF for conducting our research.

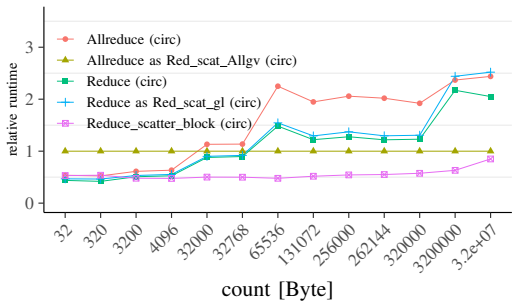
References

[1] MPI Forum, *MPI: A Message-Passing Interface Standard. Version 4.0*, June 9th 2021, www.mpi-forum.org.

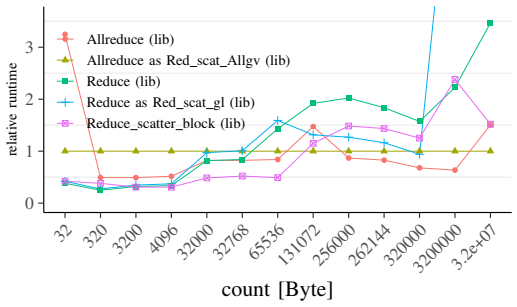
[2] N. Sultana, M. Rüfenacht, A. Skjellum, P. V. Bangalore, I. Laguna, and K. Mohror, "Understanding the use of message passing interface in exascale proxy applications," *Concurrency and Computation: Practice and Experience*, vol. 33, no. 14, p. 15 pages, 2021.

[3] D. E. Bernholdt, S. Boehm, G. Bosilca, M. G. Venkata, R. E. Grant, T. J. Naughton, H. Pritchard, M. Schulz, and G. R. Vallée, "A survey of MPI usage in the US exascale computing project," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 3, 2020.

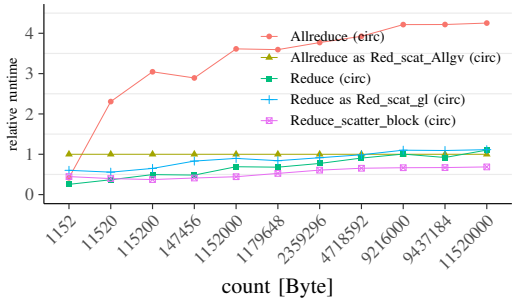
[4] S. Chunduri, S. Parker, P. Balaji, K. Harms, and K. Kumaran, "Characterization of MPI usage on a production supercomputer," in *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. IEEE/ACM, 2018, pp. 30:1–30:15.



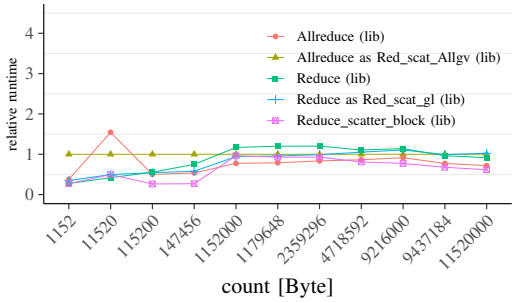
(a) 32×1 processes, circulant graph implementations



(b) 32×1 processes, default MPI



(c) 36×32 processes, circulant graph implementations



(d) 36×32 processes, default MPI

Fig. 7: Evaluation of performance consistency as expressed by Guidelines (2)–(6). Running times are shown relative to the implementation of `MPI_Allreduce` as `MPI_Reduce_scatter` followed by `MPI_Allgatherv` using either default MPI ("lib" is Open MPI 4.1.5) or circulant graph implementations ("circ") on *Hydra*.

[5] A. Castelló, E. S. Quintana-Ortí, and J. Duato, "Accelerating distributed deep neural network training with pipelined MPI allreduce," *Cluster Computing*, vol. 24, no. 4, pp. 3797–3813, 2021.

[6] M. Barnett, R. J. Littlefield, D. G. Payne, and R. A. van de Geijn, "Global combine algorithms for $2 - d$ meshes with wormhole routing," *Journal of Parallel and Distributed Computing*, vol. 24, no. 2, pp. 191–201, 1995.

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

[8] A. Bar-Noy, S. Kipnis, and B. Schieber, "An optimal algorithm for computing census functions in message-passing systems," *Parallel Processing Letters*, vol. 3, no. 1, pp. 19–23, 1993.

[9] G. Iannello, "Efficient algorithms for the reduce-scatter operation in LogGP," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 9, pp. 970–982, 1997.

[10] M. Bernaschi, G. Iannello, and M. Lauria, "Efficient implementation of reduce-scatter in MPI," *Journal of Systems Architecture*, vol. 49, no. 3, pp. 89–108, 2003.

[11] J. Bruck and C.-T. Ho, "Efficient global combine operations in multi-port message-passing systems," *Parallel Processing Letters*, vol. 3, no. 4, pp. 335–346, 1993.

[12] R. Rabenseifner and J. L. Träff, "More efficient reduction algorithms for message-passing parallel systems," in *11th European PVM/MPI Users' Group Meeting*, ser. Lecture Notes in Computer Science, vol. 3241. Springer, 2004, pp. 36–46.

[13] J. L. Träff, "An improved algorithm for (non-commutative) reduce-scatter with an application," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 12th European PVM/MPI Users' Group Meeting*, ser. Lecture Notes in Computer Science, vol. 3666. Springer, 2005, pp. 129–137.

[14] R. Thakur, W. D. Gropp, and R. Rabenseifner, "Improving the performance of collective operations in MPICH," *International Journal on High Performance Computing Applications*, vol. 19, pp. 49–66, 2005.

[15] R. van de Geijn, "On global combine operations," *Journal of Parallel and Distributed Computing*, vol. 22, pp. 324–328, 1994.

[16] E. Chan, M. Heimlich, A. Purkayastha, and R. A. van de Geijn, "Collective communication: theory, practice, and experience," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 13, pp. 1749–1783, 2007.

[17] M. Bayatpour, S. Chakraborty, H. Subramoni, X. Lu, and D. K. Panda, "Scalable reduction collectives with data partitioning-based multi-leader design," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2017, pp. 64:1–64:11.

[18] J. L. Träff and S. Hunold, "Decomposing MPI collectives for exploiting multi-lane communication," in *IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE Computer Society, 2020, pp. 270–280.

[19] J. L. Träff, I. Vardas, and N. M. Funk, "Library development with MPI: Attributes, request objects, group communicator creation, local reductions and datatypes," in *30th European MPI Users' Group Meeting (EuroMPI)*, 2023.

[20] J. L. Träff, "Fast(er) construction of round-optimal n-block broadcast schedules," in *IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE Computer Society, 2022, pp. 142–151.

[21] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There goes the neighborhood: performance degradation due to nearby jobs," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 2013, pp. 41:1–41:12.

[22] S. Hunold and A. Carpen-Amarie, "Reproducible MPI benchmarking is still not as easy as you think," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 12, pp. 3617–3630, 2016.

[23] S. Hunold and A. Carpen-Amarie, "Autotuning MPI collectives using performance guidelines," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia)*, 2018, pp. 64–74.

[24] J. L. Träff, W. D. Gropp, and R. Thakur, "Self-consistent MPI performance guidelines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 5, pp. 698–709, 2010.

[25] S. Hunold, A. Carpen-Amarie, F. D. Lübbe, and J. L. Träff, "Automatic verification of self-consistent MPI performance guidelines," in *Euro-Par Parallel Processing*, ser. Lecture Notes in Computer Science, vol. 9833, 2016, pp. 433–446.