An In-place Algorithm for Irregular All-to-All Communication with Limited Memory

Michael Hofmann[⋆] and Gudula Rünger

Department of Computer Science
Chemnitz University of Technology, Germany
{`mhofma`,`ruenger`}`@cs.tu-chemnitz.de`

Abstract In this article, we propose an in-place algorithm for irregular all-to-all communication corresponding to the `MPI_Alltoallv` operation. This in-place algorithm uses a single message buffer and replaces the outgoing messages with the incoming messages. In comparison to existing support for in-place communication in MPI, the proposed algorithm for `MPI_Alltoallv` has no restriction on the message sizes and displacements. The algorithm requires memory whose size does not depend on the message sizes. Additional memory of arbitrary size can be used to improve its performance. Performance results for a Blue Gene/P system are shown to demonstrate the performance of the approach.

Keywords: all-to-all, irregular communication, in-place, limited memory, MPI

1 Introduction

The amount of the memory required to solve a given problem can be one of the most important properties for algorithms and applications in parallel scientific computing. Since main memory is a limited resource, even for distributed memory parallel computers, the memory footprint of an application decides whether a certain problem size can be processed or not. Examples are parallel applications that use domain-decomposition techniques, e.g. mesh-based algorithms or particle codes. Adaptive or time-dependent solutions often require periodical redistributions of the workload and its associated data. This may require irregular communication based on `MPI_Alltoallv` where individual messages of arbitrary size are exchanged between processes. Even though the redistribution step may require only a small part of the runtime, it can significantly reduce the maximum problem size if a second fully-sized buffer has to be kept available only for receiving data during this step.

MPI communication operations commonly use separate send and receive buffers. MPI version 2.0 has introduced "in place" buffers for many intracommunicator collective operations using the `MPI_IN_PLACE` keyword. The resulting *in-place* communication operations use only a single message buffer and

[⋆] Supported by Deutsche Forschungsgemeinschaft (DFG).

replace the outgoing messages with the incoming messages. Support for an in-place `MPI_Alltoallv` operation was introduced in the MPI version 2.2 [1], but only with the restriction that counts and displacements of the messages to be sent and received are equal. In this article, we present an algorithm for an in-place `MPI_Alltoallv` operation with arbitrary counts and displacements. This in-place algorithm requires memory whose size does not depend on the message sizes. If additional memory is available, it can be used to speed up the in-place algorithm. The algorithm is described in the context of all-to-all communication, but can directly be adapted to many-to-many or sparse communication. We have implemented the algorithm and present performance results for a Blue Gene/P system using up to 4096 processes to demonstrate the efficiency of our approach.

The rest of this paper is organized as follows. Section 2 presents related work. Section 3 introduces the in-place algorithm for the `MPI_Alltoallv` operation. Section 4 shows performance results and Section 5 concludes the paper.

2 Related Work

Optimizations of MPI communication operations usually address latency and bandwidth results. Interconnection topologies and other architecture-specific properties are the subject of performance improvements, especially for high scaling parallel platforms [2]. Specific MPI implementations as well as the MPI specification itself are analyzed with respect to future scalability requirements [3]. Here, the memory footprint of an MPI implementation becomes an important optimization target.

Efficient data redistribution is a common problem in parallel computing. Especially data parallel programming models like High Performance Fortran provide support for flexible distributions of regular data structures to different processors. In this context, numerous algorithms for efficient redistributions of block-cyclic data distributions have been proposed (e.g. [4,5,6]). Algorithms for irregular data redistribution with limited memory have received less attention. Pinar et al. have proposed algorithms for data migration in the case of limited memory based on sequences of communication phases [7]. Siegel et al. have implemented various algorithms for data redistribution with limited memory in the MADRE library [8]. In [9], they have provided a modification to prevent livelocks in the basic algorithm of Pinar et al. In [10], we have proposed a fine-grained data distribution operation in MPI and provided several implementation variants including an in-place implementation based on parallel sorting.

3 An In-place Algorithm for `MPI_Alltoallv`

The `MPI_Alltoallv` operation is one of the most general collective communication operations in MPI. With p participating processes, each process sends p individual messages to the other processes and receives p messages from them. For each process, the sizes of the messages to be sent and received are given by arrays $scounts[1 \ldots p]$ and $rcounts[1 \ldots p]$. Additional arrays $sdispls[1 \ldots p]$ and

$rdispls[1...p]$ specify displacements that determine the locations of the messages. The standard MPI_Alltoallv operation uses separate send and receive buffers. Therefore, the locations of all messages are non-overlapping and all messages can be sent and received independently from each other.

The in-place MPI_Alltoallv operation uses a single buffer for storing the messages to be sent and received. This leads to additional dependencies for sending and receiving the messages. A message cannot be received until there is enough free space available at the destination process. Furthermore, a message cannot be stored at their target location if this location is occupied by a message that has to be sent (in advance). A trivial solution for this problem uses an intermediate buffer to receive the messages. This requires additional memory whose size depends on the size of the messages. The proposed in-place algorithm solves this problem using additional memory of a size independent from the message sizes.

3.1 Basic Algorithm

The algorithm is described from the perspective of a single process. We assume that all messages consist of data items of the same type and that the buffers used to store the messages are arrays of this type. Let S_i denote the set of indices belonging to the data items of the message to be sent to process i for $i = 1, \ldots, p$. Let R_i denote the set of indices of the locations where the incoming data items from process i should be stored. The initial index sets can be calculated from the given counts and displacements. We assume that the initial send index sets S_i are disjoint. The same applies to the initial receive index sets R_i. The messages are sent and received in several partial submessages and the index sets are updated as the algorithm proceeds. Even though the buffer can be seen as a large array, only the locations given by the initial index sets are accessible.

The in-place algorithm is based on the basic algorithm of Pinar et al. [7,9] and consists of a sequence of communication phases. In each phase the following steps are performed. (1) The number of data items that can be received in free space from every other process is determined. (2) These numbers are sent to the corresponding source processes. This represents an exchange of request messages between all processes that still have data items left to be exchanged with each other. (3) The data items are transferred. The algorithm terminates when all data items are exchanged. Algorithm 1 shows the basic algorithm adapted to our notation. Additionally, our algorithm includes procedures for the initialization of the index sets (line 2), for determining the items that can be received in free space (line 4), and for updating the index sets at the end of each communication phase (line 11). A description of these procedures is given in the following subsections. Exchanging the request messages (lines 5–7) and the data items (lines 8–10) can be implemented with non-blocking communication.

Siegel et al. have shown that the basic algorithm will neither deadlock nor livelock, provided there is additional free space on every process used to receive data items [9]. This is independent from the actual size of the additional free space and from the particular strategy that determines how free space is used

Algorithm 1 Basic algorithm of the in-place `MPI_Alltoallv` operation.

```
1:  let $recv[1...p]$ and $send[1...p]$ be arrays of integers
2:  init the index sets $S_i$ and $R_i$ for $i = 1, ..., p$ (see Sect. 3.2)
3:  while ($\sum_i |S_i| + \sum_i |R_i| > 0$) do
4:      $recv[1...p]$ = determine the number of data items to be received (see Sect. 3.3)
5:      exchange requests
6:        $\rightarrow$ send $recv[i]$ to process $i$ for all $i$ with $|R_i| > 0$
7:        $\rightarrow$ receive $send[i]$ from process $i$ for all $i$ with $|S_i| > 0$
8:      exchange data items
9:        $\rightarrow$ send $send[i]$ items at indices $S_i$ to process $i$ for all $i$ with $send[i] > 0$
10:       $\rightarrow$ receive $recv[i]$ items at indices $R_i$ from process $i$ for all $i$ with $recv[i] > 0$
11:     update the index sets (see Sect. 3.4)
12: end
```

to receive data items (line 4). The proof for deadlock-freedom applies to the basic algorithm and is independent from our modifications. The original proof for livelock-freedom assumes that all locations that become free (during the algorithm) can be used as free space. However, for the `MPI_Alltoallv` operation, only locations given by the initial index sets R_i can be used as free space. All locations that become free and do not belong to the initial index sets R_i cannot be used as free space. The original proof can be modified to distinguish between the usable and not-usable free locations, leading to the same result.

The additional available memory is used to create auxiliary buffers that are independent from the input buffer. These auxiliary buffers provide the additional free space that is required for the successful termination of the basic algorithm. The usage of the auxiliary buffers is independent from the rest of the algorithm and described in Sect. 3.5.

3.2 Initializing the Index Sets

The initial send index sets S_i are defined according to the given send counts and displacements: $S_i = \{sdispls[i], \ldots, sdispls[i] + scounts[i] - 1\}$. We assume that the indices of S_i are lower than the indices of S_{i+1} for $i = 1, \ldots, p-1$. Otherwise, a local reordering of the index sets is necessary. The receive index sets R_i are initialized analogously. Each send index set S_i is split into a finite number of disjoint subsets $S_i^1, \ldots, S_i^{n_i}$ with $S_i = S_i^1 \cup \ldots \cup S_i^{n_i}$. The splitting is performed at the positions given by the lowest and highest indices of the receive index sets. Each subset created contains contiguous indices and is either disjoint from all receive index sets or completely overlapped by one of the receive index sets. There are at most $2p$ possible splitting positions, since each receive index set provides two splitting positions. Thus the p initial send index sets can be split in at most $p + 2p$ subsets. We define two functions to specify how the send index subsets and the receive index sets overlap each other. A send subset is assigned a matching receive set and vice versa using the functions *rmatch* and *smatch*. For a send subset $S_i^k \neq \varnothing$, $k \in \{1, \ldots, n_i\}$, the matching receive set $rmatch(S_i^k)$

corresponds to the receive set R_j that overlaps with S_i^k. If there exists no such receive set then $rmatch(S_i^k) = \varnothing$. For a given receive set R_j, the matching send subset $smatch(R_j)$ corresponds to the send subset $S_i^k \neq \varnothing$ such that i and k are minimal and S_i^k is overlapped by R_j. If there exists no such send subset then $smatch(R_j) = \varnothing$. Figure 1 (left) shows an example for this initialization.

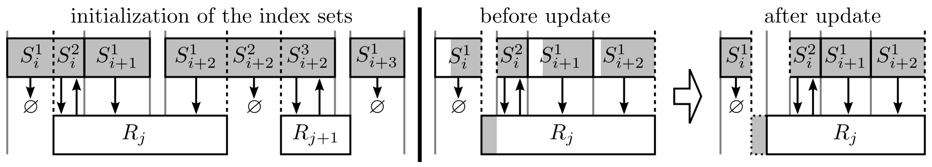

Figure 1. Examples for initialization (left) and update (right) of send and receive index sets. Arrows indicate matching sets, e.g. $\boxed{S_i^2} \rightarrow \boxed{R_j}$ corresponds to $rmatch(S_i^2) = R_j$.

3.3 Determine the Number of Data Items to be Received

For each message, the data items are received from the lowest to the highest indices. The number of data items that can be received from process j in free space at locations given by R_j is determined from the matching send subset of R_j and is stored in $recv[j]$.

$$recv[j] = \begin{cases} |R_j| & \text{if } smatch(R_j) = \varnothing \\ \min\{x|x \in smatch(R_j)\} - \min\{x|x \in R_j\} & \text{otherwise} \end{cases}$$

If no matching send subset exists, then all remaining data items from process j can be received. Otherwise, the number of data items that can be received is limited by the matching send subset $smatch(R_j)$. The lowest index of this subset corresponds to the lowest location of R_j that is not free. In addition to the data items that can be received in the input buffer, the free space that is available in the auxiliary buffers is used to receive additional data items.

3.4 Updating the Index Sets

The data items of the messages are sent and received from the lowest to the highest indices. All index sets are updated when the data items of the current phase are sent and received. Each receive set R_j is updated by removing the $recv[j]$ lowest indices, since they correspond to the data items that were previously received. Similarly, the $send[i]$ lowest indices are removed from the send subsets $S_i^1, \ldots, S_i^{n_i}$. Sending data items to other processes creates free space at the local process. However, only contiguous free space available at the lowest indices of a receive set can be used to receive data items in the next communication phase. For each receive set R_j, the free space corresponding to its indices is joined at its lowest indices. This is achieved by moving all data items that correspond to send subsets S_i^k with $rmatch(S_i^k) = R_j$ towards the highest indices of R_j (the index values of S_i^k are shifted accordingly). After that, data items from process j that are stored in the auxiliary buffers are moved to the freed space and R_j is updated again. Finally, the functions *rmatch* and *smatch* are adapted and for each send set S_i the value of $|S_i|$ is computed according to its updated subsets (only $|S_i|$ is required in Algorithm 1). Figure 1 (right) shows an example for this update procedure. The costs for updating the index sets depend on the number of data items that have to be moved. In the worst case, all data items of the remaining send subsets have to be moved during the update procedure in every communication phase.

3.5 Using Auxiliary Buffers

Additional memory of arbitrary size a is used to create auxiliary buffers on each process. These buffers are used to receive additional data items while their target locations are still occupied. The efficient management of the auxiliary buffers can have a significant influence on the performance. We use a static approach to create a fixed number of b auxiliary buffers, each of size $\frac{a}{b}$. Data items from process j can be stored in the (j mod b)-th auxiliary buffer using a *first-come, first-served* policy. This static partitioning of the additional memory allows a more flexible utilization in comparison to a single auxiliary buffer, but prevents fragmentation and overhead costs (e.g., for searching for free space). More advanced auxiliary buffer strategies (e.g., with dynamic heap-like allocations) can be subject of further optimizations.

4 Performance Results

We have performed experimental results for a Blue Gene/P system to investigate the performance of the proposed in-place algorithm for `MPI_Alltoallv`. The implementation uses the standard `MPI_Alltoallv` operation for exchanging the request messages, because using non-blocking communication for this exchange has caused performance problems for large numbers of processes (≥ 1024). In-place communication usually involves large messages that occupy a significant amount of main memory. Unless otherwise specified, each process uses 100 MB data that is randomly partitioned into blocks and sent to other processes. Results for the platform-specific `MPI_Alltoallv` operation are obtained using a separate (100 MB) receive buffer.

Figure 2 (left) shows communication times for different numbers of processes p depending on the number of auxiliary buffers b. The total size of additional memory used for the auxiliary buffers is 1 MB. Increasing the number of auxiliary buffers leads to a significant reduction in communication time, especially for large numbers of processes. Choosing the number of auxiliary buffers depending

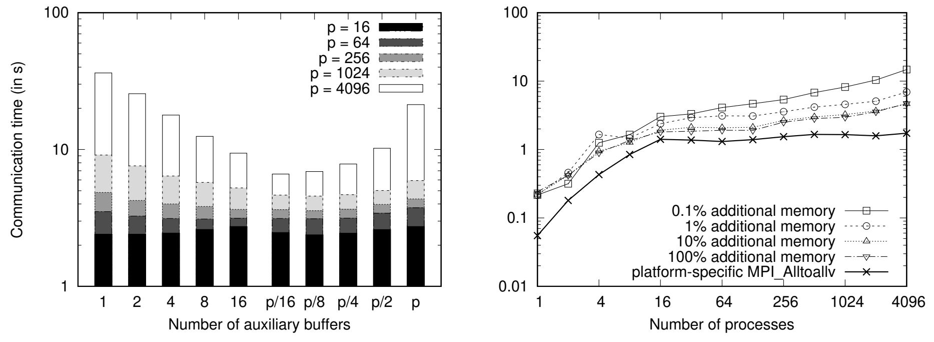

Figure 2. Communication times for in-place `MPI_Alltoallv` depending on the number of auxiliary buffers (left) and with different sizes of additional memory (right).

on the total number of processes shows good results for various values of p. For the following results we continue to use $b = \frac{p}{8}$. Figure 2 (right) shows communication times for different sizes of additional memory (in % with respect to the total message size of 100 MB) depending on the number processes. Increasing the additional memory up to 10 % leads to a significant performance improvement. A further increase up to 100% shows only small differences. This can be attributed to the static auxiliary buffer strategy. With 100 % additional memory, the sizes of the auxiliary buffers exceed the sizes of the messages to be received. This leads to an insufficient utilization of the additional memory. However, even with an optimal auxiliary buffer strategy there can be differences in performance in comparison to a platform-specific `MPI_Alltoallv` operation that includes optimizations for the specific system architecture [2]. For the following results we continue to use 1 % additional memory (1 MB).

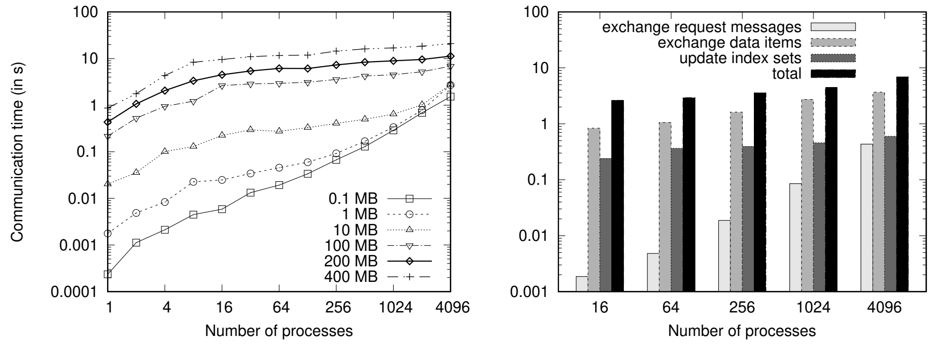

Figure 3. Communication times for in-place `MPI_Alltoallv` with different total message sizes (left). Times spend in the different parts of in-place `MPI_Alltoallv` (right).

Figure 3 (left) shows communication times for different total message sizes depending on the number of processes. The communication time with small messages strongly depends on the number of processes, while for large messages it increases more slowly. Figure 3 (right) shows the time spend on different parts of the in-place algorithm depending on the number of processes. The major part of the communication time is spent for exchanging the data items. The costs for exchanging the request messages are comparably small, but they increase with the number of processes. The costs for updating the index sets are also rather small. However, these costs strongly depend on the actual data redistribution problem. If the number of communication phases increases and a large number of data items need to be moved for each update, the total time of the in-place `MPI_Alltoallv` operation can be dominated by the local data movements.

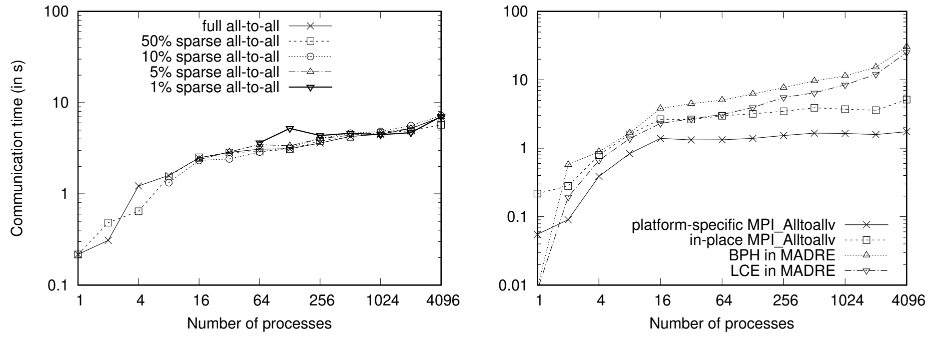

Figure 4. Communication times for in-place `MPI_Alltoallv` for sparse data redistribution schemes (left). Comparison of communication times for platform-specific `MPI_Alltoallv`, in-place `MPI_Alltoallv` and two MADRE algorithms (right).

Figure 4 (left) shows communication times for different sparse data redistribution schemes depending on the number of processes. For the sparse all-to-all communication, each process sends messages only to a limited number of random processes. The results show that the performance of the in-place algorithm is almost independent from the actual number of messages. Figure 4 (right) shows communication times for the platform-specific `MPI_Alltoallv` operation, the in-place algorithm, and for the Basic Pinar-Hendrickson algorithm (BPH) and the Local Copy Efficient algorithm (LCE) from the MADRE library [11] depending on the number of processes. MADRE and its in-place algorithms are designed to redistribute an arbitrary number of equal-sized (large) blocks according to a given destination rank and index for each block. To compare these algorithms to the `MPI_Alltoallv` operation, we increase the data item size up to 16 KB and treat each data item as a separate block in MADRE. Additional free blocks are used to provide the additional memory to the MADRE algorithms. There is a general increase in communication time for the MADRE algorithms depending on the number of processes. In comparison to that, the results for the platform-specific `MPI_Alltoallv` operation and the in-place algorithm are more stable. The communication time of the in-place algorithm is within a factor of three of the platform-specific `MPI_Alltoallv` operation.

5 Summary

In this paper, we have proposed an in-place algorithm for `MPI_Alltoallv` that performs data redistribution with limited memory. The size of required memory is independent from the message sizes and depends only linearly on the number of processes a single process has messages to exchange with. Additional memory can be used to improve the performance of the implementation. It is shown that the size of the additional memory and its efficient usage has a significant influence on the performance, especially for large numbers of processes. Performance results with large messages demonstrate the good performance of our approach.

Acknowledgment

The measurements are performed at the John von Neumann Institute for Computing, Jülich, Germany. `http://www.fz-juelich.de/nic`

References

1. MPI Forum: MPI: A Message-Passing Interface Standard Version 2.2. (2009)
2. Almási, G., Heidelberger, P., Archer, C.J., Martorell, X., Erway, C.C., Moreira, J.E., Steinmacher-Burow, B., Zheng, Y.: Optimization of MPI collective communication on BlueGene/L systems. In: Proc. of the 19th annual Int. Conf. on Supercomputing, ACM (2005) 253–262
3. Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Kumar, S., Lusk, E., Thakur, R., Träff, J.L.: MPI on a Million Processors. In: Proc. of the 16th EuroPVM/MPI Conf., Springer (2009) 20–30
4. Thakur, R., Choudhary, A., Ramanujam, J.: Efficient Algorithms for Array Redistribution. IEEE Trans. Parallel Distrib. Syst. **7**(6) (1996) 587–594
5. Walker, D.W., Otto, S.W.: Redistribution of block-cyclic data distributions using MPI. Concurrency - Practice and Experience **8**(9) (1996) 707–728
6. Lim, Y., Bhat, P., Prasanna, V.: Efficient Algorithms for Block-Cyclic Redistribution of Arrays. Algorithmica **24** (1999) 298–330
7. Pinar, A., Hendrickson, B.: Interprocessor Communication with Limited Memory. IEEE Trans. Parallel Distrib. Syst. **15**(7) (2004) 606–616
8. Siegel, S.F., Siegel, A.R.: MADRE: The Memory-Aware Data Redistribution Engine. Int. J. of High Performance Computing Applications **24** (2010) 93–104
9. Siegel, S.F., Siegel, A.R.: A Memory-Efficient Data Redistribution Algorithm. In: Proc. of the 16th EuroPVM/MPI Conf., Springer (2009) 219–229
10. Hofmann, M., Rünger, G.: Fine-Grained Data Distribution Operations for Particle Codes. In: Proc. of the 16th EuroPVM/MPI Conf., Springer (2009) 54–63
11. Siegel, S.F., Siegel, A.R.: MADRE: The Memory-Aware Data Redistribution Engine, Version 0.4 (2010) `http://vsl.cis.udel.edu/madre/`.